CR TP FreeRTOS

2024 - **Télécom Nancy**

Projet de système temps réel - Gestion de capteurs par des ESP32

Léo FORNOFF
Elise NEYENS

Projet réalisé dans le cours, de Systèmes Embarqués Temps Réels (SETR), de V. Bombardier, pour les élèves de Système Logiciel Embarqué de deuxième année à Télécom Nancy, composante de l'Université de Lorraine.

Mots clés :

ESP32 - Espressif - FreeRTOS - Framework Arduino - Tâches - Langage C/C++ - Température - DHT22 - Humidité - DHT22 - CO2 - MH-Z18B - UART - Platform IO - Modèle producteur consommateur

Descriptif du projet

1. Résumé

Ce projet s'inscrit dans le parcours d'un élève en 2ème année d'école d'ingénieur et est à but pédagogique. Il s'inscrit dans le cours de SETR.
L'objectif de ce projet est de réaliser un programme permettant à deux ESP32 d'interagir entre eux ainsi qu'avec trois capteur : humidité/température, CO_2 et distance. L'un des deux ESP étant équipé d'un écran OLED, il est aussi demandé d'afficher les données de ces capteurs sur l'écran.
La gestion des capteurs devra être effectué avec des tâches, de même que la connexion UART entre les deux microprocesseurs.

Il est important d'alimenter les deux (2) ESP32 pour faire fonctionner le système

2. Cahier des charges

- Tester les capteurs individuellement
- Utiliser l'ESP-32 OLED pour la température (Framework Arduino)
- Utiliser l'ESP32 pour le CO² (Framework IdF)
- Synchroniser les 2 ESP (communication par RS232 : Pins 25 - 26)
- Option: déclencher par détection de présence (HC – SR04)
- Mettre en œuvre le modèle Producteurs Multiples (CO²/ Temp/Hum) – Consommateur unique (Affichage)
- Utiliser « à bon escient » les outils d'exclusion mutuelle et de synchronisation
- Rédiger un CR détaillant les tâches et leur liens (avec modèle SART si possible)
- Déposer le projet sur Arche

3. Schéma récapitulatif du montage souhaité

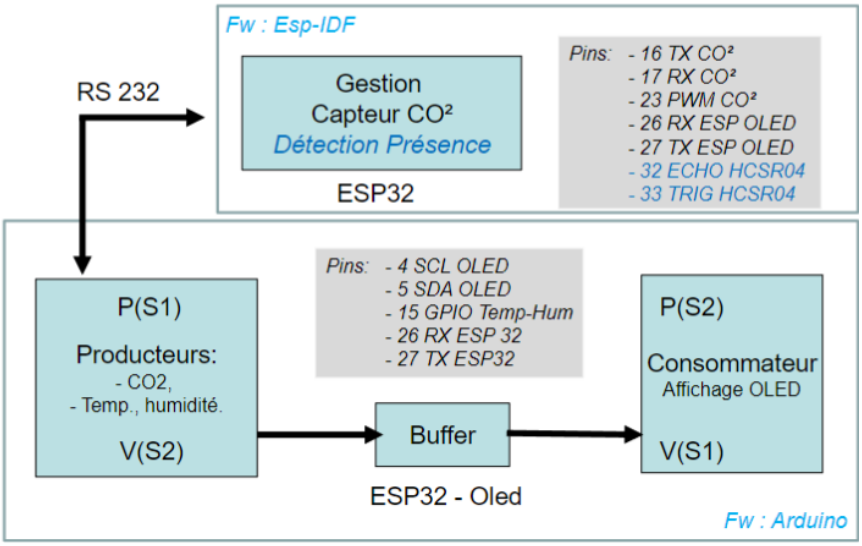
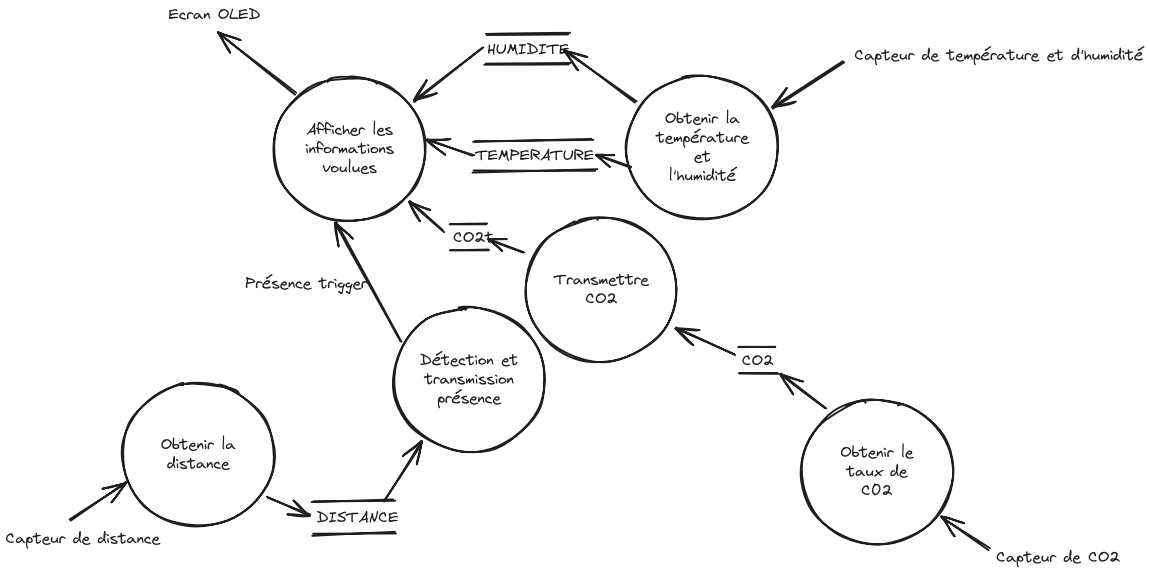


Image 1: Schéma récapitulatif du montage souhaité.png

4. Schéma SART



Téléverser le code

Pour téléverser le code, il est possible qu'il faille changer le port de communication (COM8 et COM10 par défaut) définie dans la configuration de platform IO.

Composants

> Tous les composants ont été fournis dans le cadre de notre cours de système temps réel. Une partie des documentations nous a de même été fournie par notre enseignant. Une autre partie est disponible en ligne par l'utilisation des liens fournis dans chacune des sections. Pour les autres composant, des documentations similaires sont disponibles gratuitement en ligne.

5. ESP32

Les deux ESP ont été programmé en utilisant le module platform IO de Visual Studio Code. Cette ESP a été codé en utilisant le Framwork Idf

6. ESP32 OLED

Sur cet ESP, nous avons utilisé le Framework Arduino.

7. Capteur de CO_2

Modèle du capteur **MH-Z19B**

Branché sur l'ESP32 (non OLED)

Période de la tâche dédiée : 10000ms

La communication avec le capteur de CO_2 peut se faire via l'UART ou bien le PWM. Nous avons donc développé deux codes permettant de communiquer avec le capteur. Cependant, pour la version finale du projet, nous n'avons gardé que la version avec l'UART, ce qui permet par la suite de transmettre, sans transformer l'information (sous forme de byte), les données du capteur via une deuxième connexion vers l'ESP32 OLED.

Le tâche du capteur de CO_2 à une période relativement lente car sa valeur ne change que très peu au cours de quelques secondes. Sa période est plus lente que celle de la température qui peux varier plus rapidement.

0x86- Read CO2 concentration								
Request								
Byte0	Byte1	Byte2	Byte3	Byte4	Byte5	Byte6	Byte7	Byte8
Start Byte	Sensor #	Command	-	-	-	-	-	Checksum
0xFF	0x01	0x86	0x00	0x00	0x00	0x00	0x00	0x79
Response								
Byte0	Byte1	Byte2	Byte3	Byte4	Byte5	Byte6	Byte7	Byte8
Start Byte	Sensor #	Concentration (High Byte)	Concentration (Low Byte)	-	-	-	-	Checksum
0xFF	0x86	HIGH	LOW	-	-	-	-	Checksum
CO2 concentration = HIGH * 256 + LOW								

Image 2: Extrait documentation du capteur.png

Documentation extraite du site Winsen

Paramètres

```c
#define CO2_TXD_PIN     17
#define CO2_RXD_PIN     16
#define CO2_PERIOD      10000
```

Codeblock 1 (language-c)

8. Capteur de température et d'humidité

Modèle du capteur **DHT22**

Branché sur l'ESP32 OLED

Période de la tâche dédiée : 5000ms

La communication avec le capteur de T/H était plus simple qu'avec le capteur de CO_2. Cette dernière ne requiert pas de communication particulière puisqu'une pin de data est présente sur le capteur et que de nombreuses bibliothèques existent afin de faciliter l'utilisation de ce capteur.

La bibliothèque utilisé est DHTesp, un "fork" de arduino-DHT originalement écrite par mark@paracas.nl.

Documentation sur le site de DIDEL au format PDF.

```c
const int DHT_PIN                   = 15;
const int TEMPERATURE_HUMIDITY_PERIOD = 5000;
```

Codeblock 2 (language-c)

9. Capteur de distance

Afin de simuler la captation d'une présence, nous avons utilisé un capteur ultrason.

Modèle du capteur : HC-SR04 Ultrasonic Range Finder

Branché sur l'ESP32 (non OLED)

Période de la tâche dédiée : 10ms

La période de calcul de la distance doit être rapide pour capter rapidement si un objet est à moins de 50cm du capteur (valeur modifiable dans la section dédiée du code). Si jamais une présence est captée, la tâche mets à jour la pin de sortie (défaut 25) sur high. Et inversement, lorsque la distance est supérieur à la limite pendant plus d'un certain temps (défaut 5 secondes $\pm$ 10ms), la tâche éteint la pin.

L'autre ESP gère la détection de la présence via une ISR. La détection de présence n'impacte que l'affichage et en aucun cas le modèle producteur consommateur car nous souhaitions pouvoir mettre en place un serveur web enregistrant les données même lorsque la captation n'est pas active.

Comme pour le capteur précédent, nous avons utilisé une bibliothèque afin de facilité la communication avec le capteur ultrason bien qu'il soit possible de s'en passer.

Dans notre cas, nous avons utilisé la bibliothèque ultrasonic (Copyright (c) 2016 Ruslan V. Uss unclerus@gmail.com) mais d'autres existent.

```
#define MAX_DISTANCE_CM             500  // 500 cm max
#define WAKEUP_DELAY_SENSIBILITY_MS 5000 // Temps entre la fin de la détection de présence et le signal d'extinction de la présence
#define TRIGGER_GPIO                33
#define ECHO_GPIO                   32
#define DISTANCE_PERIOD             10   // in ms, very fast to catch fast a person passing in front of the sensor
#define TRIGGER_DISTANCE            50
#define DISTANCE_PIN_OUTPUT         25   // Pin that send the data to the other ESP
```
Codeblock 3 (language-c)

10. Communication UART entre les ESP32

Connection de ESP32 -> ESP32 OLED. 2 Bytes de data sont transférés contenant les valeurs reçu du capteur de CO2.

La période de la tâche est de : 10000ms

Cette tâche à une période similaire à celle qui s'occupe du capteur CO_2 car elle n'envoie des données que lorsqu'une données est disponible du capteur. Il n'y a donc pas d'intérêt à ce qu'elle soit plus rapide.

Les données transmises par la communication UART sont les données obtenues de la tâches qui s'occupe de la mesure du CO_2 en suivant un modèle producteur consommateur.

La bibliothèque dédié aux environnements ESP32 a été utilisé. Il s'agit de driver/uart.h, SPDX-File CopyrightText: 2015-2023 Espressif Systems (Shanghai) CO LTD.

ESP32

```
#define ESP_RX_BUF_SIZE  1024
#define ESP_TXD_PIN      26
#define ESP_RXD_PIN      27
#define ESP_UART_LEN     2
#define ESP_PERIOD       10000
```
Codeblock 4 (language-c)

ESP32 OLED

```
const int UART_ESP_PERIOD = 10000;
const int RS232_RX        = 27;
const int RS232_TX        = 26;
const int RX_BUF_SIZE     = 1024;
const int LEN_RS232       = 2;
```
Codeblock 5 (language-cpp)

11. Affichage sur l'écran OLED

La période de la tâche est de 500ms.

Afin d'avoir un temps de réponse à la détection assez rapide, la période de la tâche d'affichage est de 500ms. Néanmoins, les données affichées ne sont pas rafraichies aussi rapidement et suivent un modèle producteurs multiples consommateur unique (Tâche d'affichage). Si une données n'est pas rapidement disponible, la tâche affiche la dernière valeur. Dans le cas contraire, une attente de plusieurs secondes pourrais ralentir l'affichage, la tâche étant bloquée par l'attente d'une nouvelle valeur de CO_2, d'humidité ou de température.

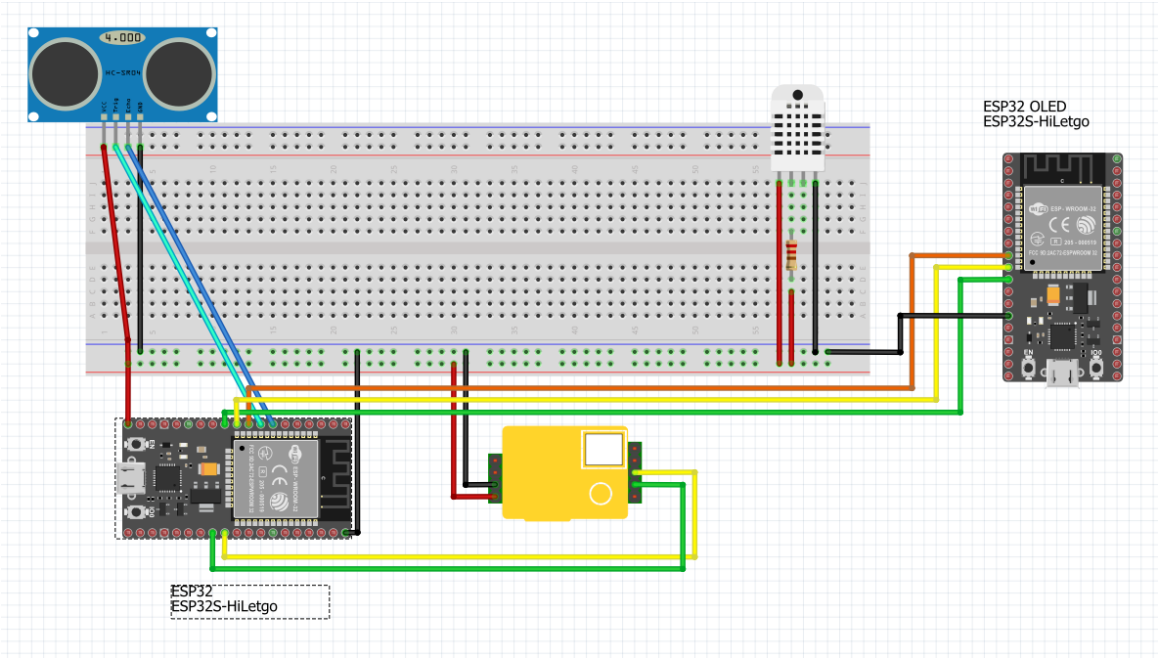
Pour l'affichage sur l'écran OLED, nous avons utilisé la bibliothèque SSD1306Wire sous licence MIT et par Copyright (c) 2018 ThingPulse, Daniel Eichhorn et Fabrice Weinberg.

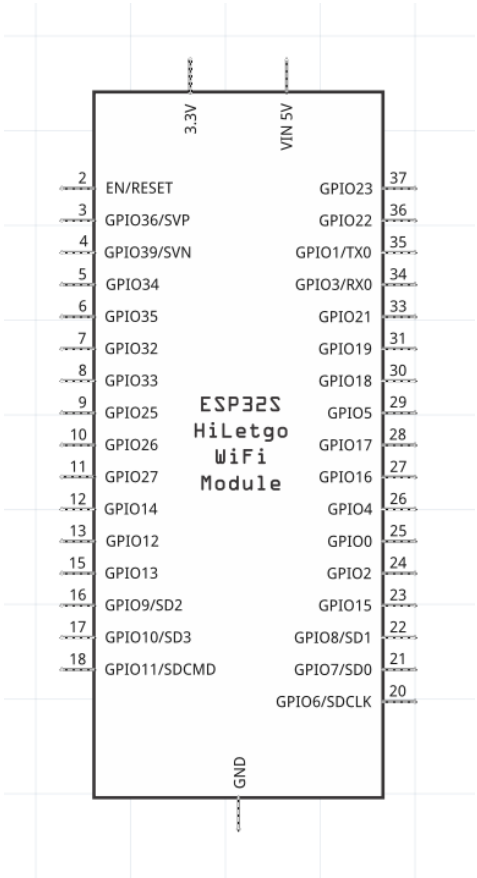
```
const u_int8_t DISTANCE_PIN_INPUT = 25;
const uint16_t DISTANCE_PERIOD    = 500;
```
Codeblock 6 (language-cpp)

Schéma de branchement

Pour le schéma de câblage nous avons utilisé le logiciel Fritzing ainsi que des pièces externes que nous avons pu trouver sur différents Github/ Forum Fritzing et qui nous ont fait gagné beaucoup de temps et rendent le schéma bien plus facile à comprendre.

Pour le capteur de MH-Z19 de CO_2 : TD-er fritzing-parts
Pour l'ESP32 : Forum Fritzing
Pour le capteur ultrason : emilio2hd arduino





Communication entre les tâches

12. Sur l'ESP32 OLED entre les producteurs et le consommateur

12.1 Producteurs

Un buffer (tableau) disposant de 50 places est disponible pour le partage avec le consommateur. Les données de ce buffer sont des DataType, une structure C possédant 2 champs, le premier "type" est une valeur d'enum qui permet de savoir quel est le producteur de la donnée. Le deuxième est value et contient la valeur de la data (par exemple la température sous forme d'entier 16 bits).

Afin de ne pas dupliquer de code à l'intérieur de chacune des tâches de production, nous avons fait une fonction qui permet d'ajouter au buffer une nouvelle valeur. Pour ce faire, nous utilisons 3 sémaphores. Une binaire représentant la disponibilité du buffer et deux autres qui représentent le nombre de places libres dans le buffer pour ajouter une production et le nombre de place contenant une valeur à lire pour le consommateur.

Afin de ne jamais perdre de données, la prise de sémaphore est bloquante sans limite de temps pour les producteur.

```c
const int bufferSize   = 50;
const int DELAY_REMOVE = 100; // After 250ms waiting a semaphore, the program abort and send error data.

enum DataTypeEnum {
  ERROR       = 0,
  Humidity    = 1,
  CO2         = 2,
  Temperature = 3
};

typedef struct {
  enum DataTypeEnum type; // Le type de production
  uint16_t value;         // Valeur de la production
} DataType;
```

Codeblock 7 (language-c)

12.2 Consommateur

Afin de faire comme pour les producteurs, nous avons réalisé une fonction permettant de consommer une valeur du buffer. Néanmoins, contrairement aux producteurs, une limite de temps a été fixée pour l'attente de nouvelle données.

En effet, si jamais la tâche était bloquée par la production, alors, lorsqu'une demande d'affichage est faire, l'affichage ne sera pas immédiat mais devra lui aussi attendre que la sémaphore se débloque. C'est pourquoi, nous avons choisi d'attendre au maximum 200 ms que les sémaphores se libèrent. Dans le cas contraire, la tâche affiche les anciennes valeurs consommées si l'affichage est actif, ou ne fait rien. Evidemment, aucune valeur du buffer n'est consommée.

```c
if (xSemaphoreTake(xBufferPCPlein, pdMS_TO_TICKS(DELAY_REMOVE)) == pdTRUE)
```

Codeblock 8 (language-c)

13. Communication entre la tâche du CO_2 et la tâche de l'UART de l'ESP32

Nous aurions pu ne faire qu'une seule tâche qui lit le CO_2 puis envoie la valeur via une connexion UART utilisé dans un seul sens seulement dans notre implémentation (ESP32 -> ESP32 OLED).

Cependant, nous trouvions ce code moins bien structuré et moins évolutif. En effet, si jamais un autre capteur venais à être ajouter, un ESP32 n'ayant que 3 connections UART, il aurait soit fallu ajouter une nouvelle connexion soit refaire le code de la fonction CO_2. Alors que dans notre cas, l'ajout d'élément par le(s) producteur(s) étant géré par une fonction dédié de même que la consommation et l'envoi, un ajout ne nécessiterait que la modification des données transférées.

De plus, en faisant des recherches sur les fonctions disponibles pour utiliser les sémaphores, les mutex etc... Nous avons découvert l'existence d'un buffer pré-fait par le module de "FreeRTOS" et permettant de largement simplifier l'implémentation d'un modèle producteur consommateur. Voulant tout de même implanté un modèle manuellement, nous avons pensé qu'il serait intéressant pédagogiquement de faire les deux.

14. Communication de la distance

La mesure de la distance est gérée par une tâche avec une très petite période afin d'augmenter la réactivité du système.
Cette tâche transmet ensuite la détection de la présence (binaire) à l'autre ESP via un branchement 15 output -> 15 input entre les ESP. Une valeur HIGH correspond à la détection et une valeur LOW à la non détection après un certain temps (défait 5 seconde).

Du côté de l'ESP32 OLED, une interruption différée (ISR) gère la modification d'une valeur consultée par la fonction d'affichage. Afin de gérer l'accès à cette variable, nous avons mis en place deux zones critiques qui semblent plus appropriées à la gestion des ISR que les sémaphores selon la documentation de FreeRTOS et les différents forums et codes que nous avons pu consulter en ligne. Encore une fois, cela nous a permis de mettre en place une autre méthode de gestion des données partagées par du code asynchrone et semble parfaitement adaptée à notre problématique.

```
void IRAM_ATTR handleInterrupt() {

  // Lire l'état actuel de la broche
  int state = digitalRead(DISTANCE_PIN_INPUT);

  portENTER_CRITICAL_ISR(&mux);
  displayEnabled = (state == HIGH);
  portEXIT_CRITICAL_ISR(&mux);

}
```

Codeblock 9 (language-c)

Copyrights

15. Libraries

A large majority of the library we used is OpenSource under MIT licence. And depend to their own licence.

16. Our code and the images