Faculté des Sciences & Techniques — Université de Limoges

Master 1ère année

Systèmes Embarqués

Corrections TP n°2

Programmation avec FreeRTOs

1 Utilisation des « *files de messages* »

a. Vous implémenterez sur votre ESP32c3 la correction de l'exercice 2 de la fiche de TD 2 :

```
#define TAILLE_MESSAGE 32


static const uint8_t msg_queue_len = 5;

static QueueHandle_t msg_queue;

void printMessages(void *parameters) {
  char buffer_message[32];

  while (1) {
  xQueueReceive(msg_queue, (void *)buffer_message, portMAX_DELAY);
  Serial.println(buffer_message);
  }
}

void sendMessage(void *paramaters) {
  int nb_messages = 0;
  char buffer_tache[32];

  while(1){
    sprintf(buffer_tache, "[Depuis 1 message %d]", ++nb_messages);
    xQueueSend(msg_queue, (void *)buffer_tache, portMAX_DELAY);
    vTaskDelay(1000 / portTICK_PERIOD_MS);
  }
}

void sendMessage2(void *paramaters) {
  int nb_messages = 0;
  char buffer_tache[32];


  while(1){
    sprintf(buffer_tache, "[Depuis 2 message %d]", ++nb_messages);
    xQueueSend(msg_queue, (void *)buffer_tache, portMAX_DELAY);
    vTaskDelay(500 / portTICK_PERIOD_MS);
  }
}

void setup() {

  // Configure Serial
  Serial.begin(115200);

  // Wait a moment to start (so we don't miss Serial output)
  vTaskDelay(1000 / portTICK_PERIOD_MS);
  Serial.println();
  Serial.println("---FreeRTOS Queue Demo---");

  // Create queue
  msg_queue = xQueueCreate(msg_queue_len, TAILLE_MESSAGE*sizeof(char));

  // Start print task
  xTaskCreate(sendMessage, "Send Message", 2048, NULL, 1, NULL);
  xTaskCreate(sendMessage2,"Send Message", 2048, NULL, 1, NULL);
  xTaskCreate(printMessages,"Print Messages", 1024, NULL, 1, NULL);
}

void loop() {

}
```

b. Est-ce qu'il peut y avoir des problèmes pour les buffer gérés par chaque tâche ?
Il peut y avoir des problèmes de place pour le contenu du message, si la taille de message dépasse la taille allouée plus le caractère zéro `'\0'` indiquant la fin de la chaîne.
Il peut y avoir également des blocages si la file de messages est pleine, c-à-d si la tâche chargée de lire les messages de la file ne le fait pas assez vite ou si elle est suspendue ou de priorité inférieure à la priorité des tâches qui écrivent.
Il ne peut, en revanche, y avoir de problème de corruption de la file de message du fait que sa manipulation est réalisée par le RTOS.

c. Rajoutez la possibilité d'identifier la **provenance du message**.

```
#define TAILLE_MESSAGE 32

static const uint8_t msg_queue_len = 5;
static QueueHandle_t msg_queue;

typedef struct {
char contenu[TAILLE_MESSAGE];
int provenance = 0;
} Message;

void printMessages(void *parameters) {
  Message un_message;

  while (1) {
  xQueueReceive(msg_queue, (void *)&un_message, portMAX_DELAY);
  Serial.printf("%s depuis %d\n",un_message.contenu, un_message.provenance);
  }
}

void sendMessage(void *paramaters) {
  int nb_messages = 0;
  Message message_a_envoyer;
  message_a_envoyer.provenance = 1;

  while(1){
    sprintf(message_a_envoyer.contenu, "[message %d]", ++nb_messages);
    xQueueSend(msg_queue, (void *)&message_a_envoyer, portMAX_DELAY);
    vTaskDelay(1000 / portTICK_PERIOD_MS);
  }
}

void sendMessage2(void *paramaters) {
  int nb_messages = 0;
  Message message_a_envoyer;
  message_a_envoyer.provenance = 2;

  while(1){
    sprintf(message_a_envoyer.contenu, "[message %d]", ++nb_messages);
    xQueueSend(msg_queue, (void *)&message_a_envoyer, portMAX_DELAY);
    vTaskDelay(500 / portTICK_PERIOD_MS);
  }
}

void setup() {

  // Configure Serial
  Serial.begin(115200);

  // Wait a moment to start (so we don't miss Serial output)
  vTaskDelay(1000 / portTICK_PERIOD_MS);
  Serial.println();

  // Create queue
  msg_queue = xQueueCreate(msg_queue_len, sizeof(Message));

  // Start print task
  xTaskCreate(sendMessage, "Send Message", 2048, NULL, 1, NULL);
  xTaskCreate(sendMessage2,"Send Message 2", 2048, NULL, 1, NULL);
  xTaskCreate(printMessages,"Print Messages", 2048, NULL, 1, NULL);
}

void loop() {
}
```

Vous noterez que la taille de la pile allouée à la tâche chargée de recevoir les messages et de les afficher est doublée pour éviter des débordements de pile entrainant le redémarrage du SoC.

d. On veut rajouter un indicateur lumineux sur l'ESP32c3 :
   - ◇ rouge quand la file de message est vide ;
   - ◇ verte quand la tâche « *sendMessage* » envoi un message ;
   - ◇ bleue qaund la tâche « *sendMessage2* » envoi un message.

   Proposez **deux versions différentes** utilisant des outils différents proposés par FreeRTOs pour mettre en œuvre cette modification.
   En première solution, on peut faire en sorte que ce soit lors de l'envoi du message que la couleur de la LED est changée.
   ⟹la LED est une ressource critique dont l'accès est partagé entre deux tâches, il faut un mutex pour en protéger l'accès.

```
#include "Freenove_WS2812_Lib_for_ESP32.h"

#define LEDS_COUNT  1
#define LEDS_PIN    8
#define CHANNEL     0

#define RED 255, 0, 0
#define GREEN 0, 255, 0
#define BLUE 0, 0, 255

#define TAILLE_MESSAGE 32

Freenove_ESP32_WS2812 strip = Freenove_ESP32_WS2812(LEDS_COUNT, LEDS_PIN, CHANNEL,
TYPE_GRB);
static SemaphoreHandle_t mutex; // Mutex pour l'acces a la LED
static const uint8_t msg_queue_len = 5;

static QueueHandle_t msg_queue;

typedef struct {
char contenu[TAILLE_MESSAGE];
int provenance = 0;
} Message;

void printMessages(void *parameters) {
  Message un_message;

  while (1) {
  xQueueReceive(msg_queue, (void *)&un_message, portMAX_DELAY);
  Serial.printf("%s depuis %d\n\n",un_message.contenu, un_message.provenance);
  xSemaphoreTake(mutex, portMAX_DELAY);
  strip.setLedColorData(0, RED);
  strip.show();
  vTaskDelay(100/portTICK_PERIOD_MS);
  xSemaphoreGive(mutex);
  }
}

void sendMessage(void *paramaters) {
  int nb_messages = 0;
  Message message_a_envoyer;
  message_a_envoyer.provenance = 1;

  while(1){
    sprintf(message_a_envoyer.contenu, "[message %d]", ++nb_messages);
    xSemaphoreTake(mutex, portMAX_DELAY);
    xQueueSend(msg_queue, (void *)&message_a_envoyer, portMAX_DELAY);
    strip.setLedColorData(0,GREEN);
    strip.show();
    vTaskDelay(200 / portTICK_PERIOD_MS);
    strip.setLedColorData(0,0,0,0);
    strip.show();
    vTaskDelay(100 / portTICK_PERIOD_MS);
    xSemaphoreGive(mutex);
    vTaskDelay(4000 / portTICK_PERIOD_MS);
  }
}

void sendMessage2(void *paramaters) {
  int nb_messages = 0;
  Message message_a_envoyer;
  message_a_envoyer.provenance = 2;

  vTaskDelay(2000 / portTICK_PERIOD_MS);
  while(1){
    sprintf(message_a_envoyer.contenu, "[message %d]", ++nb_messages);
```

```
    xSemaphoreTake(mutex, portMAX_DELAY);
    xQueueSend(msg_queue, (void *)&message_a_envoyer, portMAX_DELAY);
    strip.setLedColorData(0,BLUE);
    strip.show();
    vTaskDelay(200 / portTICK_PERIOD_MS);
    strip.setLedColorData(0,0,0,0);
    strip.show();
    vTaskDelay(100 / portTICK_PERIOD_MS);
    xSemaphoreGive(mutex);
    vTaskDelay(8000 / portTICK_PERIOD_MS);
  }
}

void setup() {

  // Configure Serial
  Serial.begin(115200);
  mutex = xSemaphoreCreateMutex();
  strip.begin();
  strip.setBrightness(10);
  // Wait a moment to start (so we don't miss Serial output)
  vTaskDelay(1000 / portTICK_PERIOD_MS);

  Serial.printf("Demo\n");
  Serial.flush();
  // Create queue
  msg_queue = xQueueCreate(msg_queue_len, sizeof(Message));

  // Start print task
  xTaskCreate(printMessages,"Print Messages", 4096, NULL, 1, NULL);
  xTaskCreate(sendMessage, "Send Message", 2048, NULL, 1, NULL);
  xTaskCreate(sendMessage2,"Send Message 2", 2048, NULL, 1, NULL);
  // Terminate Setup & loop task
  vTaskDelete(NULL);

}

void loop() {

}
```

Les 3 tâches entrent en compétition pour l'accès à la LED : un mutex protège cet accès.
Il est nécessaire d'introduire un délai pour que la bibliothèque et le hardware de la LED fonctionne correctement.
Le port série est aussi une ressource critique, mais seule la tâche `printMessages()` *y accède.*
On peut également gérer la LED lors de la réception :

```
#include "Freenove_WS2812_Lib_for_ESP32.h"

#define LEDS_COUNT  1
#define LEDS_PIN    8
#define CHANNEL     0

#define RED 255, 0, 0
#define GREEN 0, 255, 0
#define BLUE 0, 0, 255

#define TAILLE_MESSAGE 32

Freenove_ESP32_WS2812 strip = Freenove_ESP32_WS2812(LEDS_COUNT, LEDS_PIN, CHANNEL,
TYPE_GRB);

static const uint8_t msg_queue_len = 5;

static QueueHandle_t msg_queue;

typedef struct {
char contenu[TAILLE_MESSAGE];
int provenance = 0;
} Message;

void printMessages(void *parameters) {
  Message un_message;
  bool pas_de_message = true;

  while (1) {
    if (xQueueReceive(msg_queue, (void *)&un_message, 0) == pdTRUE)
    {
      Serial.printf("%s depuis %d\n\n",un_message.contenu, un_message.provenance);
      pas_de_message = false;
```

```
      if (un_message.provenance == 1) {
        strip.setLedColorData(0,GREEN);
        strip.show();
      }
      else
      {
        strip.setLedColorData(0,BLUE);
        strip.show();
      }
      vTaskDelay(500 / portTICK_PERIOD_MS);
      strip.setLedColorData(0,0,0,0);
      strip.show();
    }
    else
    {
      if (!pas_de_message){
      strip.setLedColorData(0, RED);
      strip.show();
      pas_de_message = true;
      }
    }
    vTaskDelay(500/portTICK_PERIOD_MS);
  }
}

void sendMessage(void *paramaters) {
  int nb_messages = 0;
  Message message_a_envoyer;
  message_a_envoyer.provenance = 1;

  while(1){
    sprintf(message_a_envoyer.contenu, "[message %d]", ++nb_messages);
    xQueueSend(msg_queue, (void *)&message_a_envoyer, portMAX_DELAY);
    vTaskDelay(4000 / portTICK_PERIOD_MS);
  }
}

void sendMessage2(void *paramaters) {
  int nb_messages = 0;
  Message message_a_envoyer;
  message_a_envoyer.provenance = 2;

  vTaskDelay(2000 / portTICK_PERIOD_MS);
  while(1){
    sprintf(message_a_envoyer.contenu, "[message %d]", ++nb_messages);
    xQueueSend(msg_queue, (void *)&message_a_envoyer, portMAX_DELAY);
    vTaskDelay(8000 / portTICK_PERIOD_MS);
  }
}

void setup() {

  // Configure Serial
  Serial.begin(115200);
  strip.begin();
  strip.setBrightness(10);
  // Wait a moment to start (so we don't miss Serial output)
  vTaskDelay(1000 / portTICK_PERIOD_MS);

  Serial.printf("Demo\n");
  Serial.flush();
  // Create queue
  msg_queue = xQueueCreate(msg_queue_len, sizeof(Message));

  // Start print task
  xTaskCreate(printMessages,"Print Messages", 4096, NULL, 1, NULL);
  xTaskCreate(sendMessage, "Send Message", 2048, NULL, 1, NULL);
  xTaskCreate(sendMessage2,"Send Message 2", 2048, NULL, 1, NULL);
  // Terminate Setup & loop task
  vTaskDelete(NULL);

}

void loop() {

}
```

Ici, il n'y a que la tâche `printMessages()` qui accède à la LED et au port série, alors il n'y a pas besoin de mutex.

e. Est-ce que les couleurs sont discernables pour un humain ?
Quelle pourrait être une alternative ?
Si les échanges de messages sont trop rapides, les couleurs se superposent à cause de la persistence de vision.
On peut améliorer la reconnaissance des couleurs en introduisant des délais, mais cela ralentit inutilement le programme.
⟹Utiliser des LEDs pour indiquer un état du SoC pour faire du « log » n'est pas une bonne solution. Par contre, si le but est d'informer l'utilisateur, il faut prévoir des délais pour les échanges et pour les signaux lumineux : allumage suivi d'extinction.
On peut également utiliser des couleurs dont le mélange est discernable et compréhensible par l'utilisateur : par exemple du mauve pour du bleu et du rouge simultanément.
Par contre, il faut bien choisir les niveaux de couleurs RGB pour avoir un bon rendu et éventuellement, mettre un diffuseur sur la LED pour améliorer le rendu du mélange.

f. Rajoutez **l'affichage d'un message spécial** lié à l'appui du **bouton**.
La couleur associée au bouton est blanc.

```
#include "Freenove_WS2812_Lib_for_ESP32.h"
#define BUTTONPIN 9

#define LEDS_COUNT  1
#define LEDS_PIN    8
#define CHANNEL     0

#define RED 255, 0, 0
#define GREEN 0, 255, 0
#define BLUE 0, 0, 255
#define WHITE 255, 255, 255

#define TAILLE_MESSAGE 32

Freenove_ESP32_WS2812 strip = Freenove_ESP32_WS2812(LEDS_COUNT, LEDS_PIN, CHANNEL,
TYPE_GRB);
static SemaphoreHandle_t mutex;
static SemaphoreHandle_t bouton;

static const uint8_t msg_queue_len = 5;

static QueueHandle_t msg_queue;

typedef struct {
char contenu[TAILLE_MESSAGE];
int provenance = 0;
} Message;

void ARDUINO_ISR_ATTR isr(void* arg) {
  BaseType_t xHigherPriorityTaskWoken;
  // We have not woken a task at the start of the ISR.
  xHigherPriorityTaskWoken = pdTRUE;
  xSemaphoreGiveFromISR(bouton, &xHigherPriorityTaskWoken);
}

void bouton_led(void *parameters)
{
  while(1)
  {
    xSemaphoreTake(bouton, portMAX_DELAY);
    //xSemaphoreTake(mutex, portMAX_DELAY);
    Serial.printf("Bouton !\n\n");
    strip.setLedColorData(0, WHITE);
    strip.show();
    vTaskDelay(100/portTICK_PERIOD_MS);
    //xSemaphoreGive(mutex);
  }
}
void printMessages(void *parameters) {
  Message un_message;

  while (1) {
  xQueueReceive(msg_queue, (void *)&un_message, portMAX_DELAY);
  xSemaphoreTake(mutex, portMAX_DELAY);
  Serial.printf("%s depuis %d\n\n",un_message.contenu, un_message.provenance);
  strip.setLedColorData(0, RED);
  strip.show();
  vTaskDelay(100/portTICK_PERIOD_MS);
  xSemaphoreGive(mutex);
```

```
  }
}

void sendMessage(void *paramaters) {
  int nb_messages = 0;
  Message message_a_envoyer;
  message_a_envoyer.provenance = 1;

  while(1){
    sprintf(message_a_envoyer.contenu, "[message %d]", ++nb_messages);
    xSemaphoreTake(mutex, portMAX_DELAY);
    xQueueSend(msg_queue, (void *)&message_a_envoyer, portMAX_DELAY);
    strip.setLedColorData(0,GREEN);
    strip.show();
    vTaskDelay(200 / portTICK_PERIOD_MS);
    strip.setLedColorData(0,0,0,0);
    strip.show();
    vTaskDelay(100 / portTICK_PERIOD_MS);
    xSemaphoreGive(mutex);
    vTaskDelay(4000 / portTICK_PERIOD_MS);
  }
}

void sendMessage2(void *paramaters) {
  int nb_messages = 0;
  Message message_a_envoyer;
  message_a_envoyer.provenance = 2;

  vTaskDelay(2000 / portTICK_PERIOD_MS);
  while(1){
    sprintf(message_a_envoyer.contenu, "[message %d]", ++nb_messages);
    xSemaphoreTake(mutex, portMAX_DELAY);
    xQueueSend(msg_queue, (void *)&message_a_envoyer, portMAX_DELAY);
    strip.setLedColorData(0,BLUE);
    strip.show();
    vTaskDelay(200 / portTICK_PERIOD_MS);
    strip.setLedColorData(0,0,0,0);
    strip.show();
    vTaskDelay(100 / portTICK_PERIOD_MS);
    xSemaphoreGive(mutex);
    vTaskDelay(8000 / portTICK_PERIOD_MS);
  }
}

void setup() {

  // Configure Serial
  Serial.begin(115200);
  mutex = xSemaphoreCreateMutex();
  bouton = xSemaphoreCreateBinary();

  strip.begin();
  strip.setBrightness(10);
  pinMode(BUTTONPIN, INPUT_PULLUP);
  attachInterruptArg(digitalPinToInterrupt(BUTTONPIN), isr, NULL, RISING);
  // Wait a moment to start (so we don't miss Serial output)
  vTaskDelay(1000 / portTICK_PERIOD_MS);

  Serial.printf("Demo\n");
  Serial.flush();
  // Create queue
  msg_queue = xQueueCreate(msg_queue_len, sizeof(Message));

  // Start print task
  xTaskCreate(printMessages,"Print Messages", 4096, NULL, 1, NULL);
  xTaskCreate(sendMessage, "Send Message", 2048, NULL, 1, NULL);
  xTaskCreate(sendMessage2,"Send Message 2", 2048, NULL, 1, NULL);
  xTaskCreate(bouton_led,"Bouton", 2048, NULL, 2, NULL);
  // Terminate Setup & loop task
  vTaskDelete(NULL);

}

void loop() {

}
```

Ici, on a besoin d'une tâche supplémentaire `bouton_led()` *pour gérer le bouton.*
On va donner une priorité supérieure à cette tâche : elle ne pourra pas être interrompue par les autres tâches ⟹ il n'est pas nécessaire de lui faire utiliser le mutex d'accès à la LED et au port série.

L'interruption du bouton va déclencher cette tâche : il faut qu'elle libère une Sémaphore qui doit être créer obligatoirement par `xSemaphoreCreateBinary()` pour que cela fonctionne.

Et pour la version alternative :

```
#include "Freenove_WS2812_Lib_for_ESP32.h"

#define BUTTONPIN   9

#define LEDS_COUNT  1
#define LEDS_PIN    8
#define CHANNEL     0

#define RED 255, 0, 0
#define GREEN 0, 255, 0
#define BLUE 0, 0, 255
#define WHITE 255, 255, 255

#define TAILLE_MESSAGE 32

Freenove_ESP32_WS2812 strip = Freenove_ESP32_WS2812(LEDS_COUNT, LEDS_PIN, CHANNEL,
TYPE_GRB);

static const uint8_t msg_queue_len = 5;

static QueueHandle_t msg_queue;

typedef struct {
char contenu[TAILLE_MESSAGE];
int provenance = 0;
} Message;

void ARDUINO_ISR_ATTR isr(void* arg) {
  BaseType_t xHigherPriorityTaskWoken;
  // We have not woken a task at the start of the ISR.
  xHigherPriorityTaskWoken = pdFALSE;
  static Message m;
  m.provenance = 3;
  sprintf(m.contenu, "bouton !");
  xQueueSendFromISR(msg_queue, (void *)&m, &xHigherPriorityTaskWoken);
}

void printMessages(void *parameters) {
  Message un_message;
  bool pas_de_message = true;

  while (1) {
    if (xQueueReceive(msg_queue, (void *)&un_message, 0) == pdTRUE)
    {
      Serial.printf("%s depuis %d\n\n",un_message.contenu, un_message.provenance);
      pas_de_message = false;
      int source = un_message.provenance;
      switch(un_message.provenance)
      {
        case 1:
          strip.setLedColorData(0,GREEN);
          strip.show();
          break;
        case 2:
          strip.setLedColorData(0,BLUE);
          strip.show();
          break;
        case 3:
          strip.setLedColorData(0, WHITE);
          strip.show();
      }
      vTaskDelay(500 / portTICK_PERIOD_MS);
      strip.setLedColorData(0,0,0,0);
      strip.show();
    }
    else
    {
      if (!pas_de_message){
      strip.setLedColorData(0, RED);
      strip.show();
      pas_de_message = true;
      }
    }
    vTaskDelay(500/portTICK_PERIOD_MS);
  }
}
```

```
void sendMessage(void *paramaters) {
  int nb_messages = 0;
  Message message_a_envoyer;
  message_a_envoyer.provenance = 1;

  while(1){
    sprintf(message_a_envoyer.contenu, "[message %d]", ++nb_messages);
    xQueueSend(msg_queue, (void *)&message_a_envoyer, portMAX_DELAY);
    vTaskDelay(4000 / portTICK_PERIOD_MS);
  }
}

void sendMessage2(void *paramaters) {
  int nb_messages = 0;
  Message message_a_envoyer;
  message_a_envoyer.provenance = 2;

  vTaskDelay(2000 / portTICK_PERIOD_MS);
  while(1){
    sprintf(message_a_envoyer.contenu, "[message %d]", ++nb_messages);
    xQueueSend(msg_queue, (void *)&message_a_envoyer, portMAX_DELAY);
    vTaskDelay(8000 / portTICK_PERIOD_MS);
  }
}

void setup() {

  // Configure Serial
  Serial.begin(115200);
  strip.begin();
  strip.setBrightness(10);
  pinMode(BUTTONPIN, INPUT_PULLUP);
  attachInterruptArg(digitalPinToInterrupt(BUTTONPIN), isr, NULL, RISING);
  // Wait a moment to start (so we don't miss Serial output)
  vTaskDelay(1000 / portTICK_PERIOD_MS);

  Serial.printf("Demo\n");
  Serial.flush();
  // Create queue
  msg_queue = xQueueCreate(msg_queue_len, sizeof(Message));

  // Start print task
  xTaskCreate(printMessages,"Print Messages", 4096, NULL, 1, NULL);
  xTaskCreate(sendMessage, "Send Message", 2048, NULL, 1, NULL);
  xTaskCreate(sendMessage2,"Send Message 2", 2048, NULL, 1, NULL);
  // Terminate Setup & loop task
  vTaskDelete(NULL);

}

void loop() {

}
```

Dans cette version, l'interruption lié au bouton envoi un message avec une nouvelle provenance ce qui permet sa gestion facile dans la tâche `printMessages()`, il n'y a donc pas besoin de mutex ni de nouvelle tâche.

g. On veut maintenant que le bouton **réinitialise les compteurs** associés aux deux tâches.
Donnez une solution correcte à ce problème.

```
#include "Freenove_WS2812_Lib_for_ESP32.h"
#define BUTTONPIN 9

#define LEDS_COUNT  1
#define LEDS_PIN    8
#define CHANNEL     0

#define RED 255, 0, 0
#define GREEN 0, 255, 0
#define BLUE 0, 0, 255
#define WHITE 255, 255, 255

#define TAILLE_MESSAGE 32

Freenove_ESP32_WS2812 strip = Freenove_ESP32_WS2812(LEDS_COUNT, LEDS_PIN, CHANNEL,
TYPE_GRB);
static SemaphoreHandle_t mutex;
static SemaphoreHandle_t bouton;

static const uint8_t msg_queue_len = 5;

static QueueHandle_t msg_queue;

int nb_messages_task1 = 0;
int nb_messages_task2 = 0;

typedef struct {
char contenu[TAILLE_MESSAGE];
int provenance = 0;
} Message;

void ARDUINO_ISR_ATTR isr(void* arg) {
  BaseType_t xHigherPriorityTaskWoken;
  // We have not woken a task at the start of the ISR.
  xHigherPriorityTaskWoken = pdTRUE;
  xSemaphoreGiveFromISR(bouton, &xHigherPriorityTaskWoken);
}

void bouton_led(void *parameters)
{
  while(1)
  {
    xSemaphoreTake(bouton, portMAX_DELAY);
    //xSemaphoreTake(mutex, portMAX_DELAY);
    nb_messages_task1 = 0;
    nb_messages_task2 = 0;
    Serial.printf("Bouton !\n\n");
    strip.setLedColorData(0, WHITE);
    strip.show();
    vTaskDelay(100/portTICK_PERIOD_MS);
    //xSemaphoreGive(mutex);
  }
}
void printMessages(void *parameters) {
  Message un_message;

  while (1) {
  xQueueReceive(msg_queue, (void *)&un_message, portMAX_DELAY);
  xSemaphoreTake(mutex, portMAX_DELAY);
  Serial.printf("%s depuis %d\n\n",un_message.contenu, un_message.provenance);
  strip.setLedColorData(0, RED);
  strip.show();
  vTaskDelay(100/portTICK_PERIOD_MS);
  xSemaphoreGive(mutex);
  }
}

void sendMessage(void *paramaters) {
  nb_messages_task1 = 0;
  Message message_a_envoyer;
  message_a_envoyer.provenance = 1;

  while(1){
    xSemaphoreTake(mutex, portMAX_DELAY);
    sprintf(message_a_envoyer.contenu, "[message %d]", ++nb_messages_task1);
    xQueueSend(msg_queue, (void *)&message_a_envoyer, portMAX_DELAY);
    strip.setLedColorData(0,GREEN);
    strip.show();
```

```
    vTaskDelay(200 / portTICK_PERIOD_MS);
    strip.setLedColorData(0,0,0,0);
    strip.show();
    vTaskDelay(100 / portTICK_PERIOD_MS);
    xSemaphoreGive(mutex);
    vTaskDelay(4000 / portTICK_PERIOD_MS);
  }
}

void sendMessage2(void *paramaters) {
  nb_messages_task2 = 0;
  Message message_a_envoyer;
  message_a_envoyer.provenance = 2;

  vTaskDelay(2000 / portTICK_PERIOD_MS);
  while(1){
    xSemaphoreTake(mutex, portMAX_DELAY);
    sprintf(message_a_envoyer.contenu, "[message %d]", ++nb_messages_task2);
    xQueueSend(msg_queue, (void *)&message_a_envoyer, portMAX_DELAY);
    strip.setLedColorData(0,BLUE);
    strip.show();
    vTaskDelay(200 / portTICK_PERIOD_MS);
    strip.setLedColorData(0,0,0,0);
    strip.show();
    vTaskDelay(100 / portTICK_PERIOD_MS);
    xSemaphoreGive(mutex);
    vTaskDelay(8000 / portTICK_PERIOD_MS);
  }
}

void setup() {

  // Configure Serial
  Serial.begin(115200);
  mutex = xSemaphoreCreateMutex();
  bouton = xSemaphoreCreateBinary();

  strip.begin();
  strip.setBrightness(10);
  pinMode(BUTTONPIN, INPUT_PULLUP);
  attachInterruptArg(digitalPinToInterrupt(BUTTONPIN), isr, NULL, RISING);
  // Wait a moment to start (so we don't miss Serial output)
  vTaskDelay(1000 / portTICK_PERIOD_MS);

  Serial.printf("Demo\n");
  Serial.flush();
  // Create queue
  msg_queue = xQueueCreate(msg_queue_len, sizeof(Message));

  // Start print task
  xTaskCreate(printMessages,"Print Messages", 4096, NULL, 1, NULL);
  xTaskCreate(sendMessage, "Send Message", 2048, NULL, 1, NULL);
  xTaskCreate(sendMessage2,"Send Message 2", 2048, NULL, 1, NULL);
  xTaskCreate(bouton_led,"Bouton", 2048, NULL, 2, NULL);
  // Terminate Setup & loop task
  vTaskDelete(NULL);

}

void loop() {

}
```

Il faut externaliser les compteurs des deux tâches `sendMessage()` et `sendMessage2()`.

Pour en protéger l'accès, on peut inclure leur modification dans la section critique associée au mutex de protection d'accès à la LED.

Néanmoins, en donnant une priorité supérieure à la tâche `bouton_led()` qui est déclenchée par l'interruption liée au bouton, on a pas besoin de protection par mutex pour la réinitialisation des compteurs : la tâche `bouton_led()` n'est pas interruptible par une autre tâche qui voudrait avoir accès aux compteurs.

Et pour la version alternative :

```
#include "Freenove_WS2812_Lib_for_ESP32.h"

#define BUTTONPIN   9

#define LEDS_COUNT  1
#define LEDS_PIN    8
#define CHANNEL     0

#define RED 255, 0, 0
#define GREEN 0, 255, 0
#define BLUE 0, 0, 255
#define WHITE 255, 255, 255

#define TAILLE_MESSAGE 32

Freenove_ESP32_WS2812 strip = Freenove_ESP32_WS2812(LEDS_COUNT, LEDS_PIN, CHANNEL,
TYPE_GRB);

static const uint8_t msg_queue_len = 5;

static QueueHandle_t msg_queue;

int nb_messages_task1 = 0;
int nb_messages_task2 = 0;

typedef struct {
char contenu[TAILLE_MESSAGE];
int provenance = 0;
} Message;

void ARDUINO_ISR_ATTR isr(void* arg) {
  BaseType_t xHigherPriorityTaskWoken;
  // We have not woken a task at the start of the ISR.
  xHigherPriorityTaskWoken = pdFALSE;
  static Message m;
  m.provenance = 3;
  sprintf(m.contenu, "bouton !");
  nb_messages_task1 = 0;
  nb_messages_task2 = 0;
  xQueueSendFromISR(msg_queue, (void *)&m, &xHigherPriorityTaskWoken);
}

void printMessages(void *parameters) {
  Message un_message;
  bool pas_de_message = true;

  while (1) {
    if (xQueueReceive(msg_queue, (void *)&un_message, 0) == pdTRUE)
    {
      //xQueueReceive(msg_queue, (void *)&un_message, portMAX_DELAY);
      Serial.printf("%s depuis %d\n\n",un_message.contenu, un_message.provenance);
      pas_de_message = false;
      int source = un_message.provenance;
      switch(un_message.provenance)
      {
        case 1:
          strip.setLedColorData(0,GREEN);
          strip.show();
          break;
        case 2:
          strip.setLedColorData(0,BLUE);
          strip.show();
          break;
        case 3:
          strip.setLedColorData(0, WHITE);
          strip.show();
      }
      vTaskDelay(500 / portTICK_PERIOD_MS);
      strip.setLedColorData(0,0,0,0);
      strip.show();
    }
    else
    {
      if (!pas_de_message){
      strip.setLedColorData(0, RED);
      strip.show();
      pas_de_message = true;
      }
    }
```

```
    vTaskDelay(500/portTICK_PERIOD_MS);
  }
}

void sendMessage(void *paramaters) {
  nb_messages_task1 = 0;
  Message message_a_envoyer;
  message_a_envoyer.provenance = 1;

  while(1){
    sprintf(message_a_envoyer.contenu, "[message %d]", ++nb_messages_task1);
    xQueueSend(msg_queue, (void *)&message_a_envoyer, portMAX_DELAY);
    vTaskDelay(4000 / portTICK_PERIOD_MS);
  }
}

void sendMessage2(void *paramaters) {
  nb_messages_task2 = 0;
  Message message_a_envoyer;
  message_a_envoyer.provenance = 2;

  vTaskDelay(2000 / portTICK_PERIOD_MS);
  while(1){
    sprintf(message_a_envoyer.contenu, "[message %d]", ++nb_messages_task2);
    xQueueSend(msg_queue, (void *)&message_a_envoyer, portMAX_DELAY);
    vTaskDelay(8000 / portTICK_PERIOD_MS);
  }
}

void setup() {

  // Configure Serial
  Serial.begin(115200);
  strip.begin();
  strip.setBrightness(10);
  pinMode(BUTTONPIN, INPUT_PULLUP);
  attachInterruptArg(digitalPinToInterrupt(BUTTONPIN), isr, NULL, RISING);
  // Wait a moment to start (so we don't miss Serial output)
  vTaskDelay(1000 / portTICK_PERIOD_MS);

  Serial.printf("Demo\n");
  Serial.flush();
  // Create queue
  msg_queue = xQueueCreate(msg_queue_len, sizeof(Message));

  // Start print task
  xTaskCreate(printMessages,"Print Messages", 4096, NULL, 1, NULL);
  xTaskCreate(sendMessage, "Send Message", 2048, NULL, 1, NULL);
  xTaskCreate(sendMessage2,"Send Message 2", 2048, NULL, 1, NULL);
  // Terminate Setup & loop task
  vTaskDelete(NULL);

}

void loop() {

}
```

Dans cette version, on externalise les compteurs également.
Le travail de réinitialisation est très court : il est inclus dans le travail de l'interruption et il n'y a pas besoin de mutex pour la protection, car la priorité de l'interruption est la plus haute et ne peut donc pas être interrompue par une des tâches `sendMessage` *et* `sendMessage2()`.

h. Observez si **tout fonctionne bien** lors de l'appui répété et rapide du bouton.
Avez vous bien géré la réinitialisation des compteurs ?
Oui, comme expliqué précédemment pour les deux versions proposées.