**Ecole Nationale d'Ingénieurs de Tunis**
Département Technologies de l'Information et de la Communication

# RAPPORT DE MINI PROJET RTOS

## Système d'Alerte Température Embarqué

**Plateforme : STM32F446RE (Nucleo-F446RE)**
Capteur : DHT22 (température & humidité)
RTOS : FreeRTOS (CMSIS-RTOS v2)
Interface : Dashboard React + Web Serial API + EmailJS

**Réalisé par:**
Belsem Jemai
Mohamed Ramez Mabrouk
Amira Ferchichi
Mohamed Jerbi
Yosser Jabloun
Achref Kerkeni

**Encadré par:**
Mme. Asma Ben Ahmed

Année universitaire 2025/2026

# 1. Introduction et Contexte du Projet

Dans de nombreux environnements industriels, data centers, serres agricoles ou laboratoires, la surveillance continue de la température constitue un enjeu critique. Un dépassement de seuil thermique non détecté peut entraîner des dommages matériels coûteux, des pertes de données, voire des risques pour la sécurité. Ce projet propose une solution embarquée complète répondant à cette problématique.

Le système développé repose sur un microcontrôleur STM32F446RE exploité via la carte Nucleo-F446RE de STMicroelectronics. Le firmware, structuré autour du système d'exploitation temps réel FreeRTOS, pilote un capteur DHT22 et transmet les données mesurées en JSON via UART. Une interface web moderne se comportant d'un dashboard React qui remplace avantageusement l'outil terminal PuTTY traditionnel en offrant une visualisation graphique, des statistiques et une notification automatique par email.

## 1.1 Objectifs du Projet

1. Acquérir et valider la température et l'humidité en temps réel à l'aide d'un capteur DHT22
2. Implémenter une architecture multitâche sous FreeRTOS avec synchronisation par sémaphore
3. Transmettre les données au format JSON via UART 115200 baud
4. Visualiser les données dans un dashboard web React avec historique graphique
5. Déclencher automatiquement une alarme visuelle et sonore au dépassement d'un seuil (30 °C)
6. Envoyer une notification email automatique via le service EmailJS

## 1.2 Caractéristiques Techniques Globales

| Paramètre | Valeur |
|---|---|
| Microcontrôleur | STM32F446RET6 (ARM Cortex-M4, 180 MHz) |
| Carte de développement | NUCLEO-F446RE |
| Capteur | DHT22 (AM2302) |
| Système d'exploitation | FreeRTOS via CMSIS-RTOS v2 |
| Communication | USART2 @ 115200 baud, 8N1 |
| Seuil d'alerte température | 30,0 °C (configurable) |
| Intervalle de mesure | 2 secondes (contrainte DHT22) |
| Interface utilisateur | Dashboard React (Vite + Recharts) |
| Notification email | EmailJS (service cloud gratuit) |
| Format de données | JSON structuré sur flux série |

# 2. Architecture Matérielle

## 2.1 La Carte Nucleo-F446RE

La carte NUCLEO-F446RE est une plateforme de développement économique proposée par STMicroelectronics. Elle intègre un débogueur/programmeur ST-LINK V2-1 embarqué permettant une programmation directe sans programmateur externe, ainsi qu'un port USB-UART virtuel facilitant la communication série avec un PC.

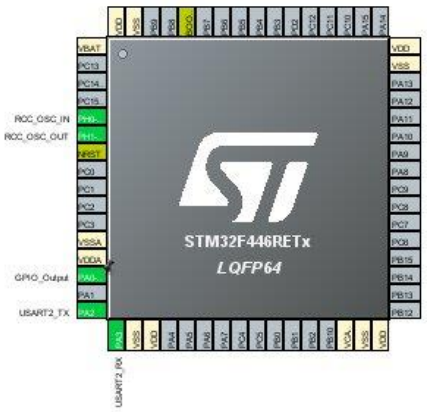



*Figure 1 : Configuration des broches du STM32F446RETx dans STM32CubeIDE*

## 2.2 Le Capteur DHT22

Le DHT22 (également dénommé AM2302) est un capteur numérique de température et d'humidité relative. Il embarque un convertisseur analogique-numérique 14 bits et communique via un protocole monofil propriétaire (single-wire). Il se distingue de son prédécesseur DHT11 par une meilleure précision et une plage de mesure étendue.

| Paramètre DHT22 | Spécification |
|---|---|
| Plage de température | -40 °C à +80 °C |
| Précision température | ± 0,5 °C |
| Plage d'humidité | 0 % à 100 % HR |
| Précision humidité | ± 2 % à 25 °C |
| Résolution | 0,1 °C / 0,1 % HR |
| Protocole | Single-wire (1-Wire propriétaire) |
| Tension d'alimentation | 3,3 V – 5,5 V |
| Courant de veille | < 60 µA |
| Intervalle minimal entre mesures | 2 secondes |
| Longueur de trame | 40 bits (16 hum. + 16 temp. + 8 checksum) |

## 2.3 Branchement et Connexions

La broche de données du DHT22 est connectée sur PA0 du STM32F446RE. Cette broche est configurée dynamiquement en sortie (push-pull) lors de l'émission du signal de start, puis en entrée (avec pull-up interne) lors de la réception des données. Cette technique de commutation dynamique de direction permet d'utiliser une seule broche pour la communication bidirectionnelle.

| Broche DHT22 | Connexion STM32 / Nucleo |
|---|---|
| VCC (pin 1) | 3.3 V (CN6-pin 4 ou CN7-pin 16) |
| DATA (pin 2) | PA0 (CN7-pin 28) + résistance pull-up 10 kΩ |
| NC (pin 3) | Non connecté |
| GND (pin 4) | GND (CN6-pin 6) |

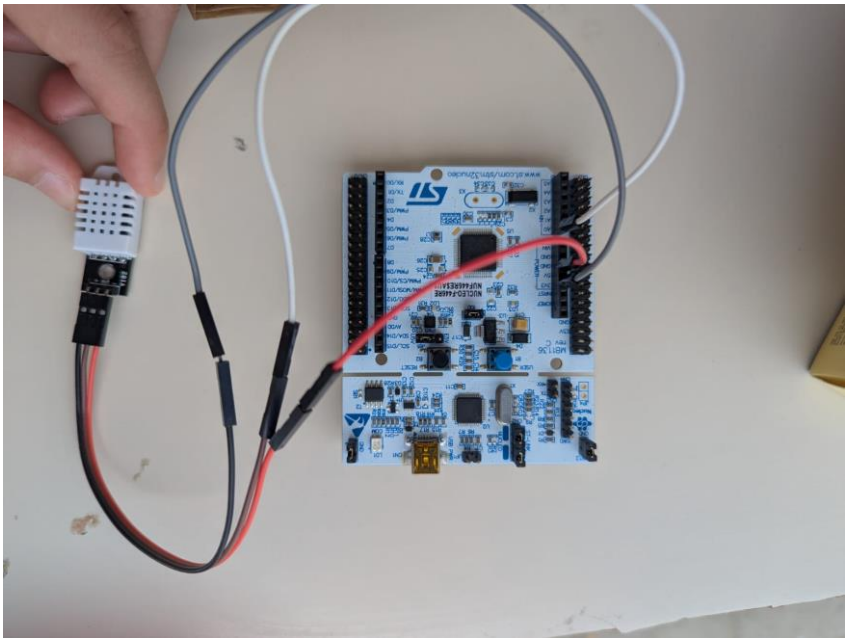


*Figure 2 : Montage physique : DHT22 connecté à la carte Nucleo-F446RE*

# 3. Firmware STM32 : Analyse Détaillée du Code

## 3.1 Configuration Système et Périphériques

### 3.1.1 Configuration de l'horloge système (SystemClock_Config)

Le microcontrôleur est configuré pour fonctionner à 50 MHz via une PLL alimentée par le HSE (oscillateur externe 8 MHz fourni par le ST-LINK). La configuration choisie est optimisée pour minimiser la consommation tout en garantissant une résolution microseconde pour le timer de délai.

```
Calcul : HSE = 8 MHz → PLL (PLLM=4, PLLN=50, PLLP=2) → SYSCLK = 50 MHz

 #define DHT22_PORT GPIOA
 #define DHT22_PIN  GPIO_PIN_0
 #define TEMP_SEUIL 30.0f  // Seuil d'alerte en degrés Celsius
```

### 3.1.2 Timer TIM6 — Générateur de délais microseconde

Le TIM6 est un timer de base 16 bits configuré avec un prescaler de 49, ce qui, pour une clock timer de 50 MHz, génère une fréquence de comptage de 1 MHz (1 tick = 1 µs). Ce timer est utilisé de façon bloquante dans le pilote DHT22 pour des délais précis inférieurs à 100 µs.

```
htim6.Init.Prescaler = 49;   // 50 MHz / (49+1) = 1 MHz → 1 tick = 1 µs
htim6.Init.Period = 0xFFFF;  // Compteur libre 16 bits
```

### 3.1.3 USART2 — Communication série

L'USART2 est configuré à 115200 baud, 8 bits de données, pas de parité, 1 bit de stop (8N1). Il est acheminé via le ST-LINK V2-1 intégré à la carte Nucleo, apparaissant comme un port COM virtuel sur le PC hôte. La fonction __io_putchar() redirige printf() vers l'UART.

## 3.2 Pilote DHT22 Bas Niveau

### 3.2.1 Protocole de communication DHT22

Le DHT22 utilise un protocole de communication monofil à 5 phases distinctes. Chaque transaction démarre par un signal de réveil envoyé par le MCU, suivi d'une réponse du capteur, puis de la transmission de 40 bits d'information.

| Phase | Description | Durée |
|---|---|---|
| 1 - Start MCU | MCU tire la ligne BAS (signal de réveil) | ≥ 1 ms (impl. : 1,2 ms) |
| 2 - MCU relâche | MCU passe en input, ligne HIGH brièvement | 30 µs |
| 3 - DHT22 BAS | DHT22 confirme réception en tirant BAS | ~80 µs |
| 4 - DHT22 HAUT | DHT22 signale prêt en relâchant (HIGH) | ~80 µs |
| 5 - 40 bits | DHT22 envoie les données | ~5 ms total |

### 3.2.2 Décodage des 40 bits

Chaque bit est décodé en mesurant la durée du niveau HAUT : si après 40 µs d'attente la ligne est toujours à l'état HAUT, le bit vaut 1, sinon il vaut 0.

```
DHT22_Delay_us(40); // Attendre 40 µs (seuil de décision)
 if (HAL_GPIO_ReadPin(PORT, PIN) == GPIO_PIN_SET)
     data[i / 8] |= 0x01; // Bit = 1 si encore HIGH après 40 µs
```

| Octets | Contenu | Interprétation |
|---|---|---|
| data[0] | Humidité — octet de poids fort | MSB humidité × 256 |
| data[1] | Humidité — octet de poids faible | LSB humidité |
| data[2] | Température — octet de poids fort (bit 7 = signe) | MSB température |
| data[3] | Température — octet de poids faible | LSB température |
| data[4] | Checksum (somme des 4 octets précédents) | Validation d'intégrité |

```
uint16_t raw_hum = ((uint16_t)data[0] << 8) | data[1];
 uint16_t raw_temp = ((uint16_t)(data[2] & 0x7F) << 8) | data[3];
 *hum = raw_hum / 10.0f;  // Ex: 556 → 55.6 %HR
```

```
*temp = raw_temp / 10.0f; // Ex: 253 → 25.3 °C
if (data[2] & 0x80) *temp = -(*temp);
```

## 3.3 Architecture FreeRTOS : Tâches et Sémaphore

Le firmware est organisé en trois tâches FreeRTOS indépendantes qui s'exécutent de manière concurrente. La synchronisation entre les tâches de mesure et d'alarme est assurée par un sémaphore binaire, évitant tout polling actif.

> *« Le sémaphore ne sert pas à rendre le programme possible. Il sert à rendre le système intelligent, efficace et temps réel. »*

| Tâche | Priorité | Stack | Rôle |
|---|---|---|---|
| DefaultTask | Normal | 128 × 4 = 512 B | Tâche idle — boucle vide (1s osDelay) |
| TemperatureTask | Normal | 256 × 4 = 1024 B | Lecture DHT22 toutes les 2s, publication JSON, déclenchement sémaphore |
| AlarmTask | Normal | 256 × 4 = 1024 B | Attend le sémaphore, émet alerte UART, anti-spam 3s |

### 3.3.1 Configuration dans STM32CubeIDE

Les tâches et le sémaphore sont configurés graphiquement dans STM32CubeIDE via l'interface CMSIS-RTOS v2. Les captures suivantes illustrent la configuration adoptée.

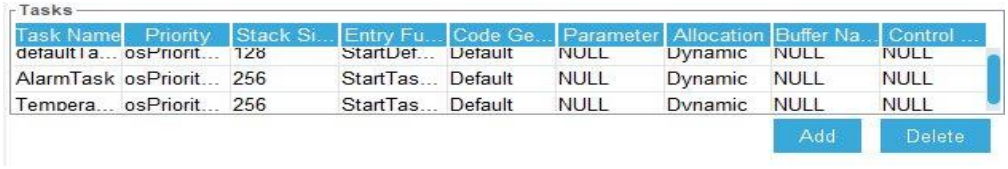



*Figure 3 : Configuration des tâches FreeRTOS dans STM32CubeIDE*

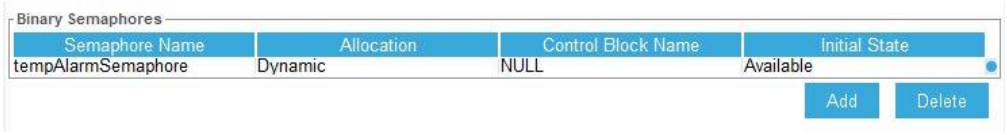



*Figure 4 : Configuration du sémaphore binaire tempAlarmSemaphore dans STM32CubeIDE*

### 3.3.2 Mécanisme de synchronisation par sémaphore

Un sémaphore binaire initialisé à 0 est utilisé pour signaler à l'AlarmTask qu'un dépassement de seuil a eu lieu. Le flag alarm_pending est critique : sans lui, chaque mesure en état d'alerte déposerait un jeton dans le sémaphore, qui s'accumulerait pendant les 3 secondes de l'anti-spam, provoquant une rafale d'alertes au réveil.

```
if (t >= TEMP_SEUIL && alarm_pending == 0) {
    alarm_pending = 1;
    osSemaphoreRelease(tempAlarmSemaphoreHandle); // Signal à AlarmTask
}
else if (t < TEMP_SEUIL) {
    alarm_pending = 0; // Réarmement pour le prochain dépassement
}
```

### 3.3.3 Analyse : Pourquoi le sémaphore ?

Il est tout à fait possible d'implémenter ce système sans aucun sémaphore. La tâche de lecture pourrait directement déclencher l'alarme dans sa propre boucle. Le code serait fonctionnel, mais cette approche présente des limitations fondamentales en termes de conception temps réel.

**Version sans sémaphore** (fonctionnelle mais non optimale) :

```
void TemperatureTask(void *pvParameters) {
    while(1) {
        float temp = DHT22_Read();
        if (temp > SEUIL_CRITIQUE) {
            UART_Send("ALERTE ! Temperature critique !\r\n");
            UART_Send("\a"); // bip sonore
        }
        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}
```

**Version avec sémaphore** (architecture découplée) :

```
// TemperatureTask : lecture + signal
void TemperatureTask(void *pvParameters) {
    while(1) {
        float temp = DHT22_Read();
        char msg[64];
        sprintf(msg, "Temperature : %.1f C\r\n", temp);
        UART_Send(msg);
        if (temp > SEUIL_CRITIQUE) {
            xSemaphoreGive(xAlarmSemaphore);
        }
        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}

// AlarmTask : bloquée en attente du signal
void AlarmTask(void *pvParameters) {
    while(1) {
        xSemaphoreTake(xAlarmSemaphore, portMAX_DELAY);
        UART_Send("[ALERTE CRITIQUE] Temperature anormale !\r\n");
        UART_Send("\a"); // BEL = bip PuTTY
    }
}
```

**Tableau comparatif :**

| Critère | Sans Sémaphore | Avec Sémaphore |
|---|---|---|
| Intelligent | Une tâche gère tout — aucune séparation des responsabilités | Chaque tâche a un rôle précis : lecture vs alarme |
| Efficace | AlarmTask tourne en boucle active (polling = CPU gaspillé) | AlarmTask bloquée → libère totalement le CPU |
| Temps réel | Réaction dépend du délai de la boucle — non déterministe | Réveil immédiat dès xSemaphoreGive() — déterministe |
| Architecture | Couplage fort entre lecture et alarme | Découplage complet — modulaire et maintenable |

### 3.3.4 Les trois qualités apportées par le sémaphore

**Intelligent — Séparation des responsabilités :** Sans sémaphore, une seule tâche remplit deux rôles différents : la surveillance et l'alarme. Avec le sémaphore, chaque tâche a une responsabilité unique et précise (TemperatureTask : surveille et signale ; AlarmTask : réagit et alerte). C'est le principe de Separation of Concerns, fondamental en génie logiciel et en systèmes embarqués industriels.

**Efficace — Pas de gaspillage CPU :** Sans sémaphore, si l'on avait une AlarmTask distincte, elle devrait vérifier en permanence un flag partagé dans une boucle active (busy-waiting ou polling). Avec xSemaphoreTake(portMAX_DELAY), AlarmTask est mise en état BLOCKED par le scheduler FreeRTOS. Elle ne consomme AUCUNE ressource processeur tant qu'aucun sémaphore n'est disponible. C'est l'approche event-driven.

**Temps réel — Réactivité déterministe :** En systèmes temps réel, la contrainte fondamentale est de garantir une réponse dans un délai maximal déterministe. Avec le sémaphore, dès que xSemaphoreGive() est appelé dans TemperatureTask, le scheduler FreeRTOS débloque immédiatement AlarmTask et la préempte pour l'exécuter. La réaction est quasi-instantanée et déterministe.

> *Principe fondamental : Un système temps réel ne doit pas seulement donner la bonne réponse — il doit la donner au bon moment. Le sémaphore garantit cette contrainte temporelle grâce au mécanisme de blocage/déblocage du scheduler FreeRTOS.*

### 3.3.5 Gestion des erreurs de lecture DHT22

La fonction DHT22_Read() retourne trois codes d'état distincts. En cas d'erreur, la TemperatureTask incrémente un compteur d'erreurs et transmet un message JSON d'erreur sans bloquer le système.

| Code retour | Valeur | Cause probable |
|---|---|---|
| DHT22_OK | 0 | Lecture réussie, données valides |
| DHT22_ERR_TIMEOUT | 1 | Capteur absent, câblage défectueux, ligne bloquée |
| DHT22_ERR_CHECKSUM | 2 | Bruit électrique, intégrité des données compromise |

## 3.4 Protocole de Communication JSON

Les données sont sérialisées en JSON et transmises via UART. Ce format textuel, lisible par un humain et parsable par JavaScript, est idéal pour l'interfaçage avec le dashboard web. Quatre types de messages sont définis :

**Message de données normales :**

```
{"temp":27.3,"hum":61.2,"seuil":30.0,"alerte":0}
```

**Message d'alerte thermique :**

```
{"temp":31.5,"hum":58.7,"seuil":30.0,"alerte":1}
```

**Message d'erreur de lecture :**

```
{"error":1,"err_count":3}
```

**Message d'alarme (AlarmTask) :**

```
{"alarm":true,"temp":31.5,"msg":"ALERTE! Temperature critique depassee!"}
```

# 4. Interface de Visualisation : Dashboard

## 4.1 Architecture et Technologies Front-End

Le dashboard est développé avec React 18 et Vite comme bundler/dev-server. Il remplace avantageusement PuTTY en proposant une interface graphique riche, interactive et en temps réel, fonctionnant directement dans le navigateur sans installation supplémentaire sur le PC hôte.

| Technologie | Version | Rôle |
|---|---|---|
| React | 18.x | Framework UI : composants réactifs, gestion d'état |
| Vite | 5.x | Bundler ultra-rapide, serveur de développement HMR |
| Recharts | 2.x | Bibliothèque de graphiques basée sur D3 pour React |
| Web Serial API | Natif (Chrome 89+) | Accès au port série USB depuis le navigateur |
| EmailJS | @emailjs/browser | Envoi d'emails via API REST depuis le front-end |
| Web Audio API | Natif | Génération de son d'alarme en temps réel |

## 4.2 Connexion Web Serial API

La Web Serial API est une API Web moderne permettant aux applications JavaScript exécutées dans le navigateur d'accéder aux ports série physiques ou USB virtuels. Elle requiert une interaction utilisateur (clic sur un bouton) pour des raisons de sécurité, et n'est disponible que sur les navigateurs basés sur Chromium (Chrome, Edge).

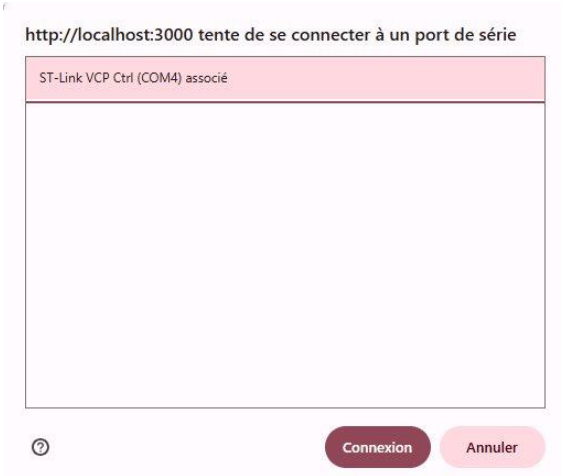



*Figure 5 : Boîte de dialogue de sélection du port série dans Chrome (Web Serial API)*

Le flux de données série est lu en continu via un reader asynchrone. Les données JSON arrivent octet par octet ; un buffer accumule les caractères jusqu'à la réception d'un saut de ligne, déclenchant alors le parsing JSON.

## 4.3 Mode Démonstration

Un mode démonstration intégré permet de tester le dashboard sans matériel physique. Il génère des données simulées toutes les 2 secondes avec une température variant de manière sinusoïdale autour de 28 °C (amplitude ±5 °C), provoquant périodiquement des dépassements du seuil de 30 °C. Ce mode est actif par défaut au démarrage.

## 4.4 Composants Visuels et Affichage Temps Réel

### 4.4.1 Jauges circulaires SVG et panneau de statistiques

Les jauges circulaires sont implémentées en SVG pur. Un cercle d'arc est dessiné avec la propriété strokeDasharray calculée dynamiquement selon le pourcentage de la valeur mesurée par rapport au maximum. Un panneau de statistiques affiche en temps réel la température maximale, minimale et moyenne de la session, ainsi que le nombre total de lectures et le statut courant.

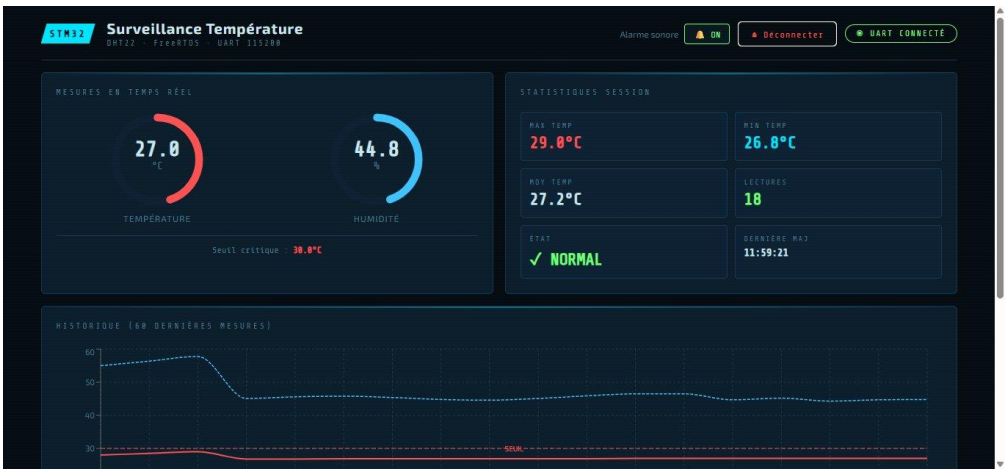



*Figure 6 : Dashboard en fonctionnement normal — jauges, statistiques et graphique temporel*

### 4.4.2 Graphique temporel et journal UART

L'historique des 60 dernières mesures est affiché sous forme de graphique linéaire avec Recharts. Une ligne de référence rouge pointillée matérialise le seuil d'alerte. Le flux JSON brut reçu depuis l'UART est affiché dans un panneau de log avec horodatage.




*Figure 7 : Graphique temporel Recharts et panneau de log UART avec flux JSON*

## 4.5 Système d'Alerte

### 4.5.1 Alerte visuelle et alarme sonore

Lorsque la température dépasse le seuil de 30 °C, une bannière d'alerte rouge clignote en haut de l'interface. Simultanément, le fond de l'application effectue un pulse rouge animé (CSS animation pulse-bg). Un signal sonore de type sirène bitonale est généré via la Web Audio API sans aucun fichier audio externe. Un oscillateur en onde carrée alterne entre 880 Hz et 660 Hz toutes les 150 ms. L'alarme peut être désactivée via le bouton bascule.

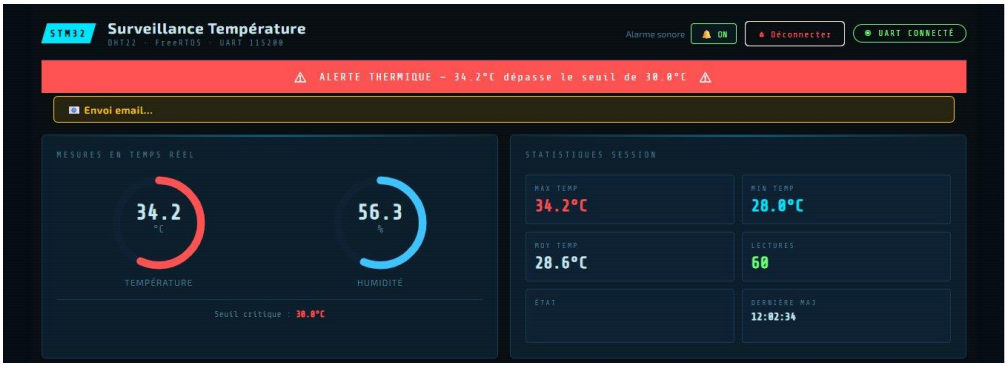



*Figure 8 : Dashboard en état d'alerte thermique (T > 30 °C) — bannière rouge et envoi email*

### 4.5.2 Anti-spam Email

L’envoi d’email utilise un mécanisme de garde via un ref React (alertEmailSentRef). Ce flag est mis à true lors du premier envoi d’email pour un dépassement donné, empêchant tout envoi redondant tant que la température reste au-dessus du seuil. Il est réinitialisé à false uniquement quand la température repasse en-dessous du seuil.

# 5. Configuration EmailJS

## 5.1 Présentation du Service

EmailJS est un service cloud qui permet d'envoyer des emails directement depuis du code JavaScript côté navigateur, sans serveur back-end. Il agit comme un intermédiaire entre l'application web et le serveur de messagerie de l'utilisateur (Gmail, Outlook, etc.). La version gratuite permet 200 emails par mois, suffisante pour un système de surveillance.

## 5.2 Procédure de Configuration

**1.** Créer un compte gratuit sur https://www.emailjs.com
**2.** Dans l'onglet « Email Services », cliquer « Add New Service » → sélectionner Gmail → Connecter le compte → Noter le Service ID
**3.** Dans l'onglet « Email Templates », cliquer « Create New Template » → Configurer le template avec les variables dynamiques
**4.** Dans « Account » → « API Keys », copier la Public Key
**5.** Reporter les 3 identifiants dans le fichier App.jsx

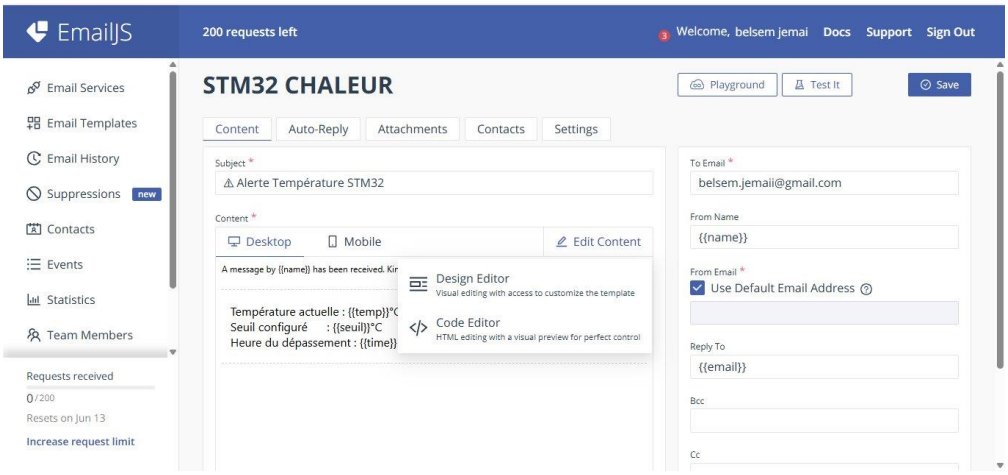



*Figure 9 : Configuration du template EmailJS avec variables dynamiques {{temp}}, {{seuil}}, {{time}}*

| Variable Template | Description | Valeur injectée |
|---|---|---|
| {{to_email}} | Adresse email destinataire | belsem.jemaii@gmail.com (configurable) |
| {{temp}} | Température mesurée au moment de l'alerte | Ex : 31.5 |
| {{seuil}} | Valeur du seuil d'alerte | Ex : 30.0 |
| {{time}} | Heure locale de déclenchement de l'alerte | Ex : 14:32:07 |

```
const EMAILJS_SERVICE_ID = 'service xxxxxxx';   // Votre Service ID
 const EMAILJS_TEMPLATE_ID = 'template xxxxxxx';  // Votre Template ID
 const EMAILJS_PUBLIC_KEY = 'xxxxxxxxxxxxxxxxxxxx'; // Votre Public Key
 const ALERT_RECIPIENT = 'votre@email.com';        // Destinataire
```

## 5.3 Exemple d'Email d'Alerte Reçu

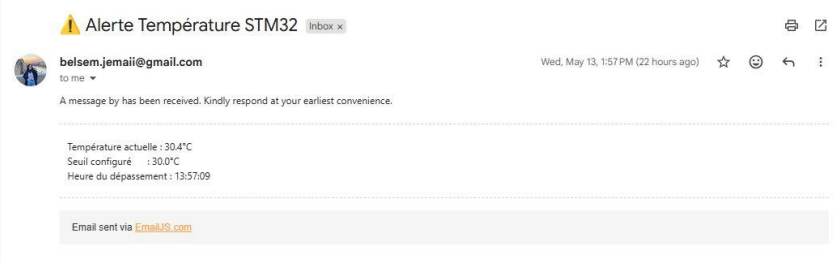



*Figure 10 : Email d'alerte reçu : notification automatique lors du dépassement de 30 °C*

# 6. Conclusion

Ce mini projet illustre parfaitement la philosophie des systèmes temps réel sous FreeRTOS. Le sémaphore n'est pas un palliatif technique pour contourner une limitation du langage — c'est un outil de conception qui transforme un programme fonctionnel en un système robuste, performant et déterministe.

Le programme existe et fonctionne sans sémaphore. Mais il devient réactif (temps réel), économe (zéro CPU gaspillé), et modulaire (tâches découplées) grâce au sémaphore — c'est précisément ce qu'on attend d'un système embarqué industriel professionnel.

L'architecture à deux tâches découplées par sémaphore est représentative des patterns utilisés dans l'industrie embarquée : supervision industrielle, domotique, IoT, systèmes d'alerte médicaux. Elle garantit la séparation des responsabilités, la réactivité aux événements critiques et l'efficience des ressources processeur.

L'ajout du dashboard React avec la Web Serial API et la notification EmailJS complète le système en offrant une interface de supervision moderne, accessible directement depuis le navigateur, avec des alertes visuelles, sonores et par email. Cette solution intégrée démontre la faisabilité d'un système de surveillance thermique complet, du capteur jusqu'à la notification utilisateur.

> *« Maîtriser le sémaphore, c'est maîtriser l'art de concevoir des systèmes embarqués temps réel de qualité industrielle. »*