

Polytech Dijon

5A Informatique et Electronique



CONCEPTION D'UN THERMOMÈTRE TEMPS RÉEL

En utilisant FreeRTOS et Harmony de la suite MPLAB X



Fait par :

**FONKOUA CHARLE LUCE SORIANE
NSENG MBAZOA OLIVIA KIMBERLY**

Superviser par :

Mr. HEYRMANN BARTHELEMY

Année 2025–2026

Table de matières

1	Introduction	4
1.1	Contexte	4
1.2	Objectif général	4
2	Environnement matériel et logiciel	6
2.1	Plateforme matérielle	6
2.1.1	Microcontrôleur PIC32MZ2048EFH100	6
2.1.2	Carte Explorer 16/32	6
2.1.3	Rôle des principaux périphériques utilisés	7
2.2	Environnement de développement	7
2.2.1	MPLAB X et MCC Harmony	7
2.2.2	Rôle du framework Harmony V3	7
2.2.3	Intégration de FreeRTOS dans le projet	8
2.3	Architecture logicielle du projet	8
2.3.1	Organisation générale de l'application	8
2.3.2	Découpage en modules logiciels	8
2.4	Gestion multitâche avec FreeRTOS	9
2.4.1	Tâche d'acquisition	9
2.4.2	Tâche de traitement	10
2.4.3	Tâche d'affichage	11
2.5	Communication inter-tâches	12
2.5.1	Files d'attente FreeRTOS (Queues)	12
2.5.2	Sémaphores binaires	13
2.5.3	Complémentarité queues / sémaphores	14
2.6	Initialisation et rôle du fichier <code>main.c</code>	14
3	RT-Therm	15
3.1	RT-Therm V1 : Implémentation séquentielle	15
3.2	RT-Therm V1.1 : Implémentation temps réel avec FreeRTOS (queues uniquement)	15
3.3	RT-Therm V2 : Implémentation temps réel avec FreeRTOS (files d'attente et sémaphores binaires)	16
3.4	Analyse et comparaison V1 / V2	17
3.4.1	Critères d'évaluation	17
3.4.2	Analyse de RT-Therm V1 : architecture séquentielle	17
3.4.3	Analyse de la version RT-Therm V1.1	18
3.4.4	Analyse de RT-Therm V2	18
3.4.5	Comparaison synthétique (niveau système)	20
3.4.6	Limites et améliorations possibles de RT-Therm V2	20
4	Conclusion	22
A	Optimisation	23
B	Organisation des fichiers	24
B.1	Final results	24

Table des Figures

A.1	Function ajouter Display temperature	23
A.2	Results	23
B.1	MPlab Harmony	24
C.1	Explorer16:32 Development Board	25
C.2	Module PIC32MZ2048EFH100	25
C.3	MPlab Harmony	25
C.4	freeRTOS	26
C.5	Fonctionnement d'une queue	26
C.6	Fonctionnement et echange de données dans les semaphore	26

Chapter 1

Introduction

1.1 Contexte

Les systèmes temps réel embarqués sont des systèmes informatiques dédiés, intégrés au sein de dispositifs matériels, dont le bon fonctionnement dépend non seulement de la validité des résultats produits, mais également du respect de contraintes temporelles strictes. Dans ce type de systèmes, une violation des délais d'exécution peut entraîner une dégradation des performances, voire une défaillance complète de l'application.

Contrairement aux systèmes informatiques généralistes, les systèmes temps réel doivent garantir un comportement déterministe, notamment en termes de latence, et de temps de réponse. Ces contraintes sont particulièrement critiques dans les applications embarquées interagissant avec un environnement physique, telles que l'acquisition de données capteurs, le contrôle de processus ou l'affichage d'informations en temps réel.

L'utilisation d'un système d'exploitation temps réel (RTOS) constitue une approche adaptée pour répondre à ces exigences. Un RTOS permet de structurer l'application sous forme de tâches concurrentes, ordonnancées selon des priorités, et synchronisées à l'aide de mécanismes spécifiques tels que les files d'attente, les sémaphores ou les mutex. Cette organisation améliore la modularité, la lisibilité et la maintenabilité du code, tout en facilitant le respect des contraintes temporelles.

Dans ce contexte, ce TP s'appuie sur un microcontrôleur de la famille PIC32MZ et le framework Harmony V3, intégrant le noyau FreeRTOS. L'application développée consiste en une version temps réel d'un thermomètre embarqué précédemment réalisé de manière séquentielle, afin de mettre en évidence les apports d'une architecture multitâche basée sur un RTOS.

1.2 Objectif général

L'objectif principal de ce travail est de concevoir et d'implémenter une application embarquée temps réel en s'appuyant sur le noyau FreeRTOS, tout en respectant une architecture logicielle multitâche clairement définie.

Plus précisément, ce TP vise à :

- mettre en œuvre un exécutif temps réel sur microcontrôleur PIC32MZ à l'aide du framework Harmony V3 ;

- décomposer l’application en plusieurs tâches indépendantes, chacune dédiée à une fonction spécifique : acquisition de la température, traitement de la donnée et affichage ;
- implémenter une communication déterministe entre les tâches à l’aide des files d’attente (*queues*) fournies par FreeRTOS ;
- analyser l’impact des choix d’ordonnancement et de gestion des délais sur le comportement global du système ;
- comparer deux stratégies d’implémentation (RT-Therm V1 et RT-Therm V2) afin d’évaluer leur cohérence, leur robustesse et leur adéquation aux contraintes temps réel.

Ce travail permet ainsi d’approfondir la compréhension des concepts fondamentaux des systèmes temps réel embarqués, notamment la concurrence, la synchronisation inter-tâches et la gestion temporelle, dans un contexte proche des applications industrielles.

Chapter 2

Environnement matériel et logiciel

2.1 Plateforme matérielle

2.1.1 Microcontrôleur PIC32MZ2048EFH100

Le microcontrôleur utilisé dans ce projet est le **PIC32MZ2048EFH100**, Figure C.2 appartenant à la famille PIC32MZ de Microchip. Il s'agit d'un microcontrôleur 32 bits basé sur une architecture *MIPS32 M-class*, cadencé jusqu'à plusieurs centaines de MHz, et disposant de ressources matérielles adaptées aux applications embarquées complexes et temps réel.

Ce microcontrôleur intègre notamment : Un cœur 32 bits haute performance avec unité de gestion des interruptions avancée, une mémoire Flash interne de grande capacité permettant l'exécution de systèmes logiciels complexes, une mémoire RAM suffisante pour supporter un RTOS et plusieurs tâches concurrentes.

Ces caractéristiques rendent le PIC32MZ particulièrement adapté à l'exécution d'un système d'exploitation temps réel tel que FreeRTOS, tout en garantissant des performances suffisantes pour le traitement et l'affichage des données en temps réel.

2.1.2 Carte Explorer 16/32

La carte **Explorer 16/32**, Figure C.1 constitue la plateforme matérielle de développement associée au microcontrôleur PIC32MZ. Elle fournit un environnement complet permettant l'exploitation rapide des périphériques internes du microcontrôleur ainsi que l'interfaçage avec des composants externes.

Cette carte intègre notamment : les connexions nécessaires à l'alimentation et à la programmation du microcontrôleur, de nombreux périphériques intégrés (ADC, timers, LCD, capteurs de température, des boutons, des leds, des interfaces parallèles et séries et plusieurs autres périphéries pouvant être utiliser pour du prototypage rapide et aux tests fonctionnels.

La carte Explorer 16/32 permet ainsi de valider le comportement temps réel de l'application dans un environnement matériel représentatif d'un système embarqué réel.

2.1.3 Rôle des principaux périphériques utilisés

Plusieurs périphériques matériels du microcontrôleur sont exploités dans le cadre de ce projet :

- **ADC (Analog-to-Digital Converter)** : utilisé pour l'acquisition de la valeur analogique issue du capteur de température. L'ADC assure la conversion de la grandeur analogique en donnée numérique exploitable par le logiciel.
- **PMP (Parallel Master Port)** : utilisé pour la communication parallèle avec l'afficheur. Ce périphérique permet des transferts rapides de données vers l'écran, limitant la charge CPU et améliorant la réactivité de l'affichage.
- **Timers matériels** : utilisés indirectement par le RTOS pour la gestion du *tick* système et des délais associés aux tâches.
- **Afficheur** : permet la restitution visuelle de la température convertie sous une forme lisible par l'utilisateur.
- **Les boutons poussoirs et les Led** : permettent de créer une IHM pour une utilisation facile avec divers utilisateurs

L'utilisation conjointe de ces périphériques permet de mettre en œuvre une chaîne fonctionnelle complète allant de l'acquisition physique jusqu'à l'affichage, sous contrôle d'un exécutif temps réel.

2.2 Environnement de développement

2.2.1 MPLAB X et MCC Harmony

Le développement logiciel est réalisé à l'aide de l'environnement **MPLAB X**, qui constitue l'IDE officiel de Microchip pour la programmation des microcontrôleurs PIC. MPLAB X permet la configuration du matériel, l'édition du code source, la compilation, le débogage et la programmation de la cible.

L'outil **MCC Harmony** (Microchip Code Configurator), Figure ?? est utilisé pour générer automatiquement l'architecture logicielle de base du projet. Il permet :

- la sélection et la configuration facile et rapide des périphériques ;
- la génération des pilotes matériels ;
- l'intégration des services système, dont le RTOS.

Cette approche réduit les erreurs de configuration bas niveau et permet de se concentrer sur la logique applicative et temps réel.

2.2.2 Rôle du framework Harmony V3

Le framework **Harmony V3** fournit une couche d'abstraction logicielle entre le matériel et l'application. Il repose sur une architecture modulaire composée de : pilotes matériels(*drivers*), services système, couche applicative.

Harmony V3 facilite l'intégration de FreeRTOS et assure la portabilité du code, tout en garantissant une gestion cohérente des périphériques et des interruptions. Il permet également une séparation claire entre le code généré automatiquement et le code applicatif spécifique au projet.

2.2.3 Intégration de FreeRTOS dans le projet

FreeRTOS, Figure C.4 est intégré au projet via les services fournis par Harmony V3. Le noyau RTOS est utilisé pour gérer l'exécution concurrente des différentes tâches de l'application, selon un ordonnancement préemptif basé sur les priorités.

L'intégration de FreeRTOS permet :

- la création de tâches indépendantes ;
- la gestion des délais et de la temporisation ;
- la communication inter-tâches à l'aide de files d'attente ;
- une meilleure maîtrise du comportement temporel du système.

Cette architecture logicielle constitue la base des implémentations RT-Therm V1 et RT-Therm V2, analysées dans la suite du rapport.

2.3 Architecture logicielle du projet

2.3.1 Organisation générale de l'application

L'application est structurée selon une architecture logicielle multitâche reposant sur le noyau FreeRTOS. Le système est décomposé en plusieurs modules fonctionnels distincts, chacun correspondant à une tâche temps réel dédiée. Cette séparation fonctionnelle permet de respecter les principes de modularité, de lisibilité et de maintenabilité du code, tout en facilitant l'analyse du comportement temporel du système.

L'architecture globale repose sur trois fonctions principales :

- l'acquisition de la température à partir du capteur analogique ;
- le traitement et la conversion de la donnée brute acquise ;
- l'affichage de la température sous une forme lisible par l'utilisateur.

Chaque fonction est implémentée sous forme de tâche FreeRTOS indépendante, exécutée de manière concurrente et synchronisée à l'aide de mécanismes de communication inter-tâches.

2.3.2 Découpage en modules logiciels

L'application est organisée en plusieurs fichiers sources distincts, correspondant chacun à une fonction précise du système :

- `acquisition.c / acquisition.h` : gestion de l'acquisition de la température via l'ADC ;

- `analysis.c` / `analysis.h` : traitement et conversion de la donnée brute ;
- `display.c` / `display.h` : affichage de la température sur l'écran ;
- `main.c` : point d'entrée de l'application et initialisation du système.

Ce découpage permet une séparation claire entre les différentes responsabilités fonctionnelles, tout en facilitant l'intégration dans l'architecture Harmony et FreeRTOS.

2.4 Gestion multitâche avec FreeRTOS

Chaque module fonctionnel est associé à une tâche FreeRTOS dédiée. Les tâches sont créées lors de l'initialisation du système et sont gérées par l'ordonnanceur du RTOS selon une politique préemptive basée sur les priorités.

2.4.1 Tâche d'acquisition

La tâche d'acquisition est responsable de la production périodique des mesures de température à partir du convertisseur analogique-numérique (ADC) intégré au microcontrôleur. Elle constitue le point d'entrée de la chaîne de traitement et joue un rôle central dans la maîtrise temporelle du système.

Le fonctionnement de cette tâche repose sur une séquence déterministe d'opérations, exécutées à chaque période d'échantillonnage :

- **Déclenchement de la conversion ADC** : Le démarrage de la conversion est effectué par l'appel à la fonction `ADCHS_GlobalEdgeConversionStart()`. Cette primitive, fournie par la couche PLIB du framework Harmony, ordonne au module ADCHS de lancer une conversion globale selon le mode *edge-triggered* configuré lors de l'initialisation du périphérique. Le module ADCHS ayant été préalablement initialisé via `ADCHS_Initialize()` lors de la phase d'initialisation du système, cet appel garantit que les paramètres matériels (canal, horloge, résolution, mode de déclenchement) sont correctement appliqués avant chaque acquisition.
- **Vérification de la disponibilité du résultat et lecture de la donnée brute** : Une fois la conversion déclenchée, la tâche teste la disponibilité du résultat sur le canal ADC dédié à la mesure de température à l'aide de :

`ADCHS_ChannelResultIsReady(ADCHS_CH4)`. Cette étape permet de s'assurer que la conversion est terminée avant toute lecture. Lorsque le résultat est disponible, la valeur analogique brute est récupérée via `ADCHS_ChannelResultGet(ADCHS_CH4)` et stockée dans une variable de type `uint16_t`. À ce stade, la donnée n'est pas interprétée ni convertie ; elle représente uniquement l'échantillon numérique issu de l'ADC.

- **Transmission de la donnée vers la chaîne de traitement** : La valeur brute acquise est transmise à la tâche de traitement à l'aide d'une file d'attente FreeRTOS, via l'appel `xQueueSend()`. Ce mécanisme assure une communication thread-safe et permet un découplage temporel entre l'acquisition et le traitement. En cas de succès de l'envoi, un sémaphore binaire est libéré afin de signaler la disponibilité d'une nouvelle mesure, et un indicateur visuel (LED) est activé à des fins de validation expérimentale.

La tâche d'acquisition est la **seule tâche strictement périodique du système**. Sa temporisation définit la fréquence d'échantillonnage globale et sert de référence temporelle pour l'ensemble de l'application. Les tâches de traitement et d'affichage sont déclenchées de manière événementielle à partir des données produites par cette tâche, ce qui permet de limiter la charge processeur et de garantir un comportement déterministe.

Le flot fonctionnel de cette tâche, depuis le déclenchement de la conversion ADC jusqu'à l'envoi de la donnée brute dans la file d'attente, est illustré par le schéma présenté en annexe (voir Fig. ??).

2.4.2 Tâche de traitement

La tâche de traitement constitue l'étape intermédiaire entre l'acquisition brute de la température et son affichage. Elle est responsable de l'interprétation de la donnée issue de l'ADC, de sa conversion en une grandeur physique exploitable et de sa transmission vers la tâche d'affichage. Cette tâche fonctionne exclusivement selon un modèle événementiel, ce qui permet d'optimiser l'utilisation du processeur.

Le fonctionnement de la tâche de traitement repose sur la séquence suivante :

- **Synchronisation avec la tâche d'acquisition :** La tâche de traitement est réveillée par la libération d'un sémaphore binaire par la tâche d'acquisition. L'appel à `xSemaphoreTake(mySemaphore, 0)` permet de tester la disponibilité d'une nouvelle mesure sans blocage prolongé. Ce mécanisme garantit que la tâche ne s'exécute que lorsqu'une donnée valide est effectivement disponible, évitant ainsi tout traitement inutile.

Ceci est aussi garantie par une longueur de queue de profondeur 10 qui permet d'avoir des valeurs à disposition dès la fin de traitement de la valeur précédemment acquise. Cela est fait en utilisant (`myQueue = xQueueCreate(10, sizeof(uint16_t))`.)

- **Réception de la donnée brute via la file d'attente :** Une fois le sémaphore acquis, la valeur brute issue de l'ADC est récupérée depuis la file d'attente partagée à l'aide de `xQueueReceive(myQueue, &temp_val_A, 0)`. La donnée est stockée dans une variable de type `uint16_t`, correspondant directement à la résolution matérielle de l'ADC.
- **Conversion en grandeur physique :** La valeur brute ADC est convertie en température réelle selon une relation linéaire, implémentée dans le code par :

$$T = \left(\frac{V_{\text{ADC}} - 0.5}{0.01} \right)$$

avec

$$V_{\text{ADC}} = \frac{\text{Code}_{\text{ADC}} \times 3.3}{4095}$$

Cette formule correspond au modèle d'un capteur de type LM35/TMP36, où la tension est proportionnelle à la température. Le calcul est réalisé en virgule flottante (`double`) afin de préserver la précision numérique lors de la conversion.

- **Transmission de la donnée traitée vers la tâche d'affichage :** La température calculée est envoyée à la tâche d'affichage via une seconde file d'attente FreeRTOS à l'aide de `xQueueSend(myQueue1, &Temp_real_value, 10)`. Ce mécanisme assure

un découplage temporel entre le traitement et l'affichage et garantit une communication thread-safe. En cas de succès, un second sémaphore est libéré afin de signaler la disponibilité d'une nouvelle donnée formatée, et un indicateur visuel (LED) est activé à des fins de diagnostic et de validation expérimentale.

La tâche de traitement ne possède aucune temporisation propre. Elle reste bloquée tant qu'aucune donnée n'est disponible, ce qui permet de réduire la charge processeur et d'assurer un comportement déterministe du système. La séparation stricte entre acquisition, traitement et affichage contribue également à une meilleure maintenabilité du code et à une validation plus aisée de chaque étape fonctionnelle.

Le flot de fonctionnement de la tâche de traitement, depuis la synchronisation par sémaphore jusqu'à l'envoi de la température convertie vers la tâche d'affichage, est illustré dans le schéma présenté en annexe (voir Fig. ??).

2.4.3 Tâche d'affichage

La tâche d'affichage est responsable de la restitution visuelle de la température calculée par la tâche de traitement. Elle constitue la dernière étape de la chaîne fonctionnelle et assure l'interface directe entre le système embarqué et l'utilisateur. Cette tâche fonctionne selon un modèle événementiel et ne s'exécute que lorsqu'une nouvelle donnée est disponible.

Le fonctionnement de la tâche d'affichage repose sur les étapes suivantes :

- **Initialisation de l'afficheur et des ressources associées** : Lors de son passage dans l'état DISPLAY_STATE_INIT, la tâche initialise les ressources nécessaires à l'affichage. Le démarrage du timer TMR2_Start() permet de fournir une base temporelle utilisée par les fonctions de temporisation logicielle du pilote LCD. L'appel à initializeLCD() configure ensuite l'afficheur (mode de communication, effacement, configuration des lignes), garantissant que le périphérique est prêt à recevoir des données avant l'entrée dans l'état de service. Il est important de noter que la disposition hiérarchique de l'appel de l'initialisation du timer et du initializedLCD est importante car le système ne fonctionnera pas si cela est fait en inverse.
- **Synchronisation événementielle avec la tâche de traitement** : La tâche d'affichage est synchronisée avec la tâche de traitement par l'intermédiaire d'un sémaphore binaire. L'appel à xSemaphoreTake(mySemaphore1, 0) permet de tester la disponibilité d'une nouvelle valeur de température sans bloquer inutilement la tâche. Ce mécanisme garantit que l'affichage est strictement déclenché par l'arrivée d'une nouvelle donnée traitée.
- **Réception de la température traitée** : Lorsque le sémaphore est acquis, la valeur de température est récupérée depuis la file d'attente associée via :
`xQueueReceive(myQueue1, &Temp, 0)`. La donnée est stockée dans une variable de type double, correspondant à la température physique calculée par la tâche de traitement.
- **Mise à jour de l'affichage** : La fonction displayTemperature(Temp) est alors appelée afin de convertir la valeur numérique en une représentation textuelle et de la transmettre à l'afficheur LCD. Cette fonction encapsule les opérations de formatage et d'écriture sur le périphérique d'affichage, permettant de maintenir une séparation claire entre logique applicative et gestion matérielle.

Cette fonction a été créée et rajoutée dans le fichier print.c qui a été fourni par l'enseignant pour

Une temporisation courte (`Delay_ms(50)`) est insérée afin de garantir la stabilité de l'affichage et de respecter les contraintes temporelles du contrôleur LCD. Un indicateur visuel (LED) est activé à des fins de validation et de diagnostic.

À l'instar de la tâche de traitement, la tâche d'affichage ne possède aucune temporisation périodique propre. Elle reste bloquée tant qu'aucune nouvelle donnée n'est disponible, ce qui permet de limiter la charge processeur et d'éviter des mises à jour inutiles de l'affichage. Cette approche garantit également une cohérence parfaite entre la valeur affichée et la dernière mesure effectivement traitée.

Le flot fonctionnel de la tâche d'affichage, depuis la réception de la température traitée jusqu'à sa restitution sur l'écran LCD, est illustré dans le schéma présenté en annexe (voir Fig. ??).

2.5 Communication inter-tâches

Dans l'architecture RT-Therm V2, la communication et la synchronisation entre les tâches sont assurées par les mécanismes natifs de FreeRTOS, à savoir les **files d'attente** (*queues*) et les **sémaphores binaires**. Ces primitives constituent des éléments fondamentaux des systèmes d'exploitation temps réel et permettent de structurer les échanges de données de manière sûre, déterministe et modulaire.

2.5.1 Files d'attente FreeRTOS (Queues)

Les files d'attente FreeRTOS sont utilisées comme principal mécanisme de transport de données entre les tâches. Deux files distinctes sont mises en œuvre :

- une file reliant la tâche d'acquisition à la tâche de traitement, transportant les valeurs brutes issues de l'ADC ;
- une file reliant la tâche de traitement à la tâche d'affichage, transportant les valeurs de température converties en grandeur physique.

Chaque file d'attente est créée dynamiquement lors de l'initialisation du système à l'aide de l'API `xQueueCreate()`, en spécifiant explicitement :

- la profondeur de la file (nombre maximal d'éléments stockables) ;
- la taille de chaque élément, correspondant au type de donnée échangée (`uint16_t` pour la donnée brute, `double` pour la température réelle).

Fonctionnement de base. Une file d'attente FreeRTOS fonctionne selon un principe FIFO (*First In, First Out*). Lorsqu'une tâche productrice appelle `xQueueSend()`, la donnée est copiée dans le buffer interne de la file. Une tâche consommatrice peut ensuite récupérer cette donnée via `xQueueReceive()`, soit de manière bloquante, soit avec un temps d'attente nul ou limité.

Ce mécanisme assure une **communication thread-safe** : aucune protection supplémentaire (section critique, mutex) n'est nécessaire pour les données transitant par la file.

Intégration dans le système. Dans ce projet, les files d'attente sont utilisées pour matérialiser la chaîne fonctionnelle :

Acquisition → Traitement → Affichage

Chaque tâche ne communique qu'avec sa tâche voisine, ce qui limite le couplage et améliore la lisibilité de l'architecture. Les tâches consommatrices restent bloquées tant qu'aucune donnée n'est disponible, ce qui évite tout polling inutile.

Apport des files d'attente. L'utilisation des files d'attente apporte plusieurs avantages majeurs :

- découplage temporel entre producteur et consommateur ;
- élimination des accès concurrents aux variables globales ;
- robustesse accrue face aux variations de charge ;
- simplification de l'ordonnancement global du système.

2.5.2 Sémaphores binaires

En complément des files d'attente, des sémaphores binaires sont utilisés pour assurer une synchronisation événementielle fine entre les tâches. Contrairement aux files d'attente, qui transportent des données, les sémaphores servent uniquement à signaler l'occurrence d'un événement.

Principe de fonctionnement. Un sémaphore binaire peut être vu comme un drapeau logique possédant deux états : pris ou libre. Une tâche peut libérer le sémaphore via `xSemaphoreGive()`, et une autre tâche peut l'acquérir via `xSemaphoreTake()`.

Dans ce projet, la libération d'un sémaphore indique explicitement qu'une nouvelle donnée a été produite et déposée dans la file d'attente correspondante.

Intégration dans l'architecture. Deux sémaphores binaires sont utilisés :

- un sémaphore associé à la file reliant l'acquisition au traitement ;
- un sémaphore associé à la file reliant le traitement à l'affichage.

Cette organisation permet de dissocier clairement :

- le **transport de la donnée** (assuré par la file d'attente) ;
- la **synchronisation événementielle** (assurée par le sémaphore).

Intérêt des sémaphores dans ce projet. L'utilisation des sémaphores présente plusieurs avantages :

- déclenchement immédiat des tâches consommatrices dès qu'une nouvelle donnée est disponible ;
- réduction du temps passé en attente active ;
- amélioration de la réactivité globale du système.

2.5.3 Complémentarité queues / sémaphores

L'association des files d'attente et des sémaphores permet de construire une architecture robuste et extensible. Les files assurent un transport sûr et ordonné des données, tandis que les sémaphores permettent un déclenchement événementiel précis des tâches.

Ce découplage fonctionnel est particulièrement adapté aux systèmes temps réel embarqués, car il permet :

- une meilleure maîtrise du déterminisme temporel ;
- une réduction de la charge processeur ;
- une séparation claire des responsabilités entre les tâches.

Cette approche constitue une base solide pour des évolutions futures, telles que l'ajout de nouvelles tâches (communication, journalisation) ou l'optimisation des mécanismes de synchronisation à l'aide de notifications de tâches.

2.6 Initialisation et rôle du fichier `main.c`

Le fichier `main.c` constitue le point d'entrée de l'application. Il est responsable :

- de l'initialisation du matériel via Harmony ;
- de la création des files d'attente FreeRTOS ;
- de la création des tâches applicatives ;
- du lancement de l'ordonnanceur FreeRTOS.

Une fois l'ordonnanceur démarré, le contrôle de l'exécution est entièrement pris en charge par FreeRTOS, et le comportement du système dépend exclusivement de l'ordonnancement des tâches et des événements inter-tâches.

Chapter 3

3.1 RT-Therm V1 : Implémentation séquentielle

La version RT-Therm V1 correspond à l'implémentation initiale du thermomètre développée dans le cadre du TD/TP Projet 2024. Elle repose sur une architecture séquentielle classique, sans recours à un système d'exploitation temps réel. L'ensemble de l'application est exécuté dans une boucle principale unique, située dans la fonction `main()`.

Dans cette approche, les différentes fonctions de l'application: acquisition de la température, traitement de la donnée et affichage sont exécutées successivement selon un flot de contrôle linéaire et non préemptif. La gestion temporelle du système repose uniquement sur l'ordre d'exécution des fonctions et sur l'utilisation de temporisations bloquantes, sans mécanisme de planification ou de priorisation explicite.

L'organisation logicielle est simple et s'articule principalement autour de :

- `main.c`, assurant le pilotage global de l'application ;
- `lcd.c / lcd.h`, regroupant les fonctions d'accès bas niveau à l'afficheur et les routines d'affichage.

L'acquisition de la température et son traitement sont réalisés dans le même contexte d'exécution, ce qui induit un couplage fort entre les différentes étapes fonctionnelles et empêche toute forme de parallélisme logique. De même, l'affichage est effectué de manière synchrone : chaque mise à jour de l'écran bloque l'exécution du programme jusqu'à sa complétion.

Bien que cette architecture permette une mise en œuvre rapide et fonctionnelle, elle présente des limitations importantes en termes de maîtrise temporelle, d'évolutivité et de robustesse. Ces contraintes motivent la transition vers une architecture temps réel multitâche, présentée dans la section suivante.

3.2 RT-Therm V1.1 : Implémentation temps réel avec FreeRTOS (queues uniquement)

La version RT-Therm V1.1 constitue une première transition entre l'architecture séquentielle de la version V1 et l'architecture temps réel optimisée de la version V2. Elle répond strictement aux consignes du sujet, en proposant une implémentation multitâche reposant sur FreeRTOS, tout en limitant volontairement les mécanismes de synchronisation aux seules files d'attente (*queues*).

L'application est structurée autour de trois tâches distinctes, dédiées respectivement à l'acquisition de la température, au traitement de la donnée brute et à l'affichage. Ces tâches communiquent exclusivement à l'aide de files d'attente FreeRTOS, utilisées à la fois comme mécanisme de transport de données et de synchronisation implicite.

Dans cette version, chaque tâche consommatrice reste bloquée sur une opération de réception (`xQueueReceive`) tant qu'aucune donnée n'est disponible. La réception d'un nouvel élément dans la file entraîne automatiquement le réveil de la tâche, assurant ainsi un fonctionnement événementiel sans recours à des sémaphores explicites.

La gestion temporelle reste distribuée : chaque tâche peut posséder sa propre temporisation ou dépendre du rythme de production des données. Cette approche permet déjà d'améliorer significativement la lisibilité de l'architecture, la cohérence des échanges de données et la robustesse du système par rapport à la version séquentielle V1.

Cependant, l'utilisation exclusive des files d'attente impose un couplage plus fort entre transport de données et synchronisation, et limite les possibilités d'optimisation fine du comportement temporel. Ces limitations motivent l'introduction d'une version V2 enrichie par l'utilisation de sémaphores.

Contrairement à la version RT-Therm V1.1, où la synchronisation repose uniquement sur les files d'attente, la version RT-Therm V2 introduit l'utilisation de sémaphores binaires en complément des queues. Cette évolution permet de dissocier explicitement le transport de la donnée de la synchronisation événementielle entre les tâches.

Les files d'attente sont alors utilisées principalement pour le transfert structuré des données, tandis que les sémaphores assurent un déclenchement précis et immédiat des tâches consommatrices. Cette séparation améliore la réactivité du système, réduit les ambiguïtés liées à l'état des files et facilite l'analyse du comportement temps réel.

3.3 RT-Therm V2 : Implémentation temps réel avec FreeRTOS (files d'attente et sémaphores binaires)

La version RT-Therm V2 constitue une évolution directe de la version RT-Therm V1.1, qui introduisait déjà une architecture multitâche basée sur FreeRTOS et l'utilisation exclusive des files d'attente pour la communication inter-tâches. Cette nouvelle version conserve le découpage fonctionnel en trois tâches (acquisition, traitement et affichage), tout en enrichissant les mécanismes de synchronisation afin d'améliorer la réactivité et la maîtrise du comportement temporel.

Comme dans la version V1.1, les tâches communiquent par l'intermédiaire de files d'attente, utilisées pour assurer le transport structuré des données entre les différentes étapes de la chaîne fonctionnelle. Toutefois, la version V2 introduit l'utilisation de sémaphores binaires en complément des queues, permettant de dissocier explicitement la synchronisation événementielle du transfert des données.

La gestion temporelle du système reste centralisée autour de la tâche d'acquisition, seule tâche strictement périodique, qui définit la fréquence d'échantillonnage globale. Les tâches de traitement et d'affichage fonctionnent selon un modèle événementiel, mais leur déclenchement est désormais assuré par des sémaphores, ce qui permet un réveil plus précis et immédiat des tâches consommatrices, indépendamment de l'état des files d'attente.

L'ajout des sémaphores dans RT-Therm V2 apporte une meilleure séparation des rôles entre les mécanismes RTOS, améliore la lisibilité de l'architecture et facilite l'analyse du comportement temps réel. Cette évolution permet ainsi d'optimiser la réactivité du système et justifie une comparaison approfondie entre les versions RT-Therm V1, RT-Therm V1.1 et RT-Therm V2, présentée dans la section suivante.

3.4 Analyse et comparaison V1 / V2

3.4.1 Critères d'évaluation

Afin de comparer de manière objective et structurée les différentes versions du thermomètre (RT-Therm V1, RT-Therm V1.1 et RT-Therm V2), l'analyse s'appuie sur des critères classiquement utilisés en ingénierie des systèmes embarqués temps réel :

- **déterminisme temporel** : maîtrise de la période d'acquisition, latences et gigue ;
- **gestion de la charge CPU** : efficacité d'exécution et utilisation des ressources processeur ;
- **modèle d'exécution** : séquentiel, multitâche coopératif ou événementiel ;
- **gestion de la concurrence et cohérence des données** : mécanismes de synchronisation et sûreté des échanges ;
- **maintenabilité et évolutivité logicielle** : facilité d'ajout de nouvelles fonctionnalités ;
- **couplage matériel/logiciel** : accès aux périphériques et isolation des responsabilités.

Ces critères permettent de mettre en évidence l'impact architectural du passage progressif d'une approche séquentielle à une architecture temps réel multitâche plus élaborée.

3.4.2 Analyse de RT-Therm V1 : architecture séquentielle

Déterminisme et latences

Dans RT-Therm V1, l'application est structurée autour d'une boucle principale (*super-loop*) exécutant successivement acquisition, traitement et affichage. Le temps de réponse du système est donc déterminé par la durée cumulée du chemin d'exécution complet. Toute variation du temps d'exécution d'un bloc (par exemple l'affichage LCD) se répercute directement sur la latence globale.

Cette architecture entraîne :

- une **latence d'affichage variable**, dépendante des accès au périphérique LCD ;
- une **gigue d'échantillonnage** (jitter), car la période d'acquisition dépend du temps consommé par les autres traitements ;
- l'absence de **garanties temporelles** formelles : aucune fonction ne peut être priorisée en cas de surcharge.

En particulier, si l'affichage est bloquant (écriture synchrones via le pilote LCD), une augmentation du temps d'écriture provoque mécaniquement un ralentissement de l'ensemble de la chaîne fonctionnelle.

Charge CPU et temporisations bloquantes

Dans une architecture séquentielle, les temporisations sont souvent réalisées par des délais bloquants (*busy-wait* ou boucles d'attente). Dans ce cas, le processeur reste actif sans exécuter de travail utile, ce qui :

- dégrade l'efficacité énergétique ;

- empêche toute exécution concurrente d'une autre fonction ;
- limite fortement l'évolutivité de l'application.

Même si des interruptions matérielles existent, l'absence de structuration RTOS implique que la logique applicative reste monolithique et fortement dépendante de la boucle principale.

Couplage fonctionnel et maintenabilité

Dans RT-Therm V1, acquisition, traitement et affichage sont couplés temporellement et logiquement : l'affichage ne peut se produire qu'après exécution complète des étapes précédentes. Ce couplage impose une forte dépendance entre modules et rend l'intégration de nouvelles fonctionnalités (journalisation, communication série, filtrage numérique) plus complexe, car tout ajout augmente le temps du cycle principal.

En résumé, RT-Therm V1 est simple à mettre en œuvre mais **peu scalable** et **faiblement robuste** vis-à-vis des contraintes temps réel.

3.4.3 Analyse de la version RT-Therm V1.1

La version RT-Therm V1.1 constitue une étape intermédiaire clé dans l'évolution du projet, faisant le lien entre l'architecture séquentielle de RT-Therm V1 et l'architecture temps réel optimisée de RT-Therm V2. Elle introduit l'utilisation d'un système d'exploitation temps réel (FreeRTOS) et le découpage fonctionnel en tâches distinctes, tout en conservant une approche volontairement simplifiée des mécanismes de synchronisation.

Par rapport à RT-Therm V1, cette version permet une exécution concurrente des fonctions d'acquisition, de traitement et d'affichage, ainsi qu'un premier découplage temporel entre ces fonctions grâce aux files d'attente. Les queues jouent alors un double rôle : transport des données et synchronisation implicite entre les tâches.

En revanche, contrairement à RT-Therm V2, la version V1.1 ne dissocie pas explicitement le transport de la donnée de la synchronisation événementielle. L'absence de sémaphores limite les possibilités d'optimisation de la réactivité et du comportement temporel fin. L'introduction des sémaphores dans RT-Therm V2 répond à ces limitations en permettant une synchronisation plus précise et une meilleure séparation des responsabilités entre mécanismes RTOS.

Cette progression V1 → V1.1 → V2 sert de fil conducteur à l'analyse comparative présentée dans la suite de ce chapitre.

3.4.4 Analyse de RT-Therm V2

Ordonnancement préemptif et maîtrise temporelle

RT-Therm V2 repose sur un noyau FreeRTOS assurant un ordonnancement préemptif basé sur les priorités. Cela introduit une capacité essentielle : **isoler temporellement les responsabilités** et contrôler le comportement global via des primitives RTOS.

L'acquisition est la seule tâche périodique, ce qui permet :

- de fixer précisément la **fréquence d'échantillonnage** ;
- de réduire la **gigue** en évitant que l'affichage ou le traitement ne perturbent la période d'acquisition ;

- d'obtenir un système où le temps est **explicitement modélisé** par le RTOS (tick, temporisations, état des tâches).

La chaîne fonctionnelle devient alors déterminée par les événements (messages dans les files) plutôt que par un parcours monolithique.

Modèle événementiel et efficacité CPU

Le choix de supprimer les temporisations sur les tâches de traitement et d'affichage permet un fonctionnement événementiel :

- les tâches restent en état *Blocked* tant qu'aucune donnée n'est disponible ;
- l'ordonnanceur n'exécute la tâche que lorsqu'un événement survient (réception file d'attente).

Ce modèle présente des avantages directs :

- **aucun polling** : le CPU n'exécute pas de boucles d'attente ;
- **réduction de la charge CPU** : exécution uniquement sur donnée nouvelle ;
- **amélioration de la consommation** (réduction du temps actif, selon la gestion d'idle/halt).

Ainsi, RT-Therm V2 est structurellement plus efficace que V1, car le temps processeur est davantage consacré à des opérations utiles.

Cohérence des données et synchronisation par files d'attente

La communication entre tâches est assurée par les *queues* FreeRTOS, jouant un rôle à la fois de synchronisation et de transport de données. Cette approche garantit :

- une communication **thread-safe** (pas de section critique applicative nécessaire pour les données transférées) ;
- une synchronisation implicite (la réception bloque tant que la donnée n'est pas disponible) ;
- un découplage temporel : chaque tâche peut traiter à son rythme, dans la limite de la capacité de la file.

D'un point de vue ingénierie, les files d'attente constituent une solution robuste permettant d'éviter les défauts classiques de concurrence (course critique, incohérence d'état) observés lorsqu'on partage directement des variables globales.

Gestion des surcharges et dimensionnement des buffers

Une architecture par files d'attente impose néanmoins des choix de dimensionnement :

- **taille des queues** : une file trop petite risque la saturation (perte ou blocage selon stratégie) ;
- **politique en cas de retard** : si le traitement ou l'affichage devient plus lent que l'acquisition, la file accumule les messages.

Deux stratégies classiques existent :

- **traiter toutes les mesures** (file dimensionnée pour absorber la latence) ;

- **ne conserver que la dernière mesure** (écrasement / overwrite) afin de limiter la latence d'affichage.

Dans le cas d'un thermomètre destiné à l'affichage utilisateur, la priorité est souvent la **fraîcheur de la donnée** plutôt que l'historique complet, ce qui peut justifier une file de petite taille ou une stratégie d'écrasement contrôlée.

Accès aux périphériques et sérialisation des I/O

En V2, la séparation des tâches permet également de mieux maîtriser l'accès aux périphériques :

- l'ADC est accédé uniquement par la tâche d'acquisition ;
- l'afficheur est accédé uniquement par la tâche d'affichage.

Cette organisation réduit les risques de contention matérielle et simplifie la validation, car chaque périphérique a un unique propriétaire logiciel (modèle *single-writer/single-owner*). Cela évite, par conception, l'usage intensif de mutex sur les périphériques.

3.4.5 Comparaison synthétique (niveau système)

- **Temps réel** : V2 offre une meilleure maîtrise du déterminisme (périodicité d'acquisition contrôlée, jitter réduit), contrairement à V1 où l'exécution séquentielle introduit des latences variables.
- **Performance** : V2 est plus efficace grâce au modèle événementiel (tâches bloquées hors activité). V1 peut gaspiller des cycles CPU via temporisations bloquantes.
- **Évolutivité** : V2 supporte mieux l'ajout de fonctionnalités (nouvelle tâche de communication UART, filtrage numérique, logging) sans déstabiliser la chaîne principale, car l'ordonnanceur gère la concurrence. V1 augmente son chemin critique à chaque ajout.
- **Robustesse** : V2 limite les erreurs de concurrence via files d'attente et séparation stricte des accès périphériques. V1 est plus sensible aux effets de bord (variables globales, dépendance d'ordre d'exécution).

3.4.6 Limites et améliorations possibles de RT-Therm V2

Malgré ses avantages, RT-Therm V2 nécessite une attention particulière sur plusieurs points :

- **choix des priorités** : une priorité trop faible pour l'acquisition peut introduire une dérive temporelle en cas de surcharge ; inversement une priorité trop élevée peut affamer l'affichage.
- **dimensionnement mémoire** : création de plusieurs piles (stacks) de tâches et buffers de queues, impactant la RAM disponible.
- **débogage temps réel** : la concurrence rend l'analyse plus complexe ; l'instrumentation (trace, mesures de charge, logs horodatés) devient nécessaire.
- **gestion d'erreurs** : prise en compte des cas d'échec d'envoi/réception en file, des timeouts, et des conditions de saturation.

Des améliorations typiques (niveau industriel) peuvent être envisagées :

- utilisation de **notifications de tâches** (*task notifications*) lorsque seule la synchronisation est nécessaire, afin de réduire l'overhead par rapport aux queues ;
- mise en place d'une **stratégie anti-retard** (drop/overwrite de mesures) pour garantir la fraîcheur des données affichées ;
- mesure du **CPU load** via la tâche idle et instrumentation FreeRTOS ;
- ajout d'un **watchdog** logiciel ou matériel pour augmenter la sûreté de fonctionnement.

L'évolution progressive de RT-Therm V1 vers RT-Therm V1.1 puis RT-Therm V2 met en évidence l'apport croissant des mécanismes RTOS dans la structuration d'un système embarqué temps réel, depuis une approche fonctionnelle minimale jusqu'à une architecture optimisée et proche des pratiques industrielles.

Chapter 4

Conclusion

Ce travail a abouti à la réalisation et à l'analyse de trois versions d'un thermomètre embarqué. Il met en évidence la transition d'une architecture séquentielle simple vers une application temps réel multitâche plus structurée et optimisée.

La version RT-Therm V1, issue du Projet 2024, repose sur une architecture séquentielle de type *superloop*. Bien que simple à concevoir et fonctionnelle, cette approche présente des limitations importantes en termes de déterminisme temporel, de gestion de la charge processeur et d'évolutivité. Le couplage fort entre acquisition, traitement et affichage rend le système sensible aux variations de temps d'exécution et complique l'ajout de nouvelles fonctionnalités.

La version intermédiaire RT-Therm V1.1 constitue une première transition vers le temps réel. Elle introduit l'utilisation de FreeRTOS et le découpage de l'application en trois tâches distinctes, communiquant exclusivement via des files d'attente. Cette architecture permet déjà un découplage fonctionnel et temporel significatif par rapport à V1, tout en restant conforme aux contraintes du sujet. Toutefois, l'utilisation des files d'attente comme unique mécanisme de synchronisation limite les possibilités d'optimisation fine de la réactivité et du comportement temporel.

La version RT-Therm V2 apporte une évolution supplémentaire en enrichissant le système multitâche par l'introduction de sémaphores binaires en complément des files d'attente. Cette séparation explicite entre transport de données et synchronisation événementielle améliore la lisibilité de l'architecture, la réactivité du système et la maîtrise des contraintes temporelles. La centralisation de la gestion temporelle autour de la tâche d'acquisition, combinée à un fonctionnement événementiel des tâches de traitement et d'affichage, conduit à une réduction de la charge CPU et à un comportement plus déterministe.

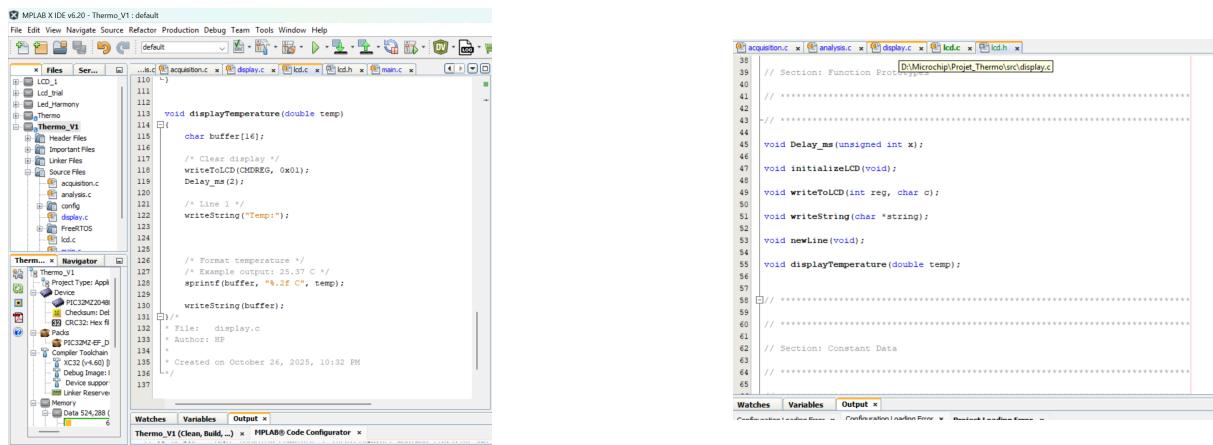
D'un point de vue ingénierie des systèmes embarqués, ce projet met en évidence l'intérêt des architectures multitâches et des mécanismes RTOS pour les applications soumises à des contraintes temporelles. Il illustre également les compromis inhérents à l'utilisation d'un RTOS, notamment en termes de complexité de conception, de dimensionnement mémoire et de débogage. Les perspectives d'amélioration envisagées, telles que l'optimisation des mécanismes de synchronisation, la gestion des surcharges ou l'instrumentation du système, rapprochent cette architecture des pratiques industrielles.

En conclusion, la progression de V1 vers V1.1 puis V2 constitue une démarche complète et formatrice, offrant une compréhension approfondie des principes fondamentaux des systèmes temps réel embarqués et de leur mise en œuvre sur microcontrôleur. Elle fournit une base solide pour la conception de systèmes embarqués plus complexes intégrant des exigences accrues en termes de déterminisme, de fiabilité et de performance.

Appendix A

Optimisation

During la configuration une nouvelle fonction a été rajouter au fichier lcd(.c et .h) pour pouvoir faciliter la l'affichage et le delays à été modifier pour mettre la frequence d'affichage du LCD a une quasiment similaire a l'envoie des données pour l'affichage comme on peut voir dans la Figure A.1



The screenshot shows the MPLAB X IDE interface. The top menu bar includes File, Edit, View, Navigate, Source, Refactor, Production, Debug, Team, Tools, Window, Help. The main window has tabs for acquisition.c, display.c, lcd.c, lcd.h, and man.c. The project navigator on the left lists files like LCD.h, Led.h, Led_Harmony, and Thermo_V1. The code editor on the right contains the display.c and lcd.h files. The display.c file includes functions for initializing the LCD, writing to it, and displaying temperature. The lcd.h file defines the LCD library.

```
acquisition.c x analysis.c x display.c x lcd.c x lcd.h x
39 // Section: Function Prototypes
40
41 // ****
42
43 // ****
44
45 void Delay_ms(unsigned int x);
46
47 void initializeLCD(void);
48
49 void writeToLCD(int reg, char c);
50
51 void writeString(char *string);
52
53 void newline(void);
54
55 void displayTemperature(double temp);
56
57 // ****
58 // Section: Constant Data
59
60 // ****
61
62 // Section: Function Definitions
63
64 // ****
65
```

Figure A.1: Function ajouter Display temperature

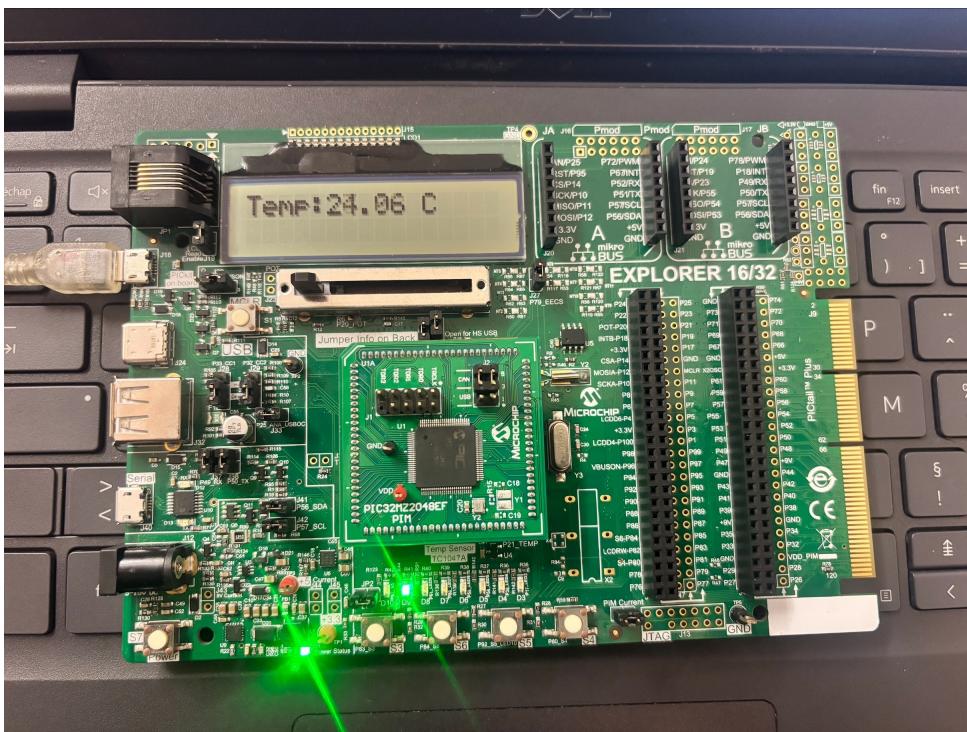


Figure A.2: Results

Appendix B

Organisation des fichiers

Le code source du projet est organisé de manière modulaire afin de refléter directement l'architecture fonctionnelle de l'application. Chaque module correspond à une tâche FreeRTOS ou à un service logiciel spécifique, ce qui facilite la lisibilité, la maintenance et l'évolution du système.

- **Fichiers de la version RT-Therm V1:** Constituer des fichiers main.c , lcd.c / lcd.h. Disponible dans le GitHub dans le document ProjetThermoX

B.1 Final results

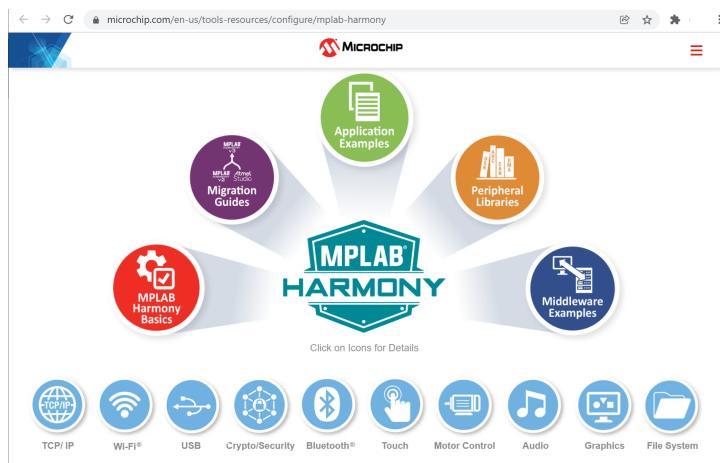


Figure B.1: MPLab Harmony

- **Fichiers de la version RT-Therm V2:** Constituer des fichiers acquisition.c / acquisition.h , analysis.c / analysis.h, display.c / display.h, main.c disponible dans le GitHub dans le document.

Appendix C

Les différentes images



Figure C.1: Explorer16:32 Development Board

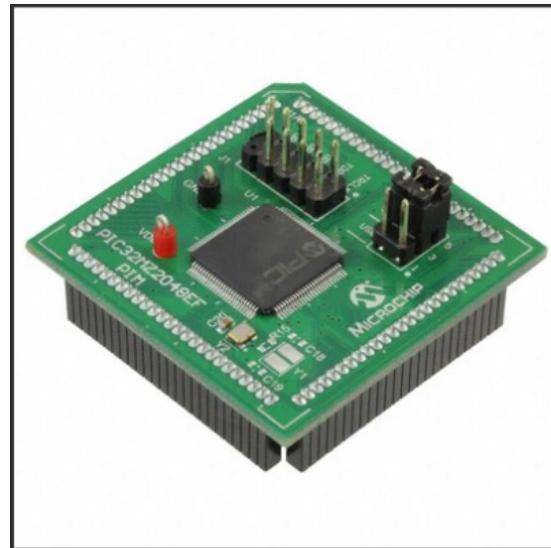


Figure C.2: PIC32MZ2048EFH100 Module

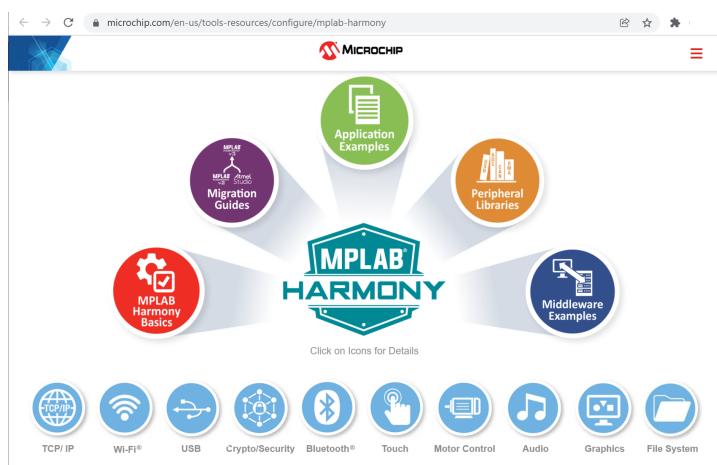


Figure C.3: MPLab Harmony



Figure C.4: freeRTOS

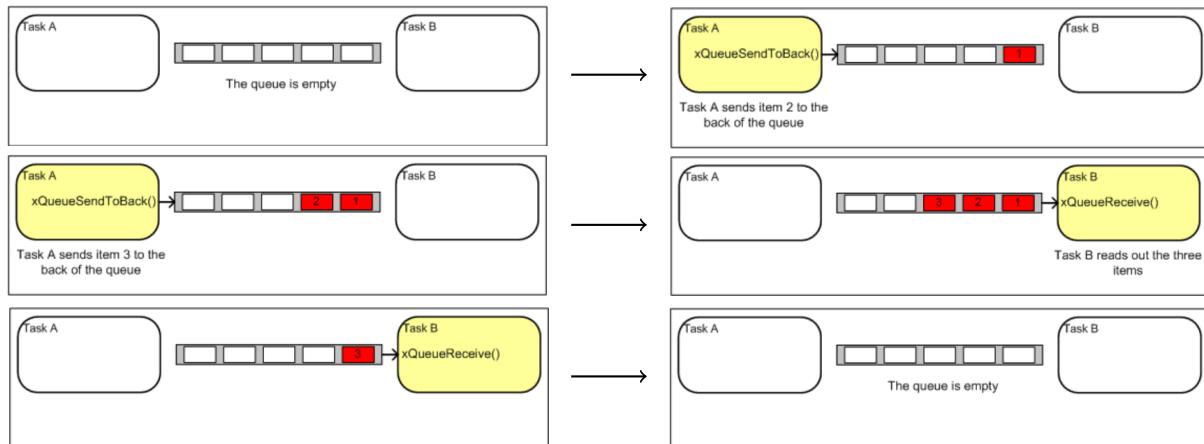


Figure C.5: Fonctionnement d'une queue

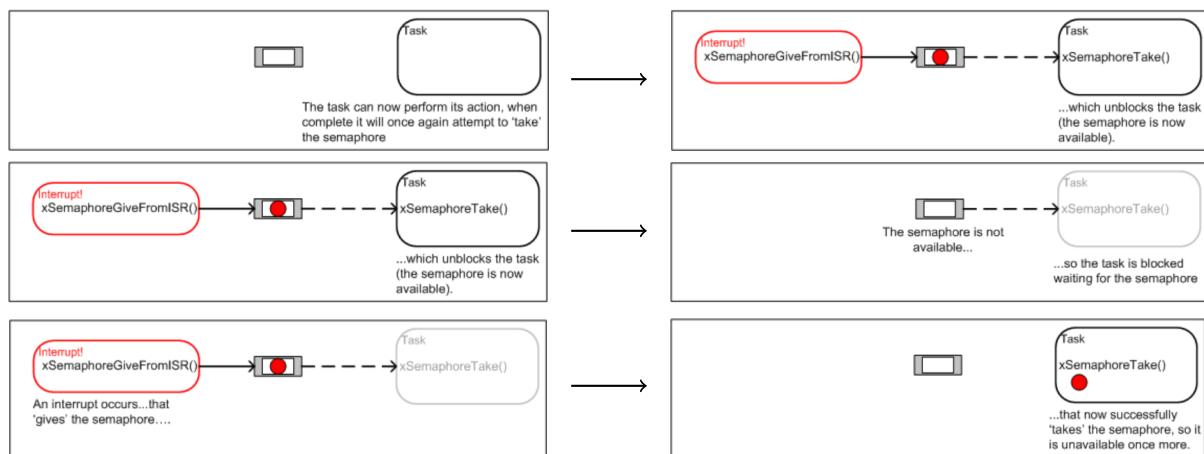


Figure C.6: Fonctionnement et échange de données dans les semaphores