Université de Douala
École Supérieure Technique La Salle

Systèmes Temps Réel

Ordonnancement, RTOS et conception
d'applications multitâches embarquées

Support de cours

Classe : **IIA5**

Volume horaire : 36 h (CM 20 h — TD 8 h — TP 8 h)

Pr Moffo Lonla Bertrand

Année académique 2026–2027

Table des matières

Présentation du cours **3**

1 Introduction aux systèmes temps réel **5**
1.1 Pourquoi le « temps réel » ? . . . 5
1.2 Un système temps réel est-il un système rapide ? . . . 6
1.3 Classes de contraintes temporelles . . . 6
1.4 Caractéristiques d'un système temps réel . . . 7
1.5 Systèmes embarqués et systèmes temps réel . . . 7
1.6 Expression des contraintes temporelles . . . 7
1.7 Exemples de systèmes temps réel . . . 7
1.8 Structure logicielle : de la boucle infinie au multitâche . . . 8

2 Modèle de tâches et ordonnancement temps réel **9**
2.1 Notion de tâche . . . 9
2.2 Paramètres temporels d'une tâche . . . 9
2.3 L'ordonnancement . . . 10
2.3.1 Taxonomie des algorithmes . . . 10
2.3.2 Conventions pour le tracé des chronogrammes . . . 11
2.4 Algorithmes d'ordonnancement classiques (non temps réel) . . . 11
2.4.1 Premier arrivé, premier servi (FIFO / FCFS) . . . 11
2.4.2 Plus court d'abord (SJF, *Shortest Job First*) . . . 12
2.4.3 Tourniquet (*Round Robin*) . . . 12
2.4.4 Priorités fixes . . . 12
2.5 Ordonnancement temps réel de tâches périodiques indépendantes . . . 13
2.5.1 Hypothèses du modèle (Liu et Layland, 1973) . . . 13
2.5.2 Rate Monotonic (RM, ou RMA) . . . 13
2.5.3 Analyse du temps de réponse (test exact à priorités fixes) . . . 14
2.5.4 Échec de RM, réussite d'EDF . . . 14
2.5.5 Earliest Deadline First (EDF) . . . 15
2.5.6 Deadline Monotonic (DM) . . . 15
2.5.7 Least Laxity First (LLF) . . . 16
2.5.8 Synthèse . . . 16
2.6 Ordonnancement non préemptif . . . 17
2.7 Tâches apériodiques . . . 17
2.8 Ordonnancement hors ligne (cyclique) . . . 17

3 Ressources partagées, synchronisation et communication **18**
3.1 Ressources partagées et section critique . . . 18
3.1.1 La condition de course . . . 18
3.1.2 Expérimenter sur PC : les threads POSIX . . . 19
3.1.3 Propriétés attendues d'une solution . . . 20
3.1.4 Mécanismes de bas niveau . . . 21
3.2 Les sémaphores . . . 21

3.3 Le producteur et le consommateur . . . 21
3.4 Autres outils de communication et de synchronisation . . . 24
3.5 Interblocage et famine . . . 24
3.6 L'inversion de priorité . . . 24
3.6.1 Protocole de l'héritage de priorité (PIP) . . . 25
3.6.2 Protocole à plafond de priorité (PCP, ICPP) . . . 25

4 Systèmes d'exploitation temps réel et multitâche 27
4.1 Les machines à états . . . 27
4.2 Architecture d'un RTOS . . . 27
4.2.1 Le changement de contexte . . . 28
4.2.2 Les trois familles d'ordonnanceurs . . . 28
4.2.3 États d'une tâche . . . 28
4.3 Un RTOS préemptif : FreeRTOS . . . 29
4.4 Le compilateur C de CCS et son RTOS . . . 29
4.4.1 Configuration : la directive `#use rtos` . . . 30
4.4.2 Déclaration d'une tâche : la directive `#task` . . . 30
4.4.3 Les fonctions du RTOS de CCS . . . 31
4.4.4 Montage de base du PIC18F45K22 . . . 31
4.4.5 Premier exemple : quatre LED indépendantes . . . 31
4.4.6 Communication par messages . . . 32
4.4.7 Exclusion mutuelle par sémaphore . . . 33
4.4.8 Mesure des temps d'exécution . . . 35
4.4.9 Ordonnançabilité avec le RTOS de CCS . . . 36
4.5 L'environnement de développement : CCS + Proteus . . . 36
4.6 Études de cas . . . 36

5 Conception d'un système temps réel : asservissement de température 39
5.1 Démarche de conception d'un STR . . . 39
5.2 Cahier des charges . . . 39
5.3 Architecture matérielle . . . 40
5.3.1 Chaîne de mesure . . . 40
5.3.2 Chaîne d'action . . . 40
5.3.3 Schéma électronique . . . 40
5.4 Modélisation et identification du procédé . . . 41
5.5 Synthèse du régulateur PID . . . 42
5.5.1 Réglage . . . 42
5.5.2 Choix de la période d'échantillonnage . . . 42
5.5.3 Discrétisation et mise en œuvre . . . 42
5.6 Conception multitâche . . . 43
5.6.1 Découpage en tâches . . . 43
5.6.2 Choix du cycle mineur et vérification temporelle . . . 44
5.6.3 Communication et synchronisation . . . 44
5.7 Programme complet . . . 45
5.8 Validation . . . 48
5.9 Mise en pratique (TP, séances 7 à 9) . . . 48

Présentation du cours

Objectifs pédagogiques

À la fin de ce cours, vous serez capable de :

— analyser un problème multitâche temps réel en termes de **synchronisation des tâches** et de **protection des accès aux ressources partagées** ;

— vérifier, par le calcul et par le tracé de chronogrammes, qu'un ensemble de tâches respecte ses contraintes temporelles (RM, DM, EDF, LLF) ;

— proposer une solution de programmation du problème en langage C/C++ en exploitant les fonctionnalités d'un noyau temps réel multitâche (RTOS de CCS C, FreeRTOS, threads POSIX) ;

— concevoir, simuler (Proteus) et valider un système temps réel complet : l'asservissement de température d'un four à micro-ondes piloté par un PIC18F45K22.

Prérequis

Programmation en langage C, architecture et programmation des microcontrôleurs PIC (ports, timers, CAN, UART, interruptions), notions de systèmes d'exploitation et d'automatique (fonction de transfert du premier ordre, correcteur PID).

Progression 2026–2027

Séance	Date	Contenu	CM	TD	TP
1	21/11/2026	Introduction aux systèmes temps réel (ch. 1)	4		
2	28/11/2026	Introduction (fin) — Ordonnancement temps réel (ch. 2)	4		
3	05/12/2026	Ordonnancement : RM, DM, EDF, LLF (ch. 2) — TD 2, TD 3	2	2	
4	12/12/2026	Ressources partagées, synchronisation (ch. 3) — RTOS (ch. 4)	4		
5	19/12/2026	RTOS — compilateur C de CCS et RTOS, environnement Proteus — TD 4, TD 5	3	1	
6	09/01/2027	RTOS de CCS : études de cas — TD 6	3	1	
7	16/01/2027	Conception d'un STR : asservissement de température (ch. 5) — TP	3		1
8	23/01/2027	Mise en pratique : phase 1 (projet)			4
9	30/01/2027	Mise en pratique : phase 2 — TD 7, TD 8 (révision)		1	3

Organisation du document

Ch. 1

Notions fondamentales : définitions, classes de contraintes, architecture d'un système de contrôle-commande.

Ch. 2
: Modèle de tâches et ordonnancement : algorithmes classiques et temps réel, tests d'ordonnançabilité.

Ch. 3
: Ressources partagées, synchronisation et communication entre tâches : sémaphores, mutex, files de messages, inversion de priorité.

Ch. 4
: Systèmes d'exploitation temps réel et multitâche : noyau, états des tâches, FreeRTOS, RTOS du compilateur CCS C, simulation sous Proteus.

Ch. 5
: Conception d'un STR : asservissement de température d'un four à micro-ondes (PIC18F45K22), découpage en tâches, PID numérique, validation temporelle.

Les chronogrammes d'ordonnancement du cours et des corrigés de TD sont *générés par un simulateur* (`scripts/ordo.py`) ; les programmes C pour PC (threads POSIX) sont compilés et exécutés lors de la génération du document. Les programmes CCS C sont destinés au compilateur PCWH et à la simulation sous Proteus.

Évaluation

Un contrôle continu sur table (annoncé à l'avance, avant-dernière séance), le projet « asservissement de température » (rapport + démonstration sous Proteus) et l'examen de fin de semestre.

Bibliographie

— F. Cottet, E. Grolleau, *Systèmes temps réel de contrôle-commande : conception et implémentation*, Dunod, 2005.

— J. W. S. Liu, *Real-Time Systems*, Prentice Hall, 2000.

— G. C. Buttazzo, *Hard Real-Time Computing Systems*, 3e éd., Springer, 2011.

— D. Ibrahim, *Advanced PIC Microcontroller Projects in C*, Newnes, 2008 (chapitre « Multitasking and RTOS »).

— Custom Computer Services, *CCS C Compiler Reference Manual* (section RTOS).

— R. Barry, *Mastering the FreeRTOS Real Time Kernel*, Real Time Engineers Ltd.

— Microchip, *PIC18(L)F2X/4XK22 Data Sheet*, DS40001412.

Chapitre 1

Introduction aux systèmes temps réel

1.1 Pourquoi le « temps réel » ?

Un programme classique (traitement de texte, compilateur, calcul scientifique) est jugé correct s'il produit le *bon résultat*. Le moment où ce résultat est obtenu n'influe que sur le confort de l'utilisateur. Dans un grand nombre d'applications industrielles, embarquées ou de contrôle de procédés, ce n'est plus suffisant : un résultat juste mais *tardif* est un résultat faux. L'airbag qui se déploie 200 ms après le choc, la commande d'injection calculée après le passage du piston, l'alarme de surchauffe d'un four traitée après la destruction de la charge sont autant de défaillances, alors même que les calculs sont exacts.

Définition 1.1 — Système temps réel

Un **système temps réel** (STR) est un système informatique dont la correction dépend non seulement de la justesse logique des résultats qu'il produit, mais aussi de la **date** à laquelle ces résultats sont produits. Il doit réagir aux événements de son environnement (le *procédé*) dans des délais imposés par la dynamique de celui-ci.

Définition 1.2 — Système de contrôle-commande

Un **système de contrôle-commande** est un STR en interaction permanente avec un procédé physique : il *acquiert* des mesures (capteurs), *calcule* des commandes (lois de commande, logique) et les *applique* (actionneurs), le tout de façon périodique ou sur événement.

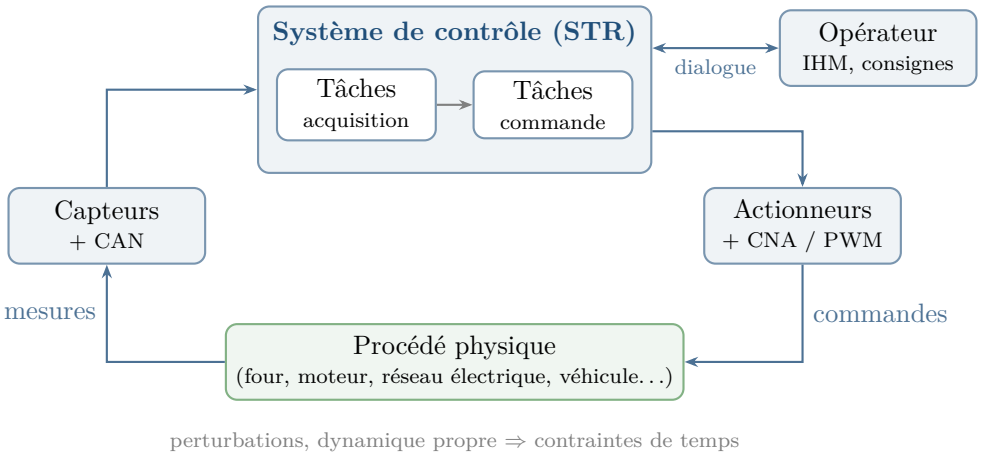


FIGURE 1.1 – Architecture générale d'un système de contrôle-commande.

Le système de contrôle (figure 1.1) observe le procédé par des **capteurs** (thermocouple, LM35, codeur, capteur de courant…) associés à un convertisseur analogique-numérique, et agit par des

actionneurs (résistance chauffante commandée en PWM, moteur, vanne…). Il dialogue souvent avec un opérateur (afficheur LCD, clavier, terminal série, supervision).

1.2 Un système temps réel est-il un système rapide ?

Attention — contresens fréquent

Temps réel ne signifie pas rapide. Un STR doit être **prévisible** (*déterministe*) : on doit pouvoir *garantir*, avant l'exécution, que chaque échéance sera respectée, y compris dans le pire cas. Un système très rapide en moyenne mais dont le temps de réponse est occasionnellement imprévisible (cache, mémoire virtuelle, ramasse-miettes, interruptions non maîtrisées) n'est *pas* temps réel.

L'échelle de temps dépend du procédé : la régulation de température d'un four (constante de temps de l'ordre de la minute) tolère une période d'échantillonnage d'une seconde, alors que le contrôle du courant d'un onduleur exige quelques dizaines de microsecondes. Un microcontrôleur à 64 MHz peut donc être « temps réel » pour le four et ne pas l'être pour l'onduleur. La rapidité aide, mais c'est la *maîtrise du pire cas* qui fait le temps réel.

1.3 Classes de contraintes temporelles

On classe les STR selon les conséquences d'un dépassement d'échéance (figure 1.2).

Définition 1.3 — Temps réel dur, ferme, souple

- **Temps réel dur** (*hard real-time*) : le non-respect d'une seule échéance peut avoir des conséquences catastrophiques (humaines, matérielles, économiques). Toutes les échéances doivent être garanties. *Ex. :* airbag, ABS, commande de vol, protection d'un réseau électrique, arrêt d'urgence.
- **Temps réel ferme** (*firm real-time*) : un résultat tardif n'a *aucune valeur* (il est jeté), mais un dépassement occasionnel est toléré sans catastrophe. *Ex. :* trame vidéo décodée trop tard, cotation boursière périmée, mesure d'un radar.
- **Temps réel souple** (*soft real-time*) : un résultat tardif garde une valeur décroissante ; on tolère des dépassements occasionnels, on cherche à minimiser les retards en moyenne. *Ex. :* streaming audio/vidéo, réservation en ligne, IHM, guichet automatique.

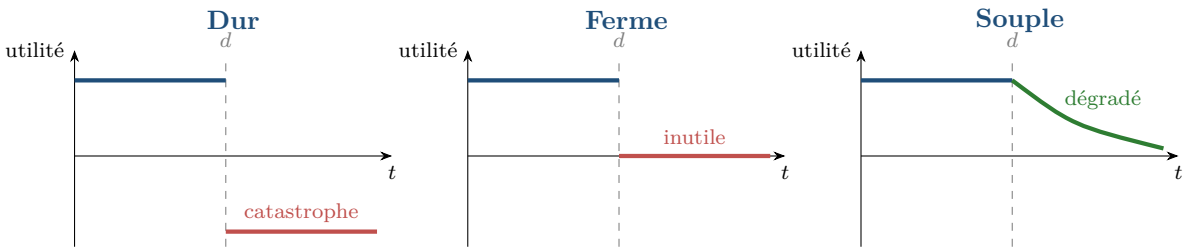


FIGURE 1.2 – Utilité d'un résultat en fonction de sa date de production (d : échéance).

1.4 Caractéristiques d'un système temps réel

À retenir

Un STR doit posséder les qualités suivantes :

- **Déterminisme / prévisibilité** : le comportement temporel (pire temps d'exécution, latence d'interruption, durée d'un appel système) est borné et connu.
- **Réactivité** : il prend en compte les événements du procédé dans les délais imposés (latence maîtrisée).
- **Respect des échéances** : garanti hors ligne (analyse d'ordonnançabilité) pour le temps réel dur.
- **Concurrence** : plusieurs activités se déroulent en parallèle (multitâche), avec synchronisation et partage de ressources.
- **Fiabilité, sûreté, tolérance aux fautes** : fonctionnement continu, souvent sans opérateur, parfois dans un environnement hostile.
- **Ressources limitées** (cas embarqué) : mémoire, énergie, puissance de calcul.

1.5 Systèmes embarqués et systèmes temps réel

Définition 1.4 — Système embarqué

Un **système embarqué** est un système électronique *et informatique* autonome, souvent temps réel, spécialisé dans une tâche bien précise et intégré au produit qu'il contrôle (voiture, téléphone, four, compteur électrique…).

Tous les systèmes embarqués ne sont pas temps réel dur (une calculatrice), et tous les STR ne sont pas embarqués (supervision d'une centrale sur un PC industriel). Mais l'intersection est très large : c'est le domaine de ce cours.

1.6 Expression des contraintes temporelles

Les contraintes s'expriment sur les **tâches** (chapitre 2) par :

- une **période** T (acquisition toutes les 100 ms, régulation toutes les secondes) ;
- une **échéance** relative D (la commande doit être appliquée moins de 5 ms après la mesure) ;
- une **date de réveil** r (arrivée d'un événement : appui sur un bouton, réception d'un caractère) ;
- une **gigue** admissible (variation de la date d'exécution d'une tâche périodique) ;
- un **temps de réponse** maximal à un événement (latence de bout en bout).

Application	Ordre de grandeur de la contrainte	Classe
Contrôle de courant d'un onduleur	10–100 µs	dur
Commande moteur (vitesse, position)	0,1–10 ms	dur
Airbag, ABS	1–10 ms	dur
Protection d'un réseau électrique (relais)	10–100 ms	dur
Régulation de température (four)	0,1–10 s	dur / ferme
Vidéo 25 images/s	40 ms	ferme / souple
Interface homme-machine	100 ms–1 s	souple

1.7 Exemples de systèmes temps réel

- **Automobile** : contrôle moteur (injection, allumage), ABS, ESP, airbag, régulateur de vitesse — des dizaines de calculateurs reliés par bus CAN.

— **Énergie** : relais de protection, régulation de tension et de fréquence, dispatching, compteurs intelligents.

— **Industrie** : automates programmables, robots, convoyeurs, régulation de procédés (température, pression, débit, niveau).

— **Aéronautique et spatial** : commandes de vol électriques, pilote automatique, contrôle d'attitude d'un satellite.

— **Médical** : pacemaker, pousse-seringue, respirateur.

— **Vie courante** : four à micro-ondes, machine à laver, climatiseur, téléphone (pile radio), distributeur de billets, ascenseur.

Exemple — un four à micro-ondes, fil rouge du cours

Le four piloté par un PIC18F45K22 que nous concevrons au chapitre 5 comporte plusieurs activités concurrentes : mesurer la température toutes les 100 ms, calculer la commande PID toutes les 500 ms, rafraîchir l'afficheur LCD, scruter le clavier, dialoguer avec un terminal série et signaler que le système vit (LED clignotante). La régulation est *temps réel dur* (une surchauffe détruit la charge), l'affichage est *souple*.

1.8 Structure logicielle : de la boucle infinie au multitâche

Trois grandes approches existent pour programmer un STR sur microcontrôleur :

1. **La boucle infinie** (*super-loop*, scrutation) : toutes les activités sont appelées séquentiellement dans `while(1)`. Simple, mais le temps de réponse à un événement dépend de la durée de toute la boucle, et une activité longue retarde toutes les autres.
2. **Boucle + interruptions** (*foreground/background*) : les événements urgents sont traités dans des routines d'interruption courtes, le reste dans la boucle de fond. Plus réactif, mais le partage de données entre interruption et boucle devient délicat.
3. **Un noyau temps réel multitâche (RTOS)** : l'application est découpée en **tâches** indépendantes ; un **ordonnanceur** décide à chaque instant quelle tâche s'exécute, et le noyau fournit les outils de synchronisation et de communication. C'est l'approche étudiée aux chapitres 2 à 4.

À retenir

— Un STR est correct s'il donne le bon résultat *au bon moment* ; « temps réel » $\neq$ « rapide » : il faut un comportement *prévisible*.

— Trois classes : **dur** (aucun dépassement toléré), **ferme** (résultat tardif inutile), **souple** (résultat tardif dégradé).

— Les contraintes s'expriment par des périodes, échéances, dates de réveil et temps de réponse ; elles découlent de la dynamique du procédé.

Chapitre 2

Modèle de tâches et ordonnancement temps réel

2.1 Notion de tâche

Définition 2.1 — Tâche, processus, instance

Une **tâche** est une entité d'exécution séquentielle (un programme ou une fonction) qui réalise une activité de l'application : acquérir une mesure, calculer une commande, afficher. Un **processus** est un programme en cours d'exécution possédant son propre espace mémoire ; un processus peut contenir plusieurs **fils d'exécution** (*threads*) partageant sa mémoire. Dans un RTOS pour microcontrôleur, toutes les tâches partagent la même mémoire : ce sont des threads au sens strict.
Une tâche **périodique** τ_i est réveillée toutes les T_i unités de temps ; chaque réveil engendre une **instance** (ou *travail*, *job*) $\tau_{i,k}$.

On distingue :

— les tâches **périodiques** (réveil régulier : échantillonnage, régulation) ;

— les tâches **apériodiques** (réveil sur événement imprévisible, sans délai minimal entre deux réveils) ;

— les tâches **sporadiques** (réveil sur événement, mais avec un délai minimal connu entre deux réveils : on les traite comme des périodiques de période égale à ce délai minimal) ;

— les tâches **indépendantes** ou **dépendantes** (partage de ressources, relations de précédence, échanges de messages).

2.2 Paramètres temporels d'une tâche

Définition 2.2 — Modèle canonique d'une tâche périodique

Une tâche périodique est notée $\tau_i(r_i, C_i, D_i, T_i)$:

— r_i : date de premier réveil (déphasage, *offset*) ; l'instance k (à partir de 0) est réveillée à $r_{i,k} = r_i + k\,T_i$;

— C_i : durée d'exécution **pire cas** (WCET, *Worst-Case Execution Time*), mesurée en temps processeur ;

— D_i : échéance (délai critique) **relative** ; l'échéance absolue de l'instance k est $d_{i,k} = r_{i,k} + D_i$;

— T_i : période.

On suppose $0 < C_i \leq D_i \leq T_i$. Si $D_i = T_i$, on parle de tâche à **échéance sur requête**.

Lors de l'exécution, on observe pour chaque instance (figure 2.1) :

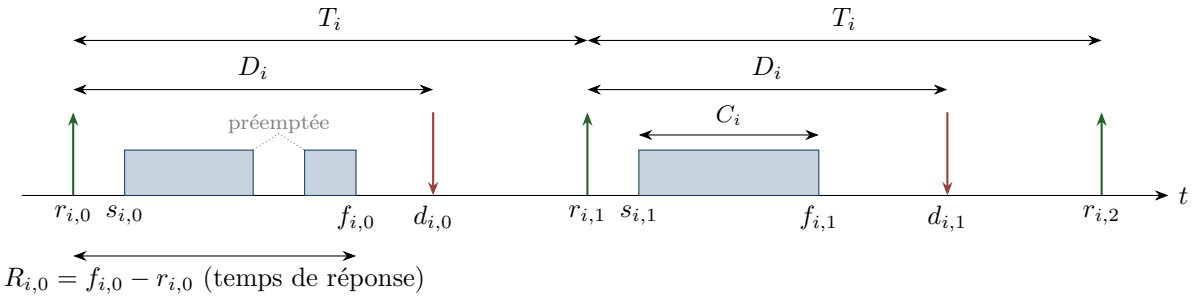


FIGURE 2.1 – Paramètres temporels d'une tâche périodique τ_i (deux premières instances).

— $s_{i,k}$: date de début d'exécution ; $f_{i,k}$: date de fin d'exécution ;

— $R_{i,k} = f_{i,k} - r_{i,k}$: **temps de réponse** de l'instance ; le pire temps de réponse de la tâche est $R_i = \max_k R_{i,k}$;

— la tâche respecte ses échéances si et seulement si $R_i \leq D_i$.

Définition 2.3 — Laxité, facteur d'utilisation, hyperpériode

— **Laxité** (marge) d'une instance à l'instant t : $L(t) = d - t - C_{\text{restant}}(t)$. C'est le temps dont on peut encore retarder l'instance sans manquer son échéance. Une laxité négative signifie une échéance forcément manquée.

— **Facteur d'utilisation** de τ_i : $u_i = C_i/T_i$; charge totale du processeur : $U = \sum_{i=1}^{n} C_i/T_i$.

— **Hyperpériode** : $H = \text{ppcm}(T_1, \ldots, T_n)$. Pour des tâches périodiques synchrones (tous les $r_i = 0$), l'ordonnancement se répète à l'identique toutes les H unités de temps : il suffit de l'étudier sur $[0, H]$.

Théorème 2.4 — Condition nécessaire

Un ensemble de tâches périodiques ne peut être ordonnancé sur un processeur que si $U \leq 1$. Si $U > 1$, aucun algorithme ne peut respecter toutes les échéances.

2.3 L'ordonnancement

Définition 2.5 — Ordonnanceur

L'**ordonnanceur** (*scheduler*) est la partie du noyau qui choisit, à chaque instant de décision, quelle tâche prête obtient le processeur. La règle qu'il applique est l'**algorithme** (ou politique) d'ordonnancement ; la suite des choix obtenue est une **séquence d'ordonnancement**, représentée par un **chronogramme** (diagramme de Gantt).

L'objectif d'un ordonnanceur classique (OS de bureau) est d'optimiser des critères *moyens* : débit, équité, temps de réponse moyen. L'objectif d'un ordonnanceur temps réel est de **respecter les contraintes temporelles de toutes les tâches** (ou du plus grand nombre, en privilégiant les plus critiques), et de pouvoir le **prouver** avant l'exécution.

2.3.1 Taxonomie des algorithmes

— **Préemptif / non préemptif** : un algorithme préemptif peut interrompre la tâche en cours au profit d'une tâche plus prioritaire ; un algorithme non préemptif (coopératif) laisse chaque instance s'exécuter jusqu'à sa fin (ou jusqu'à ce qu'elle rende la main).

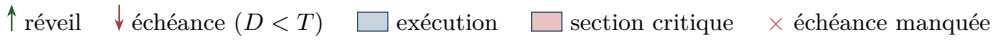
— **Hors ligne / en ligne** : hors ligne, la séquence est construite *avant* l'exécution et stockée dans une table (ordonnancement cyclique) ; en ligne, les décisions sont prises *pendant* l'exécution.

— **Priorités fixes / dynamiques** : à priorités fixes, la priorité d'une tâche ne change pas (RM, DM) ; à priorités dynamiques, elle dépend de l'instant ou de l'instance (EDF, LLF).

— **Optimal / non optimal** : un algorithme est optimal (dans une classe) s'il trouve une séquence valide chaque fois qu'il en existe une dans cette classe.

2.3.2 Conventions pour le tracé des chronogrammes

Remarque — conventions du cours Le temps est discret (unité : le « top » d'horloge du noyau). Une décision est prise à chaque top. À priorité égale, on **garde la tâche en cours** (pas de préemption inutile), sinon on choisit la tâche d'indice le plus faible. Une instance qui dépasse son échéance n'est pas abandonnée : l'échéance manquée est signalée par ×.



2.4 Algorithmes d'ordonnancement classiques (non temps réel)

Ces algorithmes ne prennent pas en compte les échéances ; on les évalue sur des tâches à exécution unique (date d'arrivée a_i, durée C_i) par les critères suivants.

Définition 2.6 — Critères de performance

— **Temps de rotation** (*turnaround*) : $\text{Rot}_i = f_i - a_i$ (fin d'exécution − date d'arrivée).

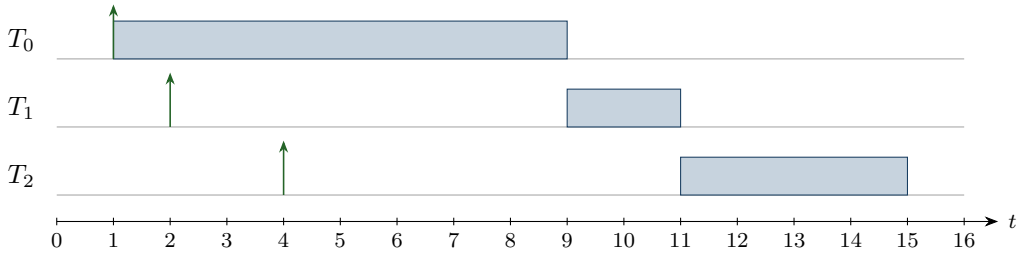
— **Temps d'attente** : $\text{Att}_i = \text{Rot}_i - C_i$ (temps passé à l'état prêt).

— **Temps de réponse** (au sens des OS interactifs) : $s_i - a_i$ (délai avant le premier accès au processeur).

— **Débit** : nombre de tâches terminées par unité de temps = n/(date de fin de la dernière tâche − date de la première arrivée).

2.4.1 Premier arrivé, premier servi (FIFO / FCFS)

Non préemptif : les tâches sont servies dans l'ordre d'arrivée. Simple et équitable, mais une tâche longue retarde toutes les suivantes (*effet convoi*). Exemple : T_0 (8, arrivée 1), T_1 (2, arrivée 2), T_2 (4, arrivée 4).

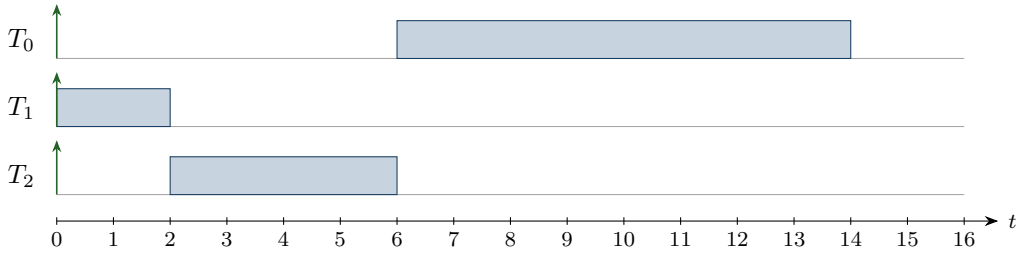


FIFO

Tâche	Durée	Arrivée	Début	Fin	Rotation	Attente	Réponse
T_0	8	1	1	9	8	0	0
T_1	2	2	9	11	9	7	7
T_2	4	4	11	15	11	7	7
Moyenne					9,3	4,7	4,7

2.4.2 Plus court d'abord (SJF, *Shortest Job First*)

Non préemptif : parmi les tâches prêtes, on choisit celle de plus petite durée. SJF minimise le temps d'attente moyen, mais suppose les durées connues et peut *affamer* les tâches longues. Sa version préemptive est SRTF (*Shortest Remaining Time First*). Exemple : trois tâches arrivées à $t = 0$, de durées 8, 2 et 4.

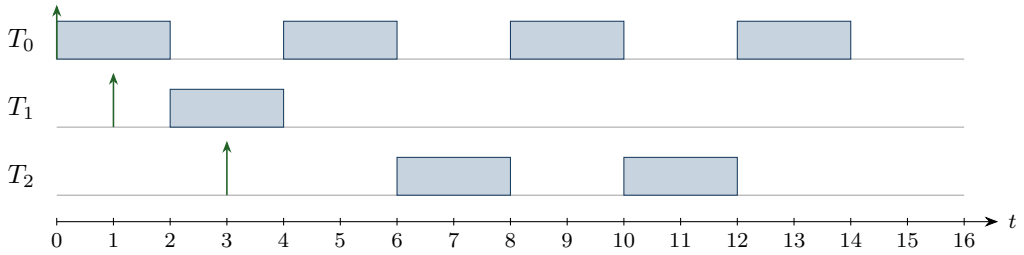


SJF (toutes les tâches arrivent à $t = 0$)

Tâche	Durée	Arrivée	Début	Fin	Rotation	Attente	Réponse
T_0	8	0	6	14	14	6	6
T_1	2	0	0	2	2	0	0
T_2	4	0	2	6	6	2	2
Moyenne					7,3	2,7	2,7

2.4.3 Tourniquet (*Round Robin*)

Préemptif : chaque tâche reçoit à tour de rôle un **quantum** de temps q ; à l'expiration du quantum, elle est replacée en fin de file. Équitable, adapté au temps partagé, mais le temps de réponse dépend du nombre de tâches et non de leur urgence. Exemple avec $q = 2$: T_0 (8, 0), T_1 (2, 1), T_2 (4, 3).

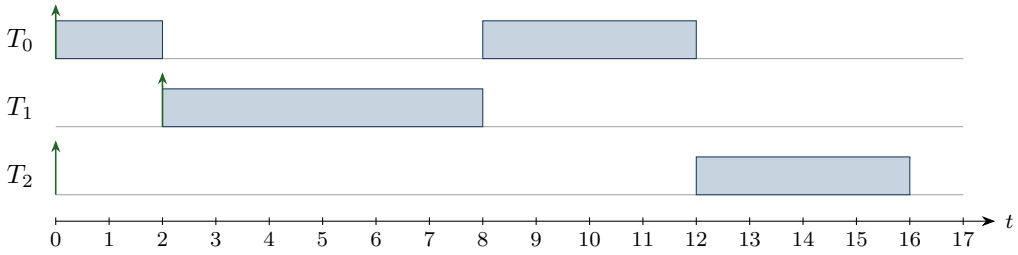


Tourniquet, quantum $q = 2$ (à un même instant, l'arrivée passe avant la tâche dont le quantum expire)

Tâche	Durée	Arrivée	Début	Fin	Rotation	Attente	Réponse
T_0	8	0	0	14	14	6	0
T_1	2	1	2	4	3	1	1
T_2	4	3	6	12	9	5	3
Moyenne					8,7	4,0	1,3

2.4.4 Priorités fixes

Chaque tâche reçoit une priorité (ici : 1 = la plus forte) ; la tâche prête la plus prioritaire s'exécute, avec ou sans préemption. Exemple préemptif : T_0 (6, arrivée 0, prio 2), T_1 (6, arrivée 2, prio 1), T_2 (4, arrivée 0, prio 3).



Priorités fixes préemptives : T_1 préempte T_0 à $t = 2$

Tâche	Durée	Arrivée	Début	Fin	Rotation	Attente	Réponse
T_0	6	0	0	12	12	6	0
T_1	6	2	2	8	6	0	0
T_2	4	0	12	16	16	12	12
Moyenne					11,3	6,0	4,0

L'ordonnancement par priorités est la base de tous les ordonnanceurs temps réel ; reste à savoir *comment attribuer les priorités* pour garantir les échéances.

2.5 Ordonnancement temps réel de tâches périodiques indépendantes

2.5.1 Hypothèses du modèle (Liu et Layland, 1973)

1. les tâches sont périodiques, de périodes T_i connues ;
2. elles sont indépendantes (pas de ressource partagée, pas de précédence) ;
3. les durées C_i (pire cas) sont connues et constantes ;
4. l'échéance est égale à la période ($D_i = T_i$) — hypothèse levée par DM ;
5. les coûts de l'ordonnanceur et des changements de contexte sont négligés ;
6. l'ordonnancement est préemptif, sur un seul processeur.

2.5.2 Rate Monotonic (RM, ou RMA)

Définition 2.7 — Rate Monotonic

RM attribue des **priorités fixes** inversement proportionnelles à la période : plus la période est courte (plus la fréquence, *rate*, est élevée), plus la tâche est prioritaire. L'ordonnancement est préemptif.

Théorème 2.8 — Optimalité de RM

Pour des tâches périodiques indépendantes, synchrones, à échéance sur requête ($D_i = T_i$), RM est **optimal parmi les algorithmes à priorités fixes** : si un jeu de tâches est ordonnançable par une affectation de priorités fixes quelconque, il l'est par RM.

Théorème 2.9 — Condition suffisante de Liu et Layland

Un ensemble de n tâches périodiques indépendantes avec $D_i = T_i$ est ordonnançable par RM si

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \;\leq\; U_{\mathrm{LL}}(n) = n\left(2^{1/n} - 1\right).$$

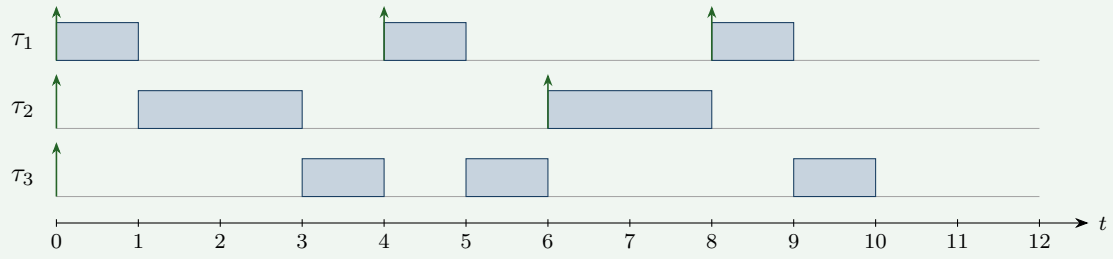
n	1	2	3	4	5	6	10	∞
$U_{\mathrm{LL}}(n)$	1	0,828	0,780	0,757	0,743	0,735	0,718	$\ln 2 \approx 0{,}693$

Attention — condition suffisante, mais pas nécessaire

Si $U \leq U_{\mathrm{LL}}(n)$, le jeu est ordonnançable par RM. Si $U_{\mathrm{LL}}(n) < U \leq 1$, **on ne peut pas conclure** : le jeu peut être ordonnançable ou non ; il faut un test exact (analyse du temps de réponse ou chronogramme sur l'hyperpériode). Si $U > 1$, le jeu n'est ordonnançable par aucun algorithme.

Exemple — test de Liu et Layland non concluant

$\tau_1(C=1,T=4)$, $\tau_2(C=2,T=6)$, $\tau_3(C=3,T=12)$: $U = \frac{1}{4}+\frac{2}{6}+\frac{3}{12} = 0{,}833 > U_{\mathrm{LL}}(3) = 0{,}780$. Le test ne conclut pas ; le chronogramme sur $H = \mathrm{ppcm}(4,6,12) = 12$ montre que toutes les échéances sont respectées (τ_3 se termine à $t = 10 \leq 12$).



RM, $H = 12$: ordonnançable bien que $U > U_{LL}(3)$

2.5.3 Analyse du temps de réponse (test exact à priorités fixes)

Pour des tâches synchrones, le pire temps de réponse d'une tâche est obtenu pour sa première instance (*instant critique* : toutes les tâches plus prioritaires sont réveillées en même temps qu'elle).

Théorème 2.10 — Joseph et Pandya, 1986

Soit $hp(i)$ l'ensemble des tâches plus prioritaires que τ_i. Le pire temps de réponse R_i est la plus petite solution de

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j,$$

calculée par itération : $R_i^{(0)} = C_i + \sum_{j\in hp(i)} C_j$, puis $R_i^{(k+1)} = C_i + \sum_{j\in hp(i)} \lceil R_i^{(k)}/T_j \rceil C_j$, jusqu'à $R_i^{(k+1)} = R_i^{(k)}$ (convergence) ou $R_i^{(k+1)} > D_i$ (échec). Le jeu est ordonnançable si et seulement si $R_i \leq D_i$ pour tout i.

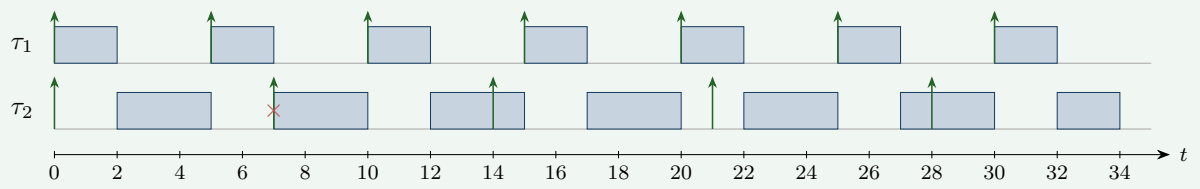
Exemple — suite

$R_1 = 1$; R_2 : $2 + \lceil 3/4 \rceil \cdot 1 = 3$ (stable) ; R_3 : $R^{(0)} = 3+1+2 = 6$, $R^{(1)} = 3 + \lceil 6/4 \rceil \cdot 1 + \lceil 6/6 \rceil \cdot 2 = 7$, $R^{(2)} = 3 + 2 + 2 \cdot 2 = 9$, $R^{(3)} = 3 + 3 + 2 \cdot 2 = 10$, $R^{(4)} = 3 + \lceil 10/4 \rceil + \lceil 10/6 \rceil \cdot 2 = 10$: stable, $R_3 = 10 \leq 12$. Ordonnançable, conformément au chronogramme.

2.5.4 Échec de RM, réussite d'EDF

Exemple — $U = 0{,}971$

$\tau_1(2,5)$, $\tau_2(4,7)$: $U = 0{,}4 + 0{,}571 = 0{,}971 \leq 1$. Par RM, τ_1 est prioritaire. Analyse : $R_2^{(0)} = 6$, $R_2^{(1)} = 4 + \lceil 6/5 \rceil \cdot 2 = 8 > 7$: **échec**. Le chronogramme confirme : la première instance de τ_2 se termine à $t = 8$ alors que son échéance est 7.



RM : τ_2 manque son échéance à $t = 7$

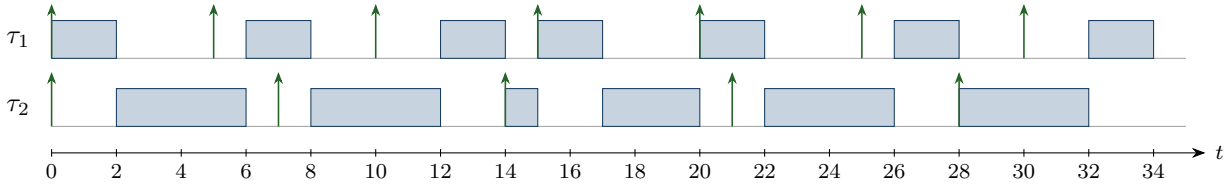
2.5.5 Earliest Deadline First (EDF)

Définition 2.11 — EDF

EDF attribue à chaque instant la plus forte priorité à l'instance prête dont l'**échéance absolue est la plus proche**. Les priorités sont **dynamiques** (elles dépendent de l'instance).

Théorème 2.12 — Condition nécessaire et suffisante d'EDF

Pour des tâches périodiques indépendantes à échéance sur requête ($D_i = T_i$), EDF préemptif respecte toutes les échéances **si et seulement si** $U \leq 1$. EDF est **optimal** parmi tous les algorithmes (monoprocesseur, préemptif). Si $D_i < T_i$, la condition $\sum C_i/D_i \leq 1$ est suffisante (non nécessaire).



EDF sur le même jeu : toutes les échéances respectées sur $H = 35$

À $t = 5$, τ_1 (échéance 10) est réveillée alors que τ_2 (échéance 7) s'exécute : EDF ne préempte pas τ_2, contrairement à RM. C'est ce qui sauve l'échéance de τ_2.

2.5.6 Deadline Monotonic (DM)

Lorsque les échéances sont contraintes ($D_i < T_i$), RM n'est plus optimal.

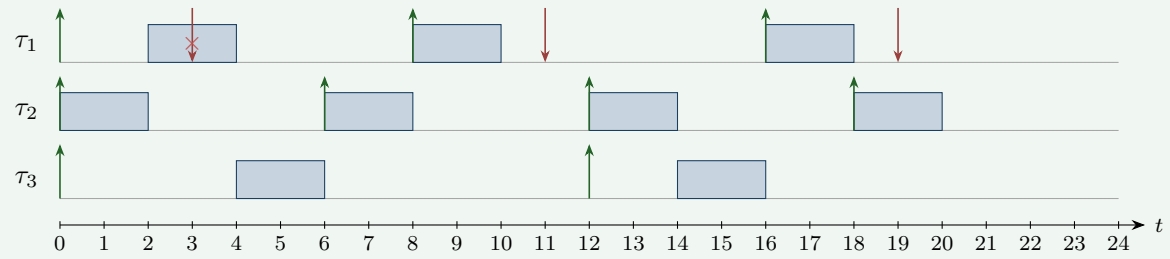
Définition 2.13 — Deadline Monotonic

DM attribue des priorités fixes inversement proportionnelles à l'**échéance relative** D_i : plus D_i est court, plus la tâche est prioritaire. Si $D_i = T_i$ pour tout i, DM et RM coïncident. DM est optimal parmi les algorithmes à priorités fixes lorsque $D_i \leq T_i$.

Le test d'ordonnançabilité est l'analyse du temps de réponse (avec l'ordre de priorité DM) ; la condition suffisante $\sum C_i/D_i \leq n(2^{1/n} - 1)$ est très pessimiste.

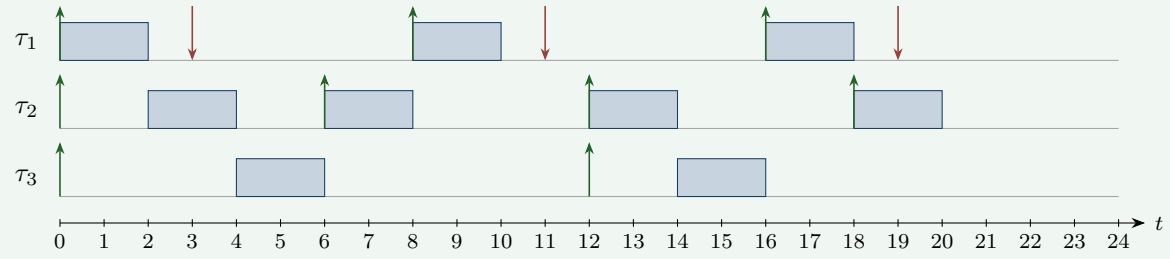
Exemple — RM échoue, DM réussit

$\tau_1(C = 2, D = 3, T = 8)$, $\tau_2(C = 2, D = 6, T = 6)$, $\tau_3(C = 2, D = 12, T = 12)$, $U = 0{,}75$. Par RM, τ_2 ($T = 6$) est plus prioritaire que τ_1 ($T = 8$) : τ_1 ne commence qu'à $t = 2$ et finit à $t = 4 > 3$.



RM : τ_1 manque son échéance à $t = 3$

Par DM, τ_1 ($D = 3$) passe en tête : $R_1 = 2 \leq 3$, $R_2 = 4 \leq 6$, $R_3 = 6 \leq 12$.



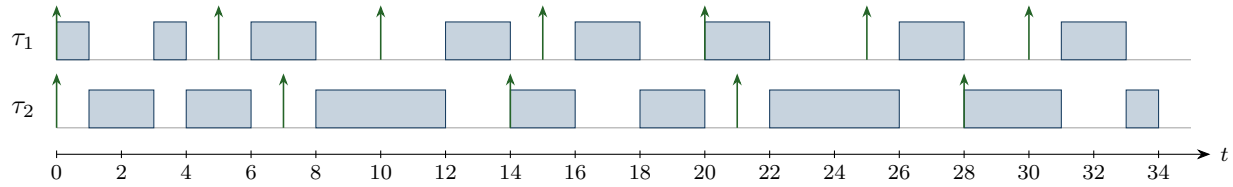
DM : ordonnançable

2.5.7 Least Laxity First (LLF)

Définition 2.14 — LLF

LLF (ou *Least Slack Time*) donne à chaque instant la plus forte priorité à l'instance prête de **plus petite laxité** $L(t) = d - t - C_{\text{restant}}(t)$. Les priorités sont dynamiques et évoluent *pendant* l'exécution : la laxité d'une tâche qui attend diminue, celle d'une tâche qui s'exécute reste constante.

LLF est, comme EDF, optimal en monoprocesseur ($U \leq 1$ suffit si $D_i = T_i$), mais il provoque beaucoup plus de **préemptions** (changements de contexte coûteux), en particulier lorsque deux tâches ont des laxités voisines.



LLF sur le même jeu : valide, mais 4 préemptions au lieu d'une seule avec EDF

Exemple — calcul des laxités à $t = 0$ et $t = 1$

À $t = 0$: $L_1 = 5 - 0 - 2 = 3$, $L_2 = 7 - 0 - 4 = 3$: égalité, τ_1 (indice le plus faible) s'exécute. À $t = 1$: $L_1 = 5 - 1 - 1 = 3$, $L_2 = 7 - 1 - 4 = 2$: τ_2 préempte τ_1.

2.5.8 Synthèse

Algorithme	Priorité	Critère	Test d'ordonnançabilité	Optimalité
RM	fixe	T_i croissant	$U \leq n(2^{1/n} - 1)$ (suff.), RTA (exact)	priorités fixes, $D = T$
DM	fixe	D_i croissant	RTA (exact)	priorités fixes, $D \leq T$
EDF	dynamique	$d_{i,k}$ le plus proche	$U \leq 1$ (n.&s., $D = T$)	optimal
LLF	dynamique	laxité minimale	$U \leq 1$ (n.&s., $D = T$)	optimal, coûteux

En pratique, les RTOS industriels (FreeRTOS, VxWorks, μC/OS, RTEMS…) implémentent un ordonnancement **préemptif à priorités fixes** (le concepteur applique RM ou DM pour choisir les

priorités), avec tourniquet entre tâches de même priorité. EDF est disponible dans certains noyaux (Linux `SCHED_DEADLINE`, ERIKA). Le RTOS du compilateur CCS est **coopératif** (non préemptif) et cadencé par le temps (section 4.4).

2.6 Ordonnancement non préemptif

Dans un ordonnancement non préemptif, une tâche prioritaire réveillée pendant l'exécution d'une tâche moins prioritaire doit attendre la fin de celle-ci : elle subit un **blocage**. Le pire blocage de τ_i est la plus longue durée d'exécution d'une tâche moins prioritaire :

$$B_i = \max_{j \in lp(i)} C_j, \qquad R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \quad \text{(approximation usuelle).}$$

Attention

Avec un noyau coopératif (CCS RTOS, machine à états), **une tâche longue retarde toutes les autres**. Il faut découper les traitements longs en morceaux courts (`rtos_yield()`, machine à états interne) pour borner B_i.

2.7 Tâches apériodiques

RM et DM sont définis pour des tâches périodiques : une tâche *apériodique* n'a pas de période et ne peut pas recevoir de priorité RM. Pour garantir les tâches périodiques tout en servant les apériodiques, on utilise :

- le **traitement en arrière-plan** : les apériodiques s'exécutent quand le processeur est libre (priorité la plus faible) — simple, mais temps de réponse imprévisible ;
- un **serveur périodique** (serveur par scrutation, *polling server* ; serveur différé, *deferrable server* ; serveur sporadique) : une tâche périodique fictive (C_s, T_s), ordonnancée par RM, consacre sa capacité C_s aux requêtes apériodiques en attente ;
- la transformation des événements *sporadiques* (délai minimal connu) en tâches périodiques de période égale à ce délai.

Avec le RTOS de CCS, on traite un événement apériodique (appui sur un bouton, caractère reçu) en le **scrutant** depuis une tâche périodique, ou en le capturant dans une routine d'interruption courte qui positionne un drapeau lu par une tâche.

2.8 Ordonnancement hors ligne (cyclique)

Pour les systèmes critiques très simples, on peut construire hors ligne une **table** donnant l'ordre d'exécution sur l'hyperpériode, découpée en *cycles mineurs* de durée fixe, le tout étant répété (*cycle majeur* $= H$). Le RTOS de CCS reprend ce vocabulaire : `minor_cycle` y est le pas de temps de l'ordonnanceur (chapitre 4). Avantages : déterminisme total, aucun coût d'ordonnancement en ligne ; inconvénients : rigidité, table difficile à maintenir, mauvais traitement de l'apériodique.

À retenir

- Paramètres d'une tâche : r, C (pire cas), D, T ; temps de réponse $R = f - r$; laxité $L = d - t - C_{\text{rest}}$; utilisation $U = \sum C_i/T_i$; hyperpériode $H = \text{ppcm}(T_i)$.
- $U > 1$: aucun ordonnancement possible. EDF et LLF : $U \leq 1$ suffit (cas $D = T$).
- RM : $U \leq n(2^{1/n} - 1)$ est **suffisant, non nécessaire** ; sinon analyse du temps de réponse ou chronogramme sur H.
- DM généralise RM aux échéances contraintes ; LLF est optimal mais multiplie les préemptions.

Chapitre 3

Ressources partagées, synchronisation et communication

Le modèle du chapitre 2 supposait des tâches *indépendantes*. Dans une application réelle, les tâches coopèrent : la tâche de mesure fournit la température à la tâche PID, qui fournit la consigne et la mesure à la tâche d'affichage ; plusieurs tâches écrivent sur le même afficheur LCD ou la même liaison série. Ce chapitre présente les problèmes posés par ces interactions et les outils du noyau qui les résolvent.

3.1 Ressources partagées et section critique

Définition 3.1 — Ressource partagée, section critique

Une **ressource partagée** est un objet (variable globale, structure de données, périphérique : LCD, UART, bus I^2C) utilisé par plusieurs tâches. Une **section critique** est une portion de code qui accède à une ressource partagée et qui ne doit pas être exécutée par plus d'une tâche à la fois : on parle d'**exclusion mutuelle**.

3.1.1 La condition de course

L'instruction C `compteur = compteur + 1;` se traduit en plusieurs instructions machine : lecture (`LOAD`), addition (`ADD`), écriture (`STORE`). Si une tâche est préemptée entre la lecture et l'écriture, une autre tâche peut modifier la variable entre-temps, et l'une des deux mises à jour est **perdue**.

Étape	Tâche A	Tâche B	`compteur` en mémoire
1	`LOAD R, compteur` (R = 5)		5
2	*préemptée*	`LOAD R, compteur` (R = 5)	5
3		`ADD R, 1 STORE`	6
4	`ADD R, 1 STORE`		**6** (au lieu de 7)

Définition 3.2 — Condition de course

Il y a **condition de course** (*race condition*) lorsque le résultat d'un calcul dépend de l'ordre, non maîtrisé, dans lequel les tâches accèdent à une donnée partagée. Ces erreurs sont intermittentes, difficiles à reproduire et donc particulièrement dangereuses.

3.1.2 Expérimenter sur PC : les threads POSIX

> **Remarque — qu'est-ce que POSIX ?** **POSIX** (*Portable Operating System Interface*, normes IEEE 1003 / ISO/IEC 9945) définit l'interface standard entre un programme et le système d'exploitation : fichiers, processus, signaux et, depuis les extensions temps réel POSIX.1b et POSIX.1c, **threads, mutex, variables de condition, sémaphores, files de messages, horloges et ordonnancement à priorités fixes** (`SCHED_FIFO`, `SCHED_RR`). Un programme qui n'utilise que l'API POSIX se recompile sans modification sur tout système conforme : Linux, macOS, BSD, QNX, VxWorks, RTEMS… Les *threads POSIX* (« pthreads ») permettent donc d'expérimenter sur un simple PC les notions de ce chapitre, avant de les retrouver sous un autre nom dans un RTOS pour microcontrôleur.

Notion	POSIX	FreeRTOS	RTOS de CCS
Créer une tâche	`pthread_create`	`xTaskCreate`	`#task(...)`
Mutex : prendre / rendre	`pthread_mutex_lock` / `unlock`	`xSemaphoreTake` / `Give`	`rtos_wait` / `rtos_signal`
Sémaphore : P / V	`sem_wait` / `sem_post`	`xSemaphoreTake` / `Give`	`rtos_wait` / `rtos_signal`
Attendre une condition	`pthread_cond_wait`	notifications, groupes d'événements	`rtos_await`
File de messages	`mq_send` / `mq_receive`	`xQueueSend` / `Receive`	`rtos_msg_send` / `read`

Compiler et exécuter les programmes C de ce chapitre. Les sources `compteur.c` et `prodcons.c` se trouvent dans le dossier `code/cours/`. Il suffit d'un compilateur C (`gcc` ou `clang`) ; l'option `-pthread` active la bibliothèque des threads à la compilation *et* à l'édition de liens.

- **Linux** (Ubuntu, Debian…) : installer le compilateur une seule fois avec `sudo apt install build-essential`.
- **macOS** : installer les outils en ligne de commande avec `xcode-select --install` (la commande `cc` appelle `clang`).
- **Windows** : la solution recommandée est **WSL** (sous-système Linux) : dans un terminal PowerShell administrateur, `wsl --install`, redémarrer, puis dans le terminal Ubuntu `sudo apt install build-essential` et procéder comme sous Linux. Autre solution : **MSYS2** (`https://www.msys2.org`), terminal *UCRT64*, puis `pacman -S mingw-w64-ucrt-x86_64-gcc` ; `gcc -pthread` y fonctionne (bibliothèque *winpthreads*). Le compilateur de Visual Studio (MSVC) ne fournit pas les pthreads.

Dans un terminal placé dans le dossier `code/cours/` :

```
gcc -std=c11 -Wall -Wextra -O0 -pthread -o compteur compteur.c
./compteur
gcc -std=c11 -Wall -Wextra -O0 -pthread -o prodcons prodcons.c
./prodcons
```

(sous macOS, `gcc` est un alias de `clang` : les mêmes commandes fonctionnent ; sous MSYS2, l'exécutable s'appelle `compteur.exe` mais se lance aussi par `./compteur`).

> **Attention — résultats non déterministes**
>
> Une condition de course ne se manifeste pas forcément à chaque exécution : relancer `./compteur` plusieurs fois et comparer les valeurs « sans verrou ». On compile en `-O0` (sans optimisation) : avec `-O2`, le compilateur peut regrouper les incréments en registre et masquer ou modifier le phénomène. De même, l'ordre des lignes affichées par `prodcons` peut varier d'une exécution à l'autre ; seules les propriétés garanties par les sémaphores (aucune mesure perdue ni dupliquée, ordre de retrait identique à l'ordre de dépôt) sont invariantes.

Le programme suivant lance deux threads POSIX qui incrémentent chacun un million de fois la même variable, d'abord sans protection, puis avec un mutex :

compteur.c

```c
/* compteur.c -- condition de course sur une variable partagée et sa correction
 * par un mutex POSIX.  Compilation : cc -Wall -pthread -o compteur compteur.c */
#include <stdio.h>
#include <pthread.h>

#define N 1000000
static volatile long compteur = 0;              /* ressource partagée          */
static pthread_mutex_t verrou = PTHREAD_MUTEX_INITIALIZER;
static int proteger;                            /* 0 : sans verrou, 1 : avec   */

static void *tache(void *arg)
{
    (void)arg;
    for (long i = 0; i < N; i++) {
        if (proteger) pthread_mutex_lock(&verrou);   /* entrée en section critique */
        compteur = compteur + 1;                     /* LOAD, ADD, STORE           */
        if (proteger) pthread_mutex_unlock(&verrou); /* sortie de section critique */
    }
    return NULL;
}

static long essai(int avec_verrou)
{
    pthread_t t1, t2;
    compteur = 0;
    proteger = avec_verrou;
    pthread_create(&t1, NULL, tache, NULL);
    pthread_create(&t2, NULL, tache, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return compteur;
}

int main(void)
{
    printf("valeur attendue : %d\n", 2 * N);
    long v = essai(0);
    printf("sans verrou     : %ld  (%s)\n", v, v == 2 * N ? "correct par chance" : "mises à
    jour perdues");
    v = essai(1);
    printf("avec mutex      : %ld  (%s)\n", v, v == 2 * N ? "correct" : "ERREUR");
    return 0;
}
```

Exécution :

```
valeur attendue : 2000000
sans verrou     : 1003765  (mises à jour perdues)
avec mutex      : 2000000  (correct)
```

La valeur obtenue sans verrou change à chaque exécution : c'est la signature d'une condition de course.

3.1.3 Propriétés attendues d'une solution

Une solution au problème de l'exclusion mutuelle doit garantir :

1. l'**exclusion mutuelle** : au plus une tâche en section critique à la fois ;
2. le **progrès** : une tâche hors section critique ne peut empêcher une autre d'y entrer ;

3. l'**attente bornée** (absence de famine) : toute tâche qui demande l'entrée l'obtient en un temps fini — et, en temps réel, *borné et calculable* ;
4. l'absence d'hypothèse sur les vitesses relatives des tâches.

3.1.4 Mécanismes de bas niveau

— **Masquage des interruptions** : sur un monoprocesseur, interdire les interruptions (donc les préemptions) pendant la section critique. Très efficace pour des sections de quelques instructions (c'est ce que font les noyaux en interne), mais dégrade la réactivité si la section est longue. Sur PIC avec CCS : `disable_interrupts(GLOBAL)` / `enable_interrupts(GLOBAL)`.

— **Instructions atomiques** (*test-and-set*, *compare-and-swap*) : permettent de construire des verrous à attente active (*spinlocks*), utiles en multiprocesseur.

— **Un noyau coopératif** : si les tâches ne sont jamais préemptées (CCS RTOS), une section critique qui ne rend pas la main (`rtos_yield`, `rtos_wait`, `rtos_await`) est automatiquement protégée vis-à-vis des autres *tâches* — mais pas vis-à-vis des *routines d'interruption*.

3.2 Les sémaphores

Définition 3.3 — Sémaphore de Dijkstra

Un **sémaphore** S est un compteur entier non négatif associé à une file d'attente de tâches, manipulé uniquement par deux opérations **atomiques** :

— $P(S)$ (*Proberen*, `wait`, `take`) : si $S > 0$, décrémenter S et continuer ; sinon **bloquer** la tâche dans la file de S ;

— $V(S)$ (*Verhogen*, `signal`, `give`) : si des tâches sont bloquées sur S, en réveiller une ; sinon incrémenter S.

Un sémaphore initialisé à 1 est un sémaphore **binaire** (exclusion mutuelle) ; initialisé à N, c'est un sémaphore **à compteur** (N exemplaires d'une ressource) ; initialisé à 0, il sert à la **synchronisation** (attendre un événement).

```
// exclusion mutuelle                  // synchronisation (rendez-vous)
S := 1                                 E := 0
Tâche A :        Tâche B :             Tâche Mesure :      Tâche PID :
  P(S)             P(S)                  lire CAN            P(E)   // attend
  section crit.    section crit.         V(E)   // signale   calculer u
  V(S)             V(S)
```

Définition 3.4 — Mutex

Un **mutex** est un sémaphore binaire spécialisé dans l'exclusion mutuelle : il a un **propriétaire** (seule la tâche qui l'a pris peut le rendre), il peut être récursif et il implémente en général un protocole contre l'inversion de priorité (section 3.6). On utilise un mutex pour protéger une ressource et un sémaphore pour signaler un événement.

3.3 Le producteur et le consommateur

C'est le schéma de communication le plus fréquent en temps réel : une tâche *produit* des données (acquisition) à son rythme, une autre les *consomme* (traitement, affichage) au sien, à travers un **tampon borné** de N cases. Trois sémaphores suffisent :

```
places := N ; mesures := 0 ; mutex := 1
Producteur :                    Consommateur :
 répéter                         répéter
```

```
    produire x                          P(mesures)    // attendre une donnée
    P(places)   // une place ?          P(mutex)
    P(mutex)                            x := tampon[il] ; il := (il+1) mod N
    tampon[ie] := x ; ie := (ie+1) mod N
    V(mutex)                            V(mutex)
    V(mesures)  // une donnée de +      V(places)     // une place de +
                                        consommer x
```

Attention — ordre des opérations P

Inverser `P(places)` et `P(mutex)` chez le producteur provoque un **interblocage** quand le tampon est plein : le producteur, bloqué sur `places` en détenant `mutex`, empêche le consommateur de libérer une place.

Implémentation en C avec des threads POSIX (le sémaphore à compteur est reconstruit à partir d'un mutex et d'une variable de condition, ce qui montre comment un noyau le réalise) :

prodcons.c

```c
/* prodcons.c -- producteur / consommateur sur un tampon circulaire borné.
 * Sémaphores à compteur construits avec un mutex et une variable de condition
 * (les sémaphores POSIX anonymes ne sont pas disponibles sous macOS).
 * Compilation : cc -Wall -pthread -o prodcons prodcons.c */
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

/* ---------- sémaphore à compteur : P (wait) et V (signal) ---------- */
typedef struct {
    int valeur;
    pthread_mutex_t m;
    pthread_cond_t  c;
} semaphore;

static void sem_init_(semaphore *s, int v)
{
    s->valeur = v;
    pthread_mutex_init(&s->m, NULL);
    pthread_cond_init(&s->c, NULL);
}
static void P(semaphore *s)                 /* attendre une unité        */
{
    pthread_mutex_lock(&s->m);
    while (s->valeur == 0)                  /* bloquer tant que 0        */
        pthread_cond_wait(&s->c, &s->m);
    s->valeur--;
    pthread_mutex_unlock(&s->m);
}
static void V(semaphore *s)                 /* rendre une unité          */
{
    pthread_mutex_lock(&s->m);
    s->valeur++;
    pthread_cond_signal(&s->c);             /* réveiller un éventuel bloqué */
    pthread_mutex_unlock(&s->m);
}

/* ---------- tampon borné ---------- */
#define TAILLE 4
#define NB_MESURES 8
static int tampon[TAILLE];
static int ie = 0, il = 0;                  /* indices d'écriture / lecture */
static semaphore places, mesures, mutex;    /* places libres, mesures dispo, exclusion */

static void *producteur(void *arg)          /* ex. : tâche d'acquisition CAN */
```

```c
{
    (void)arg;
    for (int k = 1; k <= NB_MESURES; k++) {
        int mesure = 20 + k;                /* température simulée          */
        P(&places);                         /* attendre une place libre     */
        P(&mutex);
        tampon[ie] = mesure; ie = (ie + 1) % TAILLE;
        printf("prod : depose %d\n", mesure);
        V(&mutex);
        V(&mesures);                        /* signaler une mesure de plus  */
    }
    return NULL;
}

static void *consommateur(void *arg)        /* ex. : tâche d'affichage lente */
{
    (void)arg;
    for (int k = 1; k <= NB_MESURES; k++) {
        P(&mesures);                        /* attendre une mesure          */
        P(&mutex);
        int m = tampon[il]; il = (il + 1) % TAILLE;
        printf("        cons : retire %d\n", m);
        V(&mutex);
        V(&places);                         /* libérer une place            */
        usleep(20000);                      /* traitement lent (20 ms)      */
    }
    return NULL;
}

int main(void)
{
    pthread_t p, c;
    sem_init_(&places, TAILLE);
    sem_init_(&mesures, 0);
    sem_init_(&mutex, 1);
    pthread_create(&c, NULL, consommateur, NULL);
    usleep(5000);                           /* le consommateur se bloque d'abord */
    pthread_create(&p, NULL, producteur, NULL);
    pthread_join(p, NULL);
    pthread_join(c, NULL);
    puts("fin : toutes les mesures ont été traitées, dans l'ordre");
    return 0;
}
```

Exécution :

```
prod : depose 21
prod : depose 22
prod : depose 23
prod : depose 24
        cons : retire 21
prod : depose 25
        cons : retire 22
prod : depose 26
        cons : retire 23
prod : depose 27
        cons : retire 24
prod : depose 28
        cons : retire 25
        cons : retire 26
        cons : retire 27
        cons : retire 28
fin : toutes les mesures ont été traitées, dans l'ordre
```

Le producteur remplit le tampon (4 places) puis se bloque ; il est ensuite cadencé par le consommateur, plus lent. Aucune mesure n'est perdue ni lue deux fois.

3.4 Autres outils de communication et de synchronisation

— **Files de messages** (*message queues*) : tampon FIFO géré par le noyau ; l'émetteur dépose, le récepteur retire, avec blocage optionnel si la file est vide ou pleine. C'est un producteur/consommateur « clé en main » : `xQueueSend`/`xQueueReceive` (FreeRTOS), `rtos_msg_send`/`rtos_msg_read` (CCS).

— **Boîtes aux lettres** (*mailbox*) : file de capacité 1 (la dernière valeur), adaptée à la transmission d'un état (dernière mesure).

— **Drapeaux d'événements** (*event flags*) : bits que les tâches positionnent et attendent (attente d'un ou de plusieurs événements combinés en ET/OU).

— **Tubes** (*pipes*) : flux d'octets entre tâches.

— **Variables partagées protégées** : solution la plus légère pour un état « dernière valeur » ; si la variable est atomique (un octet sur PIC18, un mot de 32 bits sur ARM) et écrite par une seule tâche, aucune synchronisation n'est nécessaire.

3.5 Interblocage et famine

Définition 3.5 — Interblocage

Un **interblocage** (*deadlock*) est une situation où un ensemble de tâches sont bloquées indéfiniment, chacune attendant une ressource détenue par une autre tâche de l'ensemble.

```
Tâche A :  P(LCD) ; P(UART) ; ... ; V(UART) ; V(LCD)
Tâche B :  P(UART) ; P(LCD) ; ... ; V(LCD) ; V(UART)
// A prend LCD, B prend UART, A attend UART, B attend LCD : interblocage
```

Théorème 3.6 — Conditions de Coffman, 1971

Un interblocage ne peut se produire que si les quatre conditions suivantes sont réunies simultanément : (1) exclusion mutuelle ; (2) détention et attente (une tâche garde des ressources en en demandant d'autres) ; (3) pas de réquisition (on ne peut retirer une ressource à une tâche) ; (4) attente circulaire.

Il suffit de casser l'une des conditions. La règle pratique la plus sûre est de casser l'attente circulaire : **toutes les tâches prennent les ressources dans le même ordre global** (par exemple toujours LCD avant UART). Le protocole à plafond de priorité (ci-dessous) empêche aussi les interblocages.

La **famine** (*starvation*) est l'attente indéfinie d'une tâche qui n'est jamais servie (par exemple une tâche peu prioritaire dans un système trop chargé).

3.6 L'inversion de priorité

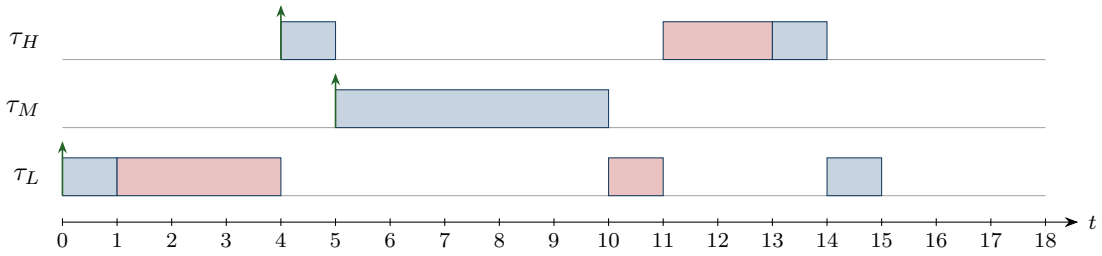
Définition 3.7 — Inversion de priorité

Il y a **inversion de priorité** lorsqu'une tâche de haute priorité τ_H est bloquée par une tâche de basse priorité τ_L qui détient une ressource dont τ_H a besoin, et que des tâches de priorité intermédiaire τ_M, qui n'utilisent pas la ressource, préemptent τ_L : τ_H attend alors indirectement τ_M, sans borne.

Exemple — le robot Mars Pathfinder, 1997

Sur Mars, la sonde redémarrait régulièrement. Une tâche de météorologie peu prioritaire détenait un mutex protégeant le bus de données ; la tâche de gestion du bus, très prioritaire, l'attendait ; des tâches de communication de priorité moyenne préemptaient la tâche météo. Le chien de garde, constatant que la tâche du bus ne s'exécutait plus, réinitialisait le système. Le défaut a été corrigé à distance en activant l'héritage de priorité sur ce mutex de VxWorks.

Exemple : τ_H (priorité 1, réveil à 4, $C = 4$ dont 2 en section critique sur R), τ_M (priorité 2, réveil à 5, $C = 5$, n'utilise pas R), τ_L (priorité 3, réveil à 0, $C = 6$ dont 4 sur R à partir de $t = 1$).

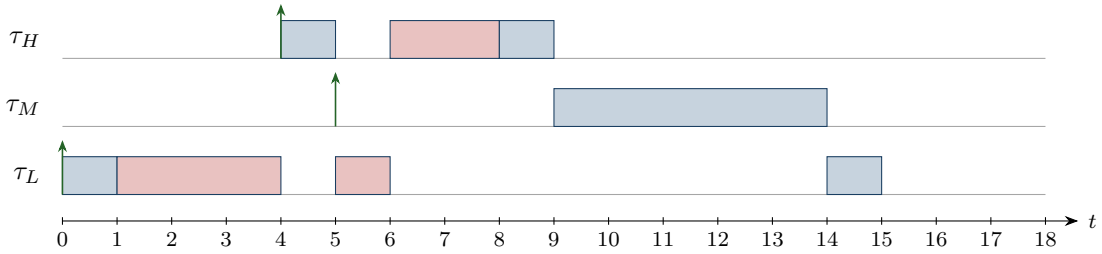


Sans protocole : τ_H, bloquée de $t = 5$ à $t = 11$, attend toute l'exécution de τ_M ($R_H = 10$)

3.6.1 Protocole de l'héritage de priorité (PIP)

Définition 3.8 — Priority Inheritance Protocol

Lorsqu'une tâche τ_L bloque une tâche plus prioritaire τ_H (en détenant une ressource que τ_H demande), τ_L **hérite temporairement de la priorité de** τ_H jusqu'à ce qu'elle libère la ressource. Les tâches intermédiaires ne peuvent plus préempter τ_L.



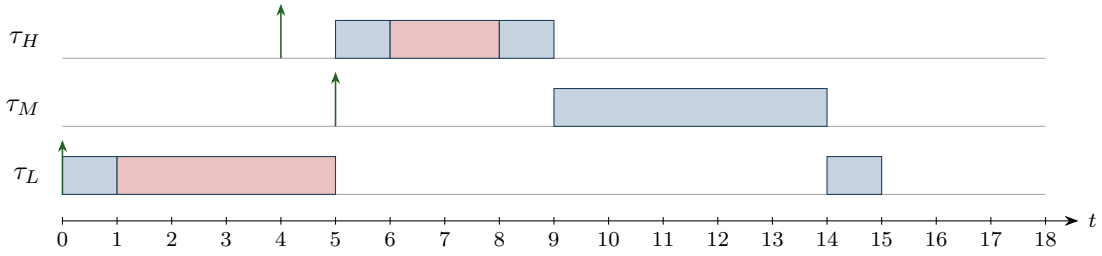
PIP : τ_L hérite de la priorité de τ_H de $t = 5$ à $t = 6$; $R_H = 5$

PIP borne la durée de blocage (au plus une section critique par ressource), mais n'empêche ni les interblocages ni les *blocages en chaîne*.

3.6.2 Protocole à plafond de priorité (PCP, ICPP)

Définition 3.9 — Plafond de priorité

Le **plafond** d'une ressource est la priorité de la tâche la plus prioritaire susceptible de l'utiliser. Dans la variante **à plafond immédiat** (ICPP, *Immediate Ceiling Priority Protocol*, ou *Highest Locker*), une tâche qui prend une ressource voit **immédiatement** sa priorité élevée au plafond de cette ressource, jusqu'à sa libération.



ICPP : τ_L s'exécute au plafond de R dès $t = 1$; τ_H démarre à $t = 5$ sans jamais être bloquée en cours d'exécution ; $R_H = 5$

Théorème 3.10 — Propriétés du plafond de priorité

Avec PCP (ou ICPP), une tâche est bloquée **au plus une fois**, pendant au plus la durée de la plus longue section critique d'une tâche moins prioritaire portant sur une ressource de plafond supérieur ou égal à sa priorité ; **aucun interblocage** n'est possible. Le facteur de blocage B_i s'ajoute à l'analyse du temps de réponse :

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j.$$

Le test de Liu et Layland devient, avec blocage : $\forall i,\ \sum_{k=1}^{i} \frac{C_k}{T_k} + \frac{B_i}{T_i} \leq i\,(2^{1/i} - 1)$ (tâches classées par priorité décroissante). Dans FreeRTOS, les mutex (`xSemaphoreCreateMutex`) implémentent l'héritage de priorité ; POSIX propose `PTHREAD_PRIO_INHERIT` et `PTHREAD_PRIO_PROTECT` (plafond).

À retenir

- Toute donnée partagée modifiée par plusieurs tâches doit être protégée (section critique, exclusion mutuelle).
- Sémaphore : compteur + file, opérations atomiques P/V ; binaire (exclusion), à compteur (N ressources), initialisé à 0 (synchronisation). Mutex : sémaphore binaire avec propriétaire.
- Producteur/consommateur : `places`$= N$, `mesures`$= 0$, `mutex`$= 1$; toujours P(ressource comptée) *avant* P(mutex).
- Interblocage : 4 conditions de Coffman ; prendre les ressources dans un ordre global unique.
- Inversion de priorité : bornée par l'héritage (PIP), supprimée avec interblocages par le plafond (PCP/ICPP) ; on ajoute le blocage B_i dans l'analyse du temps de réponse.

Chapitre 4

Systèmes d'exploitation temps réel et multitâche

Presque tous les systèmes à microcontrôleur exécutent plusieurs activités « à la fois ». Un simple régulateur de température comporte au moins trois activités qui se répètent : lire la température, calculer et appliquer la commande, afficher. Avec un seul processeur, il faut organiser le partage du temps CPU entre ces activités ; comme chacune ne dure que quelques millisecondes, elles *semblent* s'exécuter en parallèle. Ce chapitre présente les solutions, de la plus simple (machine à états) à la plus complète (noyau temps réel), et détaille le RTOS intégré au compilateur CCS C, utilisé en TD et en TP.

4.1 Les machines à états

Une **machine à états** exécute plusieurs activités en séquence ; l'état courant détermine l'activité à réaliser et l'état suivant. En C, on l'implémente avec `switch`/`case` dans une boucle infinie :

```
int8 etat = 1;
for (;;) {
    switch (etat) {
        case 1: lire_temperature();     etat = 2; break;
        case 2: calculer_commande();    etat = 3; break;
        case 3: afficher();             etat = 1; break;
    }
    delay_ms(10);
}
```

L'état suivant peut aussi dépendre de conditions (machine à laver : remplissage → lavage si niveau atteint…). Facile à écrire et totalement déterministe, la machine à états est un ordonnancement **coopératif** primitif : aucune notion de priorité, aucune préemption, et le temps de réponse à un événement dépend de la durée de toute la séquence. Une alarme de surchauffe doit pouvoir s'exécuter *dès* qu'elle devient nécessaire : il faut alors un RTOS.

4.2 Architecture d'un RTOS

Définition 4.1 — RTOS

Un **système d'exploitation temps réel** (RTOS, *Real-Time Operating System*) est un logiciel construit autour d'un **noyau multitâche** qui alloue le processeur aux tâches selon une politique d'ordonnancement *déterministe*, et qui fournit des services de temps, de synchronisation et de communication dont la durée d'exécution est bornée.

Ses composants de base sont :

— l'**ordonnanceur** (*scheduler*), cœur du noyau, qui choisit la tâche à exécuter ;

— les **services** : gestion du temps (tick, délais, alarmes), des interruptions, de la mémoire, des entrées-sorties ;

— les **outils de synchronisation et de communication** : sémaphores, mutex, drapeaux d'événements, boîtes aux lettres, files de messages, tubes (chapitre 3).

4.2.1 Le changement de contexte

Le **contexte** d'une tâche est l'ensemble des informations nécessaires à la reprise de son exécution : compteur ordinal, registres du processeur, pointeur de pile, état. Il est rangé dans un **bloc de contrôle de tâche** (TCB, *Task Control Block*). Lors d'un **changement de contexte**, le noyau sauvegarde le contexte de la tâche qui perd le processeur dans son TCB, puis restaure celui de la tâche élue. Cette opération coûte du temps (quelques microsecondes) : un algorithme qui multiplie les préemptions (LLF) la paie.

4.2.2 Les trois familles d'ordonnanceurs

— **Coopératif** : chaque tâche s'exécute jusqu'à ce qu'elle rende volontairement la main. Simple, sans problème de réentrance, mais une tâche peut monopoliser le CPU et l'ordonnanceur n'a aucun contrôle sur les durées. La machine à états en est la forme la plus simple ; le RTOS de CCS est coopératif.

— **Tourniquet** (*round-robin*) : chaque tâche reçoit une tranche de temps égale ; peu adapté au temps réel, car les besoins des tâches sont très différents.

— **Préemptif à priorités** : la tâche prête la plus prioritaire obtient le CPU, quitte à interrompre la tâche en cours (sur une interruption du tick ou d'un périphérique). C'est l'ordonnanceur temps réel par excellence (FreeRTOS, VxWorks, μC/OS). Les priorités vont typiquement de 0 à 255 ; selon le noyau, 0 est la plus forte (μC/OS) ou la plus faible (FreeRTOS). Les tâches de même priorité se partagent le CPU en tourniquet.

4.2.3 États d'une tâche

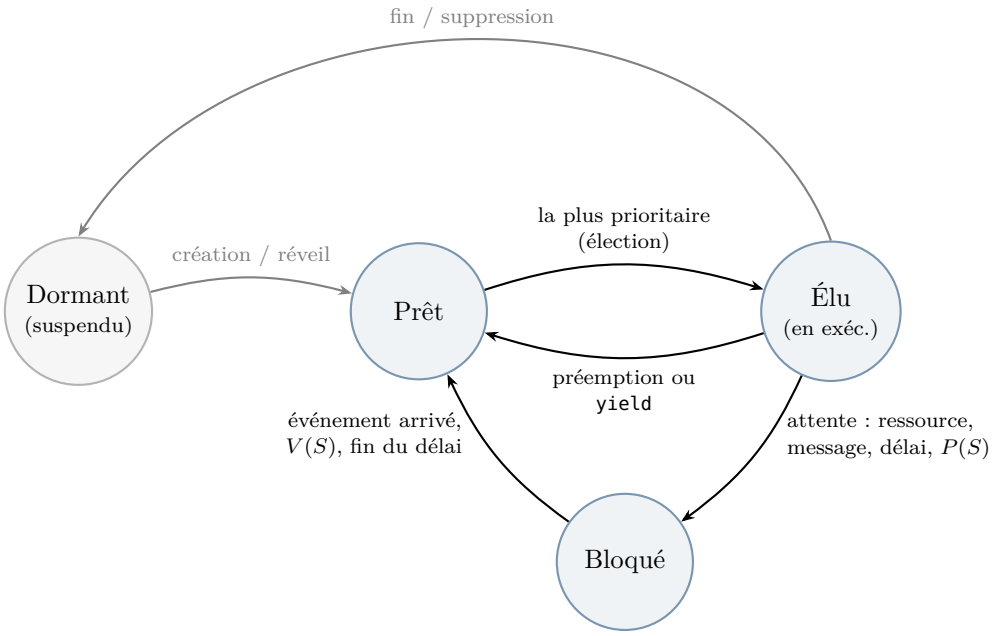


FIGURE 4.1 – Diagramme des états d'une tâche dans un noyau temps réel.

— **Prêt** (*ready*) : la tâche dispose de tout sauf du processeur ; elle attend d'être élue.

— **Élu** (*running*) : la tâche s'exécute ; une seule tâche est élue à la fois sur un monoprocesseur.
— **Bloqué** (*blocked, waiting*) : la tâche attend une ressource, un message, un sémaphore ou l'écoulement d'un délai ; elle ne consomme pas de CPU.
— **Dormant / suspendu** : la tâche existe mais n'est pas candidate (pas encore créée ou activée, terminée, désactivée).

Attention

Seule une tâche **élue** peut se bloquer (c'est en s'exécutant qu'elle demande une ressource) ; une tâche prête ne peut pas passer directement à l'état bloqué. Une tâche débloquée retourne à l'état **prêt** ; elle ne devient élue que si elle est la plus prioritaire. Les enchaînements possibles sont donc par exemple : Élu → Bloqué → Prêt → Élu.

Le noyau offre des opérations sur les tâches : création, suppression, suspension/reprise, changement de priorité, attente d'un délai.

4.3 Un RTOS préemptif : FreeRTOS

FreeRTOS (licence MIT) est le noyau le plus utilisé dans l'embarqué (ARM Cortex-M, ESP32, PIC32, AVR…). Son ordonnanceur est **préemptif à priorités fixes** (la plus grande valeur est la plus prioritaire), avec tourniquet optionnel entre tâches de même priorité. Une tâche périodique s'écrit avec `vTaskDelayUntil`, qui garantit une période sans dérive :

```c
void tache_mesure(void *p)                  // période 100 ms
{
    TickType_t reveil = xTaskGetTickCount();
    for (;;) {
        int16_t t = lire_temperature();
        xQueueSend(file_mesures, &t, 0);         // vers la tâche PID
        vTaskDelayUntil(&reveil, pdMS_TO_TICKS(100));  // Bloqué jusqu'au réveil
    }
}
void tache_pid(void *p)
{
    int16_t t;
    for (;;) {
        xQueueReceive(file_mesures, &t, portMAX_DELAY); // Bloqué sur la file
        xSemaphoreTake(mutex_cons, portMAX_DELAY);      // mutex à héritage de priorité
        u = pid(consigne, t);
        xSemaphoreGive(mutex_cons);
        appliquer_pwm(u);
    }
}
int main(void)
{
    file_mesures = xQueueCreate(8, sizeof(int16_t));
    mutex_cons   = xSemaphoreCreateMutex();
    xTaskCreate(tache_mesure, "mes", 256, NULL, 3, NULL);  // priorité 3
    xTaskCreate(tache_pid,    "pid", 256, NULL, 2, NULL);  // priorité 2
    vTaskStartScheduler();                                 // ne revient pas
}
```

Avec un noyau préemptif, le concepteur fixe les priorités selon RM/DM (chapitre 2) et vérifie l'ordonnançabilité par l'analyse du temps de réponse, en tenant compte des blocages sur mutex (chapitre 3).

4.4 Le compilateur C de CCS et son RTOS

Le compilateur **CCS C** (Custom Computer Services, compilateurs PCM/PCH/PCD, IDE PCW/PCWH) est l'un des plus utilisés pour les PIC10/12/16/18, PIC24 et dsPIC. Sa syntaxe

diffère légèrement du C ANSI (types `int1`, `int8`, `int16`, `int32` ; `int` vaut 8 bits non signés), et il fournit de nombreuses fonctions intégrées (`output_toggle`, `read_adc`, `set_pwm1_duty`, `printf` sur LCD ou UART…). Il intègre un **RTOS coopératif cadencé par le temps**.

> **Remarque — principe du RTOS de CCS** Le RTOS de CCS utilise un *timer* matériel comme horloge. À chaque **cycle mineur**, le **répartiteur** (*dispatcher*) appelé par `rtos_run()` examine les tâches dont la date d'exécution est échue et leur donne le contrôle *à tour de rôle*. Le RTOS **n'utilise pas d'interruptions et n'est pas préemptif** : une tâche garde le processeur jusqu'à ce qu'elle se termine ou rende la main (`rtos_yield`, `rtos_wait`, `rtos_await`) ; l'utilisateur doit donc s'assurer qu'aucune tâche ne s'exécute indéfiniment. Les interruptions restent disponibles pour l'application.

4.4.1 Configuration : la directive `#use rtos`

```
#use rtos(timer=0, minor_cycle=10ms)           // éventuellement : , statistics
```

— `timer` (0 à 4) : timer matériel réservé au RTOS ;

— `minor_cycle` : pas de temps de l'ordonnanceur ; il doit être **supérieur ou égal à la plus longue durée d'exécution** d'une tâche (paramètre `max` de chaque tâche) et les périodes (`rate`) des tâches doivent en être des **multiples**. L'unité suit le nombre : `s`, `ms`, `us`, `ns`. Si on l'omet, le compilateur le calcule ;

— `statistics` : le RTOS mesure les temps d'exécution des tâches (nécessaire pour `rtos_stats` et `rtos_overrun`).

4.4.2 Déclaration d'une tâche : la directive `#task`

Une tâche est une fonction C **sans argument ni valeur de retour**, précédée de :

```
#task(rate=20ms, max=1ms, queue=2)
void ma_tache() { ... }
```

— `rate` : période d'appel de la tâche (multiple de `minor_cycle`) ;

— `max` : temps processeur maximal prévu pour une exécution de la tâche ($\leq$ `minor_cycle`) ; sert au dimensionnement et à la détection des dépassements (`rtos_overrun`) ;

— `queue` (optionnel, défaut 0) : nombre d'octets réservés pour la file de messages de la tâche.

4.4.3 Les fonctions du RTOS de CCS

Fonction	Rôle
`rtos_run()`	démarre le RTOS : appelle les tâches selon leur période ; ne revient qu'après `rtos_terminate()`
`rtos_terminate()`	arrête le RTOS ; le programme reprend après l'appel `rtos_run()`
`rtos_enable(t)`	active la tâche `t` (elle sera appelée à sa période)
`rtos_disable(t)`	désactive la tâche `t` jusqu'au prochain `rtos_enable(t)`
`rtos_yield()`	(dans une tâche) rend la main au répartiteur ; la tâche reprendra *après* cette instruction à sa prochaine activation
`rtos_msg_send(t, b)`	envoie l'octet `b` dans la file de messages de la tâche `t`
`rtos_msg_poll()`	(dans une tâche) nombre d'octets présents dans sa propre file
`rtos_msg_read()`	(dans une tâche) lit le prochain octet de sa file ; appeler `rtos_msg_poll()` avant
`rtos_wait(s)`	opération P : attend que le sémaphore `s` (variable `int8` globale) soit > 0, puis le décrémente ; si `s`$= 0$, la tâche rend la main et réessaiera
`rtos_signal(s)`	opération V : incrémente le sémaphore `s`
`rtos_await(expr)`	la tâche rend la main tant que l'expression est fausse
`rtos_overrun(t)`	vrai si la tâche `t` a dépassé son temps `max` (option `statistics`)
`rtos_stats(t, &st)`	statistiques de `t` : temps total, min, max (en ticks) et durée d'un tick

Attention

`rtos_yield`, `rtos_wait`, `rtos_await`, `rtos_msg_poll` et `rtos_msg_read` ne peuvent être utilisées *qu'à l'intérieur d'une tâche*, jamais dans `main()` ni dans une routine d'interruption. Les sémaphores doivent être initialisés dans `main()` *avant* `rtos_run()`. Une tâche ne doit jamais contenir de boucle infinie ni de longue temporisation (`delay_ms(500)`) : elle bloquerait toutes les autres.

4.4.4 Montage de base du PIC18F45K22

Tous les montages de ce chapitre et du chapitre 5 reposent sur le même montage de base (figure 4.2) : alimentation 5 V avec un condensateur de découplage de 100 nF au plus près de chaque paire de broches VDD/VSS, quartz de 16 MHz et ses deux condensateurs de 22 pF (la PLL interne porte l'horloge à 64 MHz), résistance de tirage de 10 kΩ et bouton de remise à zéro sur $\overline{\text{MCLR}}$. Pour alléger, les schémas suivants ne représentent que les périphériques propres à chaque application ; les numéros de broches sont ceux du boîtier PDIP-40. Sous Proteus, les broches d'alimentation sont reliées implicitement et le quartz peut être omis (la fréquence est fixée dans les propriétés du composant), mais ils sont indispensables sur une carte réelle.

4.4.5 Premier exemple : quatre LED indépendantes

Quatre LED sont reliées à RB0–RB3 d'un PIC18F45K22 (quartz 16 MHz, PLL ×4 : 64 MHz). Quatre tâches les font clignoter à 250 ms, 500 ms, 1 s et 2 s. Chaque tâche ne fait qu'inverser une broche (`output_toggle`) : son exécution dure quelques microsecondes, bien moins que `max = 10ms`.

`leds_rtos.c` *(CCS C — PIC18F45K22)*

```
// leds_rtos.c -- quatre LED clignotant à des rythmes différents (RTOS de CCS)
#include <18F45K22.h>
#fuses NOWDT, NOPBADEN, NOLVP, NOXINST
#use delay(clock=64MHz, crystal=16MHz)          // quartz 16 MHz, PLL x4
#use rtos(timer=0, minor_cycle=10ms)            // Timer0 = horloge du RTOS

#task(rate=250ms, max=10ms)                     // tâche 1 : toutes les 250 ms
void task_B0() { output_toggle(PIN_B0); }

#task(rate=500ms, max=10ms)                     // tâche 2 : toutes les 500 ms
```

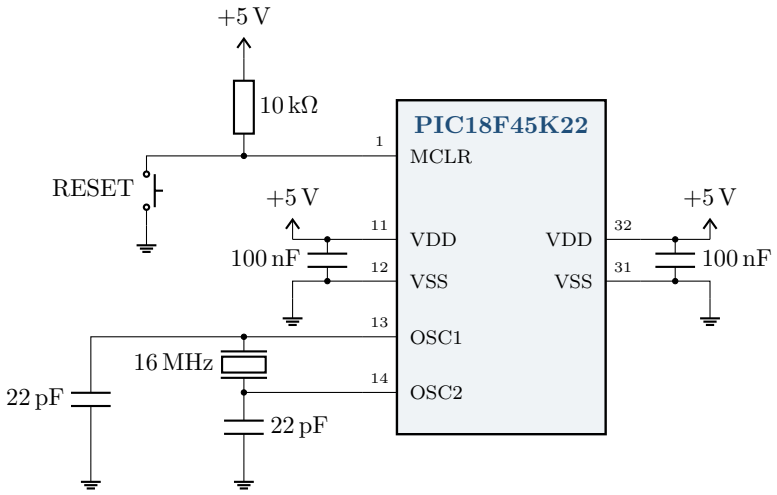


FIGURE 4.2 – Montage de base du PIC18F45K22 : alimentation, découplage, quartz 16 MHz et circuit de remise à zéro.

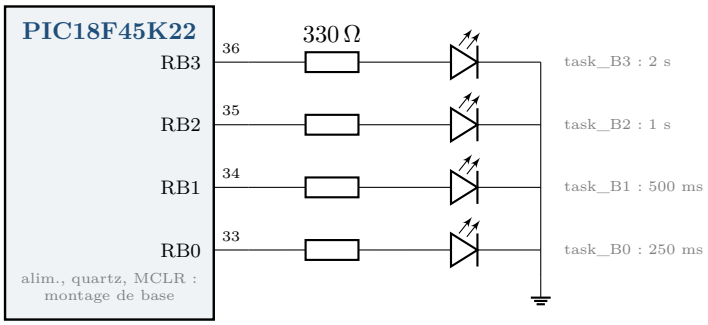


FIGURE 4.3 – Schéma du montage à quatre LED (`leds_rtos.c`) : chaque LED est reliée à la masse à travers une résistance de 330 Ω ($I \approx (5-2)/330 \approx 9$ mA).

```
11 void task_B1() { output_toggle(PIN_B1); }
12
13 #task(rate=1s, max=10ms)                        // tâche 3 : toutes les secondes
14 void task_B2() { output_toggle(PIN_B2); }
15
16 #task(rate=2s, max=10ms)                        // tâche 4 : toutes les 2 s
17 void task_B3() { output_toggle(PIN_B3); }
18
19 void main()
20 {
21     set_tris_b(0x00);                          // PORTB en sortie
22     output_b(0x00);
23     rtos_run();                                // lance l'ordonnanceur (ne revient pas)
24 }
```

Chaque LED clignote indépendamment des autres : la LED RB0 a une période de 500 ms (allumée 250 ms, éteinte 250 ms). Toutes les périodes (250 ms, 500 ms, 1 s, 2 s) sont des multiples de `minor_cycle = 10ms`.

4.4.6 Communication par messages

La tâche `Generator` compte en permanence ; lorsqu'on appuie sur le bouton RD0, elle envoie la valeur courante à la tâche `Display`, qui l'affiche sur les LED du PORTB (générateur de nombres pseudo-aléatoires). `rtos_await` évite d'envoyer un message à chaque milliseconde tant que le bouton reste enfoncé.

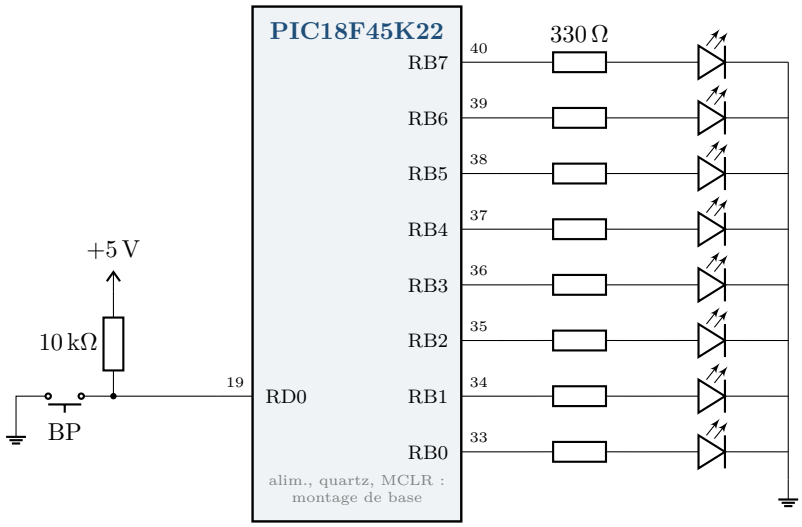


FIGURE 4.4 – Schéma du générateur de nombres (`messages_rtos.c`) : huit LED sur le PORTB, bouton-poussoir sur RD0 avec résistance de tirage (niveau bas à l'appui).

`messages_rtos.c` *(CCS C — PIC18F45K22)*

```c
// messages_rtos.c -- communication par file de messages entre deux tâches
#include <18F45K22.h>
#fuses NOWDT, NOPBADEN, NOLVP, NOXINST
#use delay(clock=64MHz, crystal=16MHz)
#use rtos(timer=0, minor_cycle=1ms)

int8 count;                                  // compteur « libre »

#task(rate=1ms, max=1ms)
void Generator()                             // compte en permanence
{
    count++;
    if (input(PIN_D0) == 0) {                // bouton appuyé (actif bas)
        rtos_msg_send(Display, count);       // dépose un octet dans la file
        rtos_await(input(PIN_D0) == 1);      // attend le relâchement
    }
}

#task(rate=10ms, max=1ms, queue=2)           // file de 2 octets
void Display()
{
    if (rtos_msg_poll() > 0)                 // un message est-il arrivé ?
        output_b(rtos_msg_read());           // l'afficher sur les 8 LED
}

void main()
{
    set_tris_b(0x00);
    rtos_run();
}
```

4.4.7 Exclusion mutuelle par sémaphore

Deux tâches écrivent sur le même afficheur LCD. Avec un RTOS coopératif, une tâche n'est jamais interrompue par une autre tâche *tant qu'elle ne rend pas la main* ; le sémaphore devient indispensable dès qu'une section critique contient un `rtos_yield()` ou un `rtos_await()` (affichage découpé en plusieurs morceaux), ou lorsque la ressource est aussi utilisée sous interruption (dans ce dernier cas, il faut masquer l'interruption). On prend donc l'habitude de protéger systématiquement les ressources partagées :

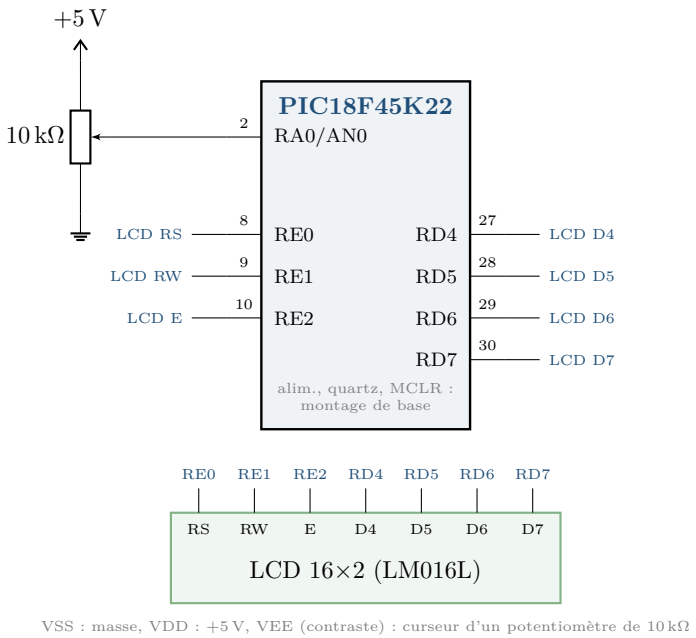


FIGURE 4.5 – Schéma du montage LCD + potentiomètre (`semaphore_rtos.c`) : afficheur en mode 4 bits (RS, RW, E sur RE0–RE2, D4–D7 sur RD4–RD7, reliés par étiquettes) et potentiomètre de 10 kΩ sur AN0.

`semaphore_rtos.c` *(CCS C — PIC18F45K22)*

```c
// semaphore_rtos.c -- deux tâches partagent l'afficheur LCD (ressource critique)
#include <18F45K22.h>
#device ADC=10
#fuses NOWDT, NOPBADEN, NOLVP, NOXINST
#use delay(clock=64MHz, crystal=16MHz)
#use rtos(timer=0, minor_cycle=5ms)

#define LCD_ENABLE_PIN PIN_E2                  // brochage du LCD (driver lcd.c)
#define LCD_RS_PIN     PIN_E0
#define LCD_RW_PIN     PIN_E1
#define LCD_DATA4      PIN_D4
#define LCD_DATA5      PIN_D5
#define LCD_DATA6      PIN_D6
#define LCD_DATA7      PIN_D7
#include <lcd.c>

int8 sem_lcd;                                  // sémaphore protégeant le LCD
int16 secondes = 0;

#task(rate=1s, max=5ms)
void Horloge()
{
    secondes++;
    rtos_wait(sem_lcd);                        // P(sem_lcd) : rend la main si occupé
    lcd_gotoxy(1, 1);
    printf(lcd_putc, "t = %5lu s   ", secondes);
    rtos_signal(sem_lcd);                      // V(sem_lcd)
}

#task(rate=200ms, max=5ms)
void Mesure()
{
    int16 n;
    set_adc_channel(0);
    delay_us(10);
```

```
    n = read_adc();
    rtos_wait(sem_lcd);
    lcd_gotoxy(1, 2);
    printf(lcd_putc, "AN0 = %4lu    ", n);
    rtos_signal(sem_lcd);
}

void main()
{
    setup_adc_ports(sAN0);
    setup_adc(ADC_CLOCK_DIV_64);
    lcd_init();
    sem_lcd = 1;                                // une seule tâche à la fois
    rtos_run();
}
```

4.4.8 Mesure des temps d'exécution

L'option `statistics` permet de vérifier, sur cible ou en simulation, que les durées réelles respectent les valeurs `max` déclarées — c'est la mesure du WCET indispensable à l'analyse d'ordonnançabilité :

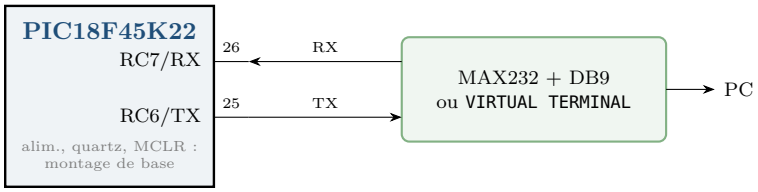


FIGURE 4.6 – Liaison série vers le PC (`stats_rtos.c`) : TX (RC6) et RX (RC7) de l'UART1, adaptateur de niveaux MAX232 et connecteur DB9 sur une carte réelle, `VIRTUAL TERMINAL` sous Proteus.

stats_rtos.c *(CCS C — PIC18F45K22)*

```
// stats_rtos.c -- surveillance des dépassements et statistiques d'exécution
#include <18F45K22.h>
#fuses NOWDT, NOPBADEN, NOLVP, NOXINST
#use delay(clock=64MHz, crystal=16MHz)
#use rs232(baud=115200, UART1, bits=8, parity=N) // 87 µs par caractère
#use rtos(timer=0, minor_cycle=5ms, statistics)  // statistics : rtos_stats / rtos_overrun

struct rtos_stats st;                           // structure définie par le compilateur

#task(rate=10ms, max=1ms)
void Calcul()
{
    float x = 1.0;
    int8 i;
    for (i = 0; i < 20; i++) x = x * 1.01;      // traitement à mesurer
}

#task(rate=1s, max=5ms)                         // trame courte : < 25 caractères
void Superviseur()
{
    if (rtos_overrun(Calcul))                   // Calcul a-t-elle dépassé max ?
        printf("DEPASSEMENT\r\n");
    rtos_stats(Calcul, &st);
    printf("max=%lu t x%lu00ns\r\n", st.task_max_ticks, st.hns_per_tick);
}

void main()
{
    rtos_run();
```

```
30 }
```

4.4.9 Ordonnançabilité avec le RTOS de CCS

Le RTOS de CCS est **non préemptif** : à chaque cycle mineur, les tâches échues sont exécutées les unes après les autres. Pour que les périodes soient respectées, il faut :

1. que chaque tâche dure moins que `max` et que `max` ≤ `minor_cycle` ;
2. que la somme des durées des tâches susceptibles d'être échues *au même cycle mineur* reste inférieure au cycle mineur (sinon les cycles suivants sont décalés : la gigue augmente) ;
3. que la charge totale $U = \sum C_i/T_i$ reste nettement inférieure à 1.

Le pire retard d'une tâche est celui du modèle non préemptif (chapitre 2) : une tâche urgente peut attendre la fin de toutes les tâches échues avant elle. Les traitements longs (affichage LCD complet, calcul en flottant, envoi d'une longue chaîne sur l'UART) doivent être découpés avec `rtos_yield()`.

4.5 L'environnement de développement : CCS + Proteus

La mise au point se fait en simulation avant le passage sur carte :

1. **Projet CCS** (PCWH) : *Project* → *PIC Wizard* ou *Create*, choisir le PIC18F45K22, l'horloge (quartz 16 MHz, PLL : 64 MHz) ; écrire le programme ; compiler (*Build*) : le compilateur produit le fichier `.hex` (programmation) et le fichier `.cof` (débogage symbolique sous Proteus).
2. **Schéma Proteus** (ISIS/*Schematic Capture*) : placer le `PIC18F45K22`, les LED et résistances, les boutons, l'afficheur `LM016L` (LCD 16×2), le `VIRTUAL TERMINAL` relié à TX/RX (RC6/RC7) — ou `COMPIM` pour relier un vrai port COM —, le capteur `LM35`, le modèle `OVEN` (four) pour le projet.
3. **Propriétés du PIC** (double clic) : *Program File* = fichier `.cof` (ou `.hex`), *Processor Clock Frequency* = 64 MHz (fréquence *effective* après PLL, cohérente avec `#use delay`).
4. **Simulation** : lancer, observer LED, LCD, terminal ; utiliser l'oscilloscope et l'analyseur logique virtuels pour mesurer les périodes réelles des tâches (une broche basculée en entrée/sortie de tâche donne sa durée d'exécution) ; en pas à pas, suivre les variables grâce au fichier `.cof`.

> **Remarque** Sous Proteus, la simulation peut être plus lente que le temps réel lorsque le schéma est chargé : l'indicateur de charge CPU en bas de la fenêtre le signale. Les durées mesurées avec les instruments virtuels restent celles du temps *simulé*, seules pertinentes.

4.6 Études de cas

Les deux études de cas suivantes sont traitées en TD (fiche 6) puis réalisées sous Proteus.

Étude de cas n° 1 : générateur de nombres aléatoires

Huit LED sur le PORTB d'un PIC18F45K22, un bouton-poussoir sur RD0, une LED « de vie » sur RD7 et un terminal série sur l'UART1. Quatre tâches : `Live` (clignotement de RD7 toutes les 200 ms), `Generator` (comptage 0–255 en continu, envoi de la valeur courante aux tâches `Display` et `Terminal` par message lors d'un appui sur le bouton *ou* de la réception du caractère `'N'`), `Display` (affichage de l'octet reçu sur les LED) et `Terminal` (transmission à `Generator` des caractères reçus de l'UART, et envoi au PC des nombres reçus de `Generator`).

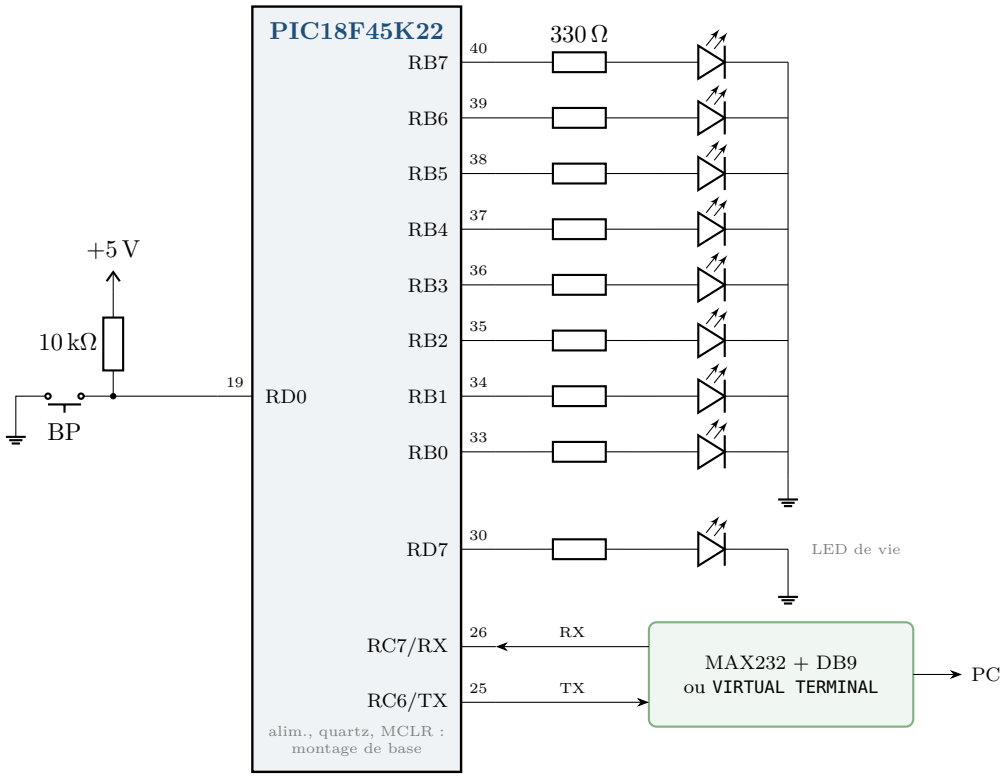


FIGURE 4.7 – Schéma de l'étude de cas n° 1 (générateur de nombres aléatoires).

Étude de cas n° 2 : voltmètre/thermomètre avec liaison RS232

La tension d'un capteur LM35 (10 mV/°C) est lue sur AN0, convertie et envoyée au PC. Trois tâches : `Live` (200 ms), `Get_voltage` (lecture et mise en forme toutes les 2 s) et `To_RS232` (envoi de la tension et de la température toutes les secondes). L'accès aux données partagées entre `Get_voltage` et `To_RS232` est synchronisé par un sémaphore.

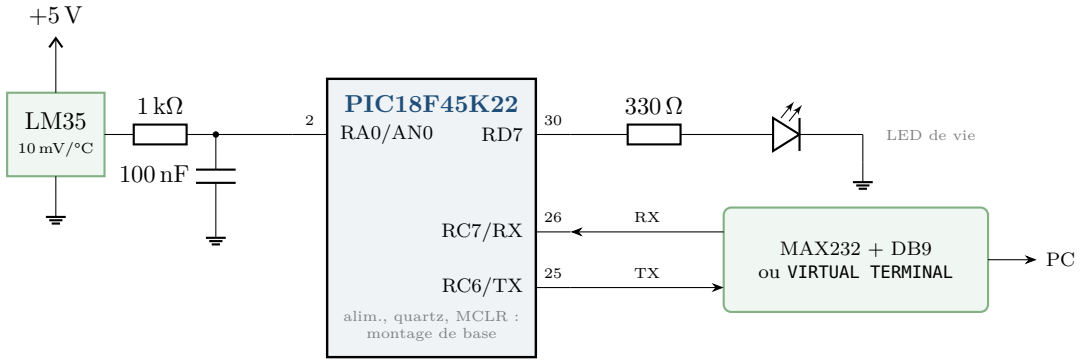


FIGURE 4.8 – Schéma de l'étude de cas n° 2 (voltmètre/thermomètre) : LM35 suivi d'un filtre RC (1 kΩ, 100 nF) sur AN0, LED de vie sur RD7, liaison série.

À retenir

- Un RTOS = ordonnanceur + services + outils de synchronisation/communication, à durées bornées.
- États d'une tâche : prêt, élu, bloqué (+ dormant) ; seule une tâche élue peut se bloquer.
- FreeRTOS : préemptif à priorités fixes ; CCS : coopératif, cadencé par `minor_cycle`, tâches déclarées par `#task(rate, max, queue)`.

— Avec CCS : `max` $\leq$ `minor_cycle`, `rate` multiple de `minor_cycle`, aucune attente active dans une tâche, sémaphores initialisés avant `rtos_run()`.

Chapitre 5

Conception d'un système temps réel : asservissement de température

Ce chapitre applique l'ensemble du cours à un cas complet : l'asservissement de température d'un four à micro-ondes (simulé sous Proteus par le modèle `OVEN`) par un PIC18F45K22. La démarche — cahier des charges, architecture matérielle, modélisation du procédé, synthèse de la loi de commande, découpage en tâches, analyse temporelle, codage, validation — est celle de tout projet temps réel.

5.1 Démarche de conception d'un STR

1. **Spécification** : fonctions, interfaces, contraintes temporelles (périodes, échéances, latences), modes de fonctionnement et de défaillance.
2. **Conception architecturale** : choix du matériel (microcontrôleur, capteurs, actionneurs, IHM), découpage du logiciel en tâches, choix des mécanismes de communication et de synchronisation.
3. **Conception détaillée** : algorithmes de chaque tâche (ici le PID numérique), estimation des durées d'exécution.
4. **Validation temporelle** : choix des périodes, analyse d'ordonnançabilité (chapitre 2), blocages (chapitre 3).
5. **Codage** avec le RTOS (chapitre 4), **tests** en simulation puis sur cible : mesure des WCET, vérification des échéances et des performances de la régulation.

5.2 Cahier des charges

— Réguler la température du four à une consigne w réglable de 30 à 250 °C, par pas de 5 °C (boutons PLUS sur RB0 et MOINS sur RB1), ou par commande `S<valeur>` reçue du terminal série (9600 bauds).

— Performances : erreur statique nulle, dépassement ≤ 10 %, temps de réponse à 5 % inférieur à 4 min pour un échelon de 25 à 150 °C.

— Afficher sur le LCD 16×2 : ligne 1 la consigne, ligne 2 la température mesurée et la commande (%).

— Envoyer chaque seconde au terminal une trame `W=... T=... U=...`.

— **Sécurité** : si la température dépasse $T_{\max} = 260$ °C, couper la chauffe, allumer la LED d'alarme (RB6) et suspendre la régulation jusqu'au réarmement par la commande `R` (possible seulement si $T < T_{\max} - 20$ °C). Délai de réaction ≤ 200 ms.

— Signaler le bon fonctionnement par une LED de vie (RB7, période 400 ms).

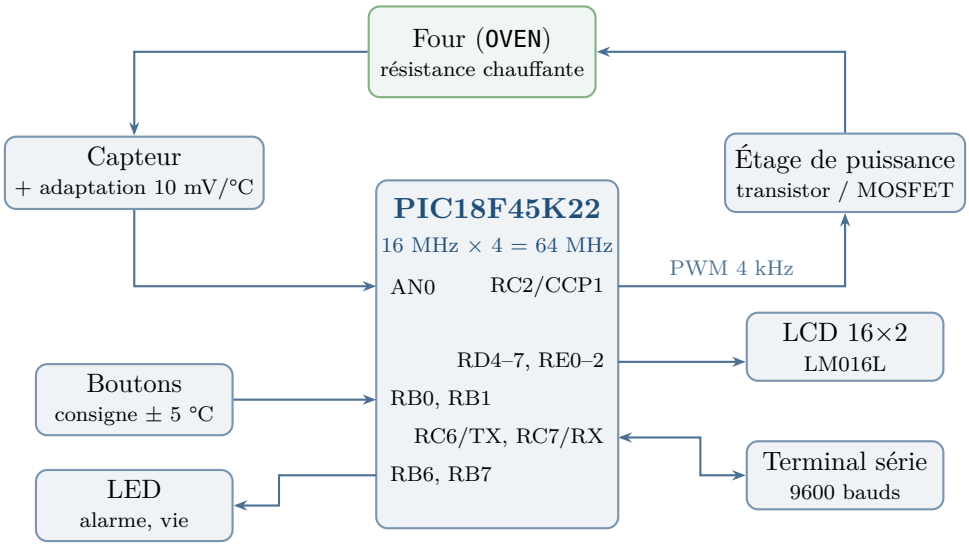


FIGURE 5.1 – Architecture matérielle du système de régulation.

5.3 Architecture matérielle

5.3.1 Chaîne de mesure

Le modèle `OVEN` de Proteus possède une entrée de chauffe et une sortie `T` image de la température (les paramètres du modèle — température ambiante, résistance thermique, constantes de temps du four et de l'élément chauffant, coefficient de température, puissance de chauffe — se règlent dans ses propriétés ; vérifier l'échelle de la sortie `T` dans la documentation de la version utilisée). On adapte cette sortie (pont diviseur, amplificateur) pour obtenir **10 mV/°C**, comme un capteur LM35, ce qui donne 0–5 V pour 0–500 °C. Avec le CAN 10 bits ($V_{\text{ref}} = 5$ V) :

$$\theta\,[^\circ\text{C}] = N \cdot \frac{5000\ \text{mV}}{1023} \cdot \frac{1\ ^\circ\text{C}}{10\ \text{mV}} = N \cdot \frac{500}{1023} \approx 0{,}489\,N, \qquad \text{résolution} \ \approx 0{,}49\ ^\circ\text{C}.$$

La tâche de mesure moyenne 4 conversions pour réduire le bruit.

5.3.2 Chaîne d'action

La puissance de chauffe est modulée par le rapport cyclique d'un signal PWM produit par le module CCP1 (broche RC2), associé au Timer2 :

$$f_{\text{PWM}} = \frac{F_{\text{osc}}}{4 \cdot \text{prédiv} \cdot (PR2 + 1)} = \frac{64\ \text{MHz}}{4 \times 16 \times 250} = 4\ \text{kHz}, \qquad \text{résolution} = 4\,(PR2 + 1) = 1000\ \text{pas}.$$

Une commande $u \in [0, 100]$ % s'applique par `set_pwm1_duty((int16)(u * 10))`. Le four filtre naturellement la PWM (constantes de temps de plusieurs secondes) : seule la puissance moyenne compte. Sur un four réel alimenté en 230 V, on commanderait un triac par train d'ondes avec une période de modulation de l'ordre de la seconde.

5.3.3 Schéma électronique

La figure 5.2 détaille le schéma (montage de base de la figure 4.2 non repris) :

— **mesure** : pont diviseur 99 kΩ / 1 kΩ (rapport 1/100) entre la sortie `T` du four et AN0, à adapter à l'échelle réelle de la sortie mesurée en TP ;

— **action** : la sortie PWM (RC2/CCP1) commande, à travers une résistance de grille de 100 Ω, un MOSFET à seuil logique (saturé dès $V_{GS} = 5$ V) placé côté bas de l'entrée de chauffe ;

— **IHM** : LCD en mode 4 bits, boutons PLUS (RB0) et MOINS (RB1) avec résistances de tirage de 10 kΩ, LED de vie (RB7) et d'alarme (RB6), liaison série (RC6/RC7).

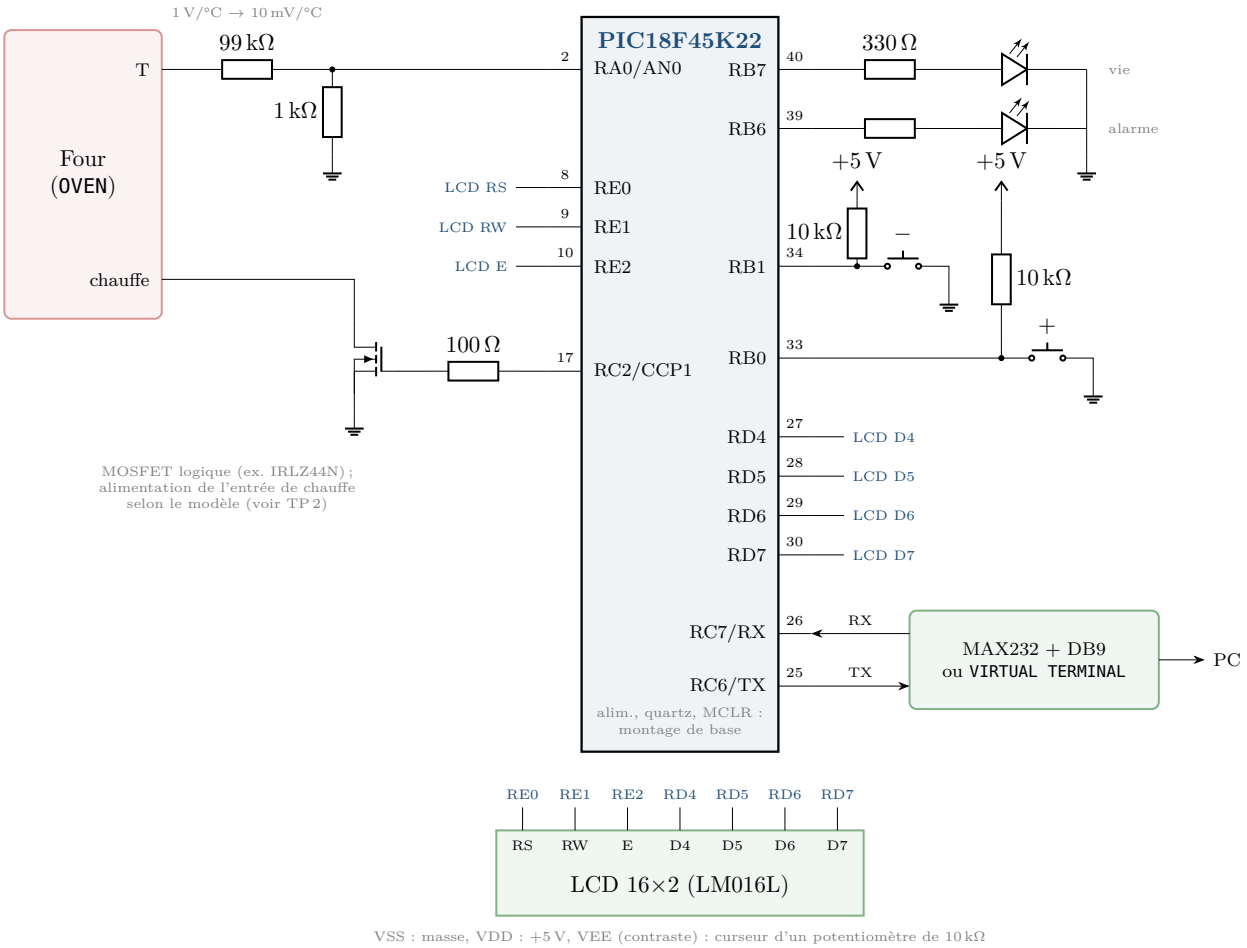


FIGURE 5.2 – Schéma électronique du régulateur de température (`four.c`). Les liaisons avec le LCD sont représentées par des étiquettes.

5.4 Modélisation et identification du procédé

Un four est bien représenté par un modèle du **premier ordre avec retard** (modèle de Strejc-Broïda) :

$$G(s) = \frac{\Delta\theta(s)}{U(s)} = \frac{K\, e^{-\theta_r s}}{1 + \tau s},$$

où K est le gain statique (°C/%), τ la constante de temps et θ_r le retard pur. On l'identifie par un **essai indiciel en boucle ouverte** : partant de l'équilibre (température ambiante, $u = 0$), on applique un échelon Δu et on enregistre $\theta(t)$ (sous Proteus : sonde de tension sur l'entrée du CAN + graphe, ou trame série toutes les secondes).

Définition 5.1 — Méthode de Broïda

On relève les instants t_{28} et t_{40} où la réponse atteint 28 % et 40 % de sa variation finale $\Delta\theta_\infty$. Alors :

$$K = \frac{\Delta\theta_\infty}{\Delta u}, \qquad \tau = 5{,}5\,(t_{40} - t_{28}), \qquad \theta_r = 2{,}8\,t_{28} - 1{,}8\,t_{40}.$$

Exemple — identification

Sur la figure 5.3, $\Delta u = 50$ %, $\Delta\theta_\infty = 175 - 25 = 150$ °C, d'où $K = 3$ °C/%. Les niveaux 28 % et 40 % correspondent à $25 + 0{,}28 \times 150 = 67$ °C et $25 + 0{,}40 \times 150 = 85$ °C, atteints à $t_{28} \approx 59{,}4$ s

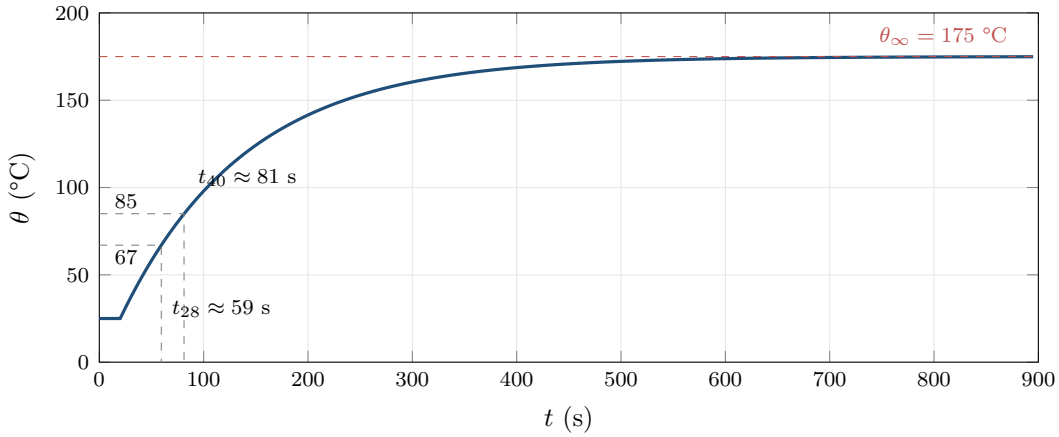


FIGURE 5.3 – Essai indiciel en boucle ouverte : u passe de 0 à 50 % à $t = 0$, $\theta_0 = 25$ °C.

et $t_{40} \approx 81{,}2$ s. Donc $\tau = 5{,}5 \times 21{,}8 \approx 120$ s et $\theta_r = 2{,}8 \times 59{,}4 - 1{,}8 \times 81{,}2 \approx 20$ s.

5.5 Synthèse du régulateur PID

5.5.1 Réglage

Le correcteur PID « mixte » s'écrit $u(t) = K_p\left(e(t) + \frac{1}{T_i}\int_0^t e\,\mathrm{d}t + T_d\frac{\mathrm{d}e}{\mathrm{d}t}\right)$, $e = w - \theta$. Pour un modèle du premier ordre avec retard, on dispose de règles empiriques :

Méthode (PID)	K_p	T_i	T_d
Ziegler-Nichols (boucle ouverte)	$1{,}2\,\tau/(K\theta_r)$	$2\,\theta_r$	$0{,}5\,\theta_r$
Chien-Hrones-Reswick, consigne, 0 % de dépassement	$0{,}6\,\tau/(K\theta_r)$	τ	$0{,}5\,\theta_r$
Application ($K = 3$, $\tau = 120$ s, $\theta_r = 20$ s) : ZN	2,4 %/°C	40 s	10 s
CHR	1,2 %/°C	120 s	10 s

5.5.2 Choix de la période d'échantillonnage

La période T_e de la tâche PID doit être petite devant la dynamique du procédé ; règles usuelles : $T_e \le \tau/10$ et $T_e \le \theta_r/4$, soit ici $T_e \le 5$ s. On choisit $T_e = 0{,}5$ s : la régulation numérique se comporte alors pratiquement comme le régulateur continu, et la charge processeur reste très faible. Une période trop grande dégrade la réponse (et ajoute un retard équivalent $\approx T_e/2$) ; une période trop petite augmente la charge et le bruit sur l'action dérivée.

5.5.3 Discrétisation et mise en œuvre

Avec l'approximation d'Euler, la forme **position** du PID, à dérivée sur la mesure (pas de « coup de pied » au changement de consigne) et filtrée ($N \approx 10$), s'écrit :

$$
\begin{aligned}
P_k &= K_p\,e_k, \qquad e_k = w_k - \theta_k,\\
I_k &= I_{k-1} + \frac{K_p T_e}{T_i}\,e_k,\\
D_k &= \frac{T_d}{T_d + N T_e}\,D_{k-1} - \frac{K_p T_d N}{T_d + N T_e}\,(\theta_k - \theta_{k-1}),\\
u_k &= \mathrm{sat}_{[0,\,100]}\,(P_k + I_k + D_k)\,.
\end{aligned}
$$

La forme **incrémentale** (vitesse) calcule seulement la variation de la commande : $u_k = u_{k-1} + q_0 e_k + q_1 e_{k-1} + q_2 e_{k-2}$ avec $q_0 = K_p(1 + T_e/T_i + T_d/T_e)$, $q_1 = -K_p(1 + 2T_d/T_e)$, $q_2 = K_p T_d/T_e$; elle facilite le passage manuel/automatique sans à-coup (*bumpless*).

Attention — emballement de l'intégrale (*windup*)

Au démarrage, l'erreur est grande et la commande sature à 100 %. Si l'intégrale continue de croître pendant la saturation, elle atteint une valeur énorme qui ne se « vide » qu'après un long dépassement de la consigne. On protège le régulateur par un **anti-windup** : intégration conditionnelle (on n'intègre pas quand la sortie est saturée et que l'erreur la pousse dans le même sens), ou limitation de l'intégrale.

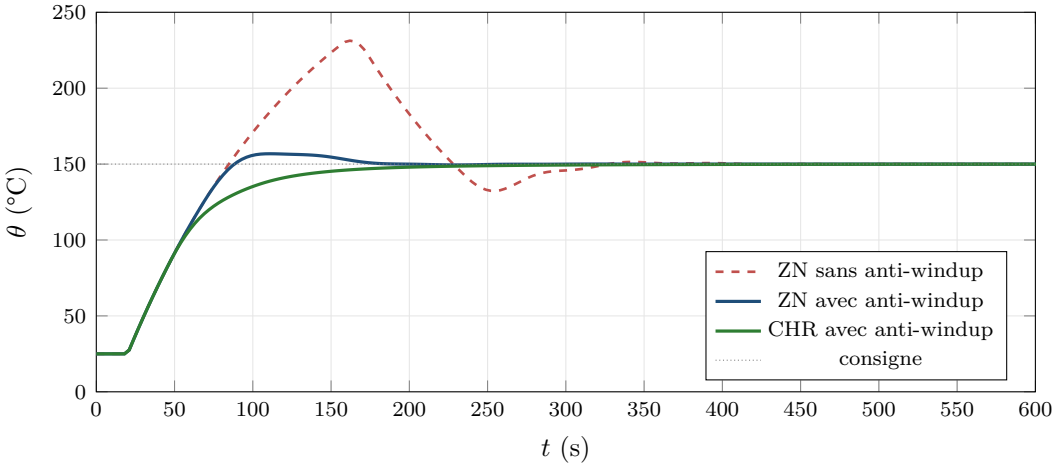


FIGURE 5.4 – Réponses simulées à un échelon de consigne 25 → 150 °C ($T_e = 0{,}5$ s, $u \in [0, 100]$ %).

Réglage	Dépassement	$t_{90\,\%}$	$t_{5\,\%}$
ZN sans anti-windup	65 % (231 °C)	—	284 s
ZN avec anti-windup	5,5 % (157 °C)	77 s	132 s
CHR avec anti-windup	0 %	107 s	137 s

Les deux réglages avec anti-windup satisfont le cahier des charges ; on retient le réglage CHR (pas de dépassement, intéressant pour ne pas brûler la charge). Sans anti-windup, la consigne est dépassée de 81 °C et le temps de réponse est doublé.

5.6 Conception multitâche

5.6.1 Découpage en tâches

On associe une tâche à chaque activité ayant son propre rythme :

Tâche	Rôle	T_i	C_i estimé	C_i/T_i
`Live`	bascule la LED de vie RB7	200 ms	10 µs	0,005 %
`Clavier`	lit RB0/RB1, détecte les fronts, modifie la consigne	50 ms	40 µs	0,08 %
`Mesure`	4 conversions, moyenne, conversion en °C	100 ms	200 µs	0,2 %
`PID`	loi de commande, mise à jour de la PWM	500 ms	900 µs	0,18 %
`Securite`	surveille $T > T_{\max}$, coupe la chauffe	100 ms	20 µs	0,02 %
`Afficheur`	écrit *une* ligne du LCD	250 ms	2,5 ms	1 %
`Terminal`	interprète les commandes, émet *un* caractère	10 ms	60 µs	0,6 %
			Charge totale U	$\approx 2{,}1$ %

Les durées sont des estimations à confirmer par `rtos_stats` : le calcul en flottant du PID (multiplications et additions de 32 bits émulées par logiciel sur PIC18) et l'écriture sur le LCD (environ 40 µs par caractère plus les temporisations du pilote) dominent.

5.6.2 Choix du cycle mineur et vérification temporelle

— Les périodes (10, 50, 100, 200, 250, 500 ms) sont toutes multiples de 10 ms : `minor_cycle = 10ms` ; l'hyperpériode vaut ppcm = 1000 ms.

— La plus longue tâche (`Afficheur`, 2,5 ms) est bien inférieure au cycle mineur. À $t = 0$ (puis toutes les secondes), *toutes* les tâches sont échues au même cycle mineur : la somme $\sum C_i = 10 + 40 + 200 + 900 + 20 + 2500 + 60 = 3730$ µs $= 3{,}73$ ms < 10 ms. Chaque tâche est donc exécutée dans le cycle mineur où elle est réveillée : retard (gigue) maximal 3,7 ms, négligeable devant les périodes (0,75 % de T_e pour le PID).

— Délai de réaction de la sécurité : au pire une période de `Securite` (100 ms) + une période de `Mesure` (100 ms, donnée la plus ancienne) + le retard d'exécution (< 10 ms), soit < 210 ms. Pour garantir ≤ 200 ms, on peut lire directement le CAN dans `Securite` ou ramener sa période à 50 ms.

Attention — les pièges classiques

— `printf` d'une trame de 23 caractères à 9600 bauds bloque le processeur environ 24 ms (1,04 ms par caractère) : c'est plus que le cycle mineur ! On émet donc **un caractère par activation** de la tâche `Terminal` (toutes les 10 ms), ce qui ne bloque jamais.

— La réception série est confiée à une **interruption** (`#int_rda`) : à 9600 bauds un caractère peut arriver toutes les 1,04 ms, une scrutation à 10 ms en perdrait. L'interruption ne fait que ranger l'octet dans un tampon circulaire, lu par la tâche.

— Rafraîchir les deux lignes du LCD en une fois doublerait la durée de la tâche ; on écrit une ligne par activation.

5.6.3 Communication et synchronisation

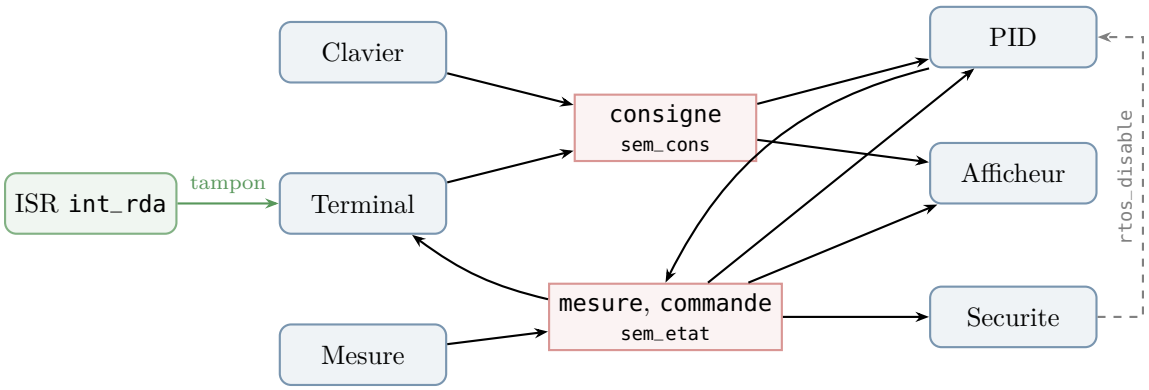


FIGURE 5.5 – Flots de données entre tâches et ressources partagées (PID lit la mesure et y écrit la commande).

— `consigne` est écrite par deux tâches (`Clavier`, `Terminal`) et lue par `PID` et `Afficheur` : sémaphore `sem_cons`.

— `mesure` et `commande` forment l'état du procédé, lu par quatre tâches : sémaphore `sem_etat`. Avec le RTOS coopératif de CCS, une tâche n'est jamais préemptée par une autre, et une variable `float` (4 octets) ne peut donc pas être lue « à moitié écrite » ; les sémaphores restent néanmoins une bonne pratique : ils rendent le code correct s'il est porté sur un noyau préemptif et si une section critique contient un jour un `rtos_yield()`.

— Le tampon de réception est partagé entre l'ISR et `Terminal` : l'ISR est seule à écrire `rx_w`, la tâche seule à écrire `rx_r` ; ces index d'un octet sont lus et écrits atomiquement sur PIC18, il n'y a donc pas besoin de masquer l'interruption.

5.7 Programme complet

four.c *(CCS C — PIC18F45K22)*

```c
// four.c -- asservissement de température d'un four (PIC18F45K22, RTOS de CCS)
// Tâches : Live, Clavier, Mesure, PID, Securite, Afficheur, Terminal
#include <18F45K22.h>
#device ADC=10
#fuses NOWDT, NOPBADEN, NOLVP, NOXINST, NOPROTECT
#use delay(clock=64MHz, crystal=16MHz)
#use rs232(baud=9600, UART1, bits=8, parity=N, errors)
#use rtos(timer=0, minor_cycle=10ms, statistics)
#include <stdlib.h>
#include <string.h>

#define LCD_ENABLE_PIN PIN_E2
#define LCD_RS_PIN     PIN_E0
#define LCD_RW_PIN     PIN_E1
#define LCD_DATA4      PIN_D4
#define LCD_DATA5      PIN_D5
#define LCD_DATA6      PIN_D6
#define LCD_DATA7      PIN_D7
#include <lcd.c>

#define LED_VIE    PIN_B7
#define LED_ALARME PIN_B6
#define BP_PLUS    PIN_B0                    // boutons actifs à l'état bas
#define BP_MOINS   PIN_B1
#define T_MAX      260.0                     // seuil de sécurité (°C)

// ---- régulateur PID (identification : K = 3 °C/%, tau = 120 s, theta = 20 s)
#define TE  0.5                              // période de la tâche PID (s)
#define KP  1.2                              // réglage Chien-Hrones-Reswick 0 %
#define TI  120.0
#define TD  10.0
#define NF  10.0                             // filtre de l'action dérivée

// ---- données partagées et sémaphores
float consigne = 150.0;          int8 sem_cons;   // écrite par Clavier et Terminal
float mesure = 25.0;                              // écrite par Mesure
float commande = 0.0;            int8 sem_etat;   // protège mesure + commande
int1  alarme = 0;

// ---- réception UART par interruption : tampon circulaire ISR -> Terminal
#define RXN 8
char  rx_buf[RXN];
int8  rx_w = 0, rx_r = 0;                    // rx_w écrit par l'ISR seule (octet : atomique)

#int_rda
void isr_rx(void)
{
    rx_buf[rx_w] = getc();
    rx_w = (rx_w + 1) % RXN;
}

// ---- émission : une trame envoyée caractère par caractère
char  tx_buf[32];
int8  tx_i = 0, tx_n = 0;

// ====================================================================
#task(rate=200ms, max=1ms)
void Live()
{
    output_toggle(LED_VIE);                  // « le système vit »
}
```

```
62
63  #task(rate=50ms, max=1ms)
64  void Clavier()
65  {
66      static int1 prec_p = 1, prec_m = 1;
67      int1 p, m;
68      p = input(BP_PLUS);
69      m = input(BP_MOINS);
70      rtos_wait(sem_cons);
71      if (!p && prec_p && consigne < 250.0) consigne += 5.0;   // front descendant
72      if (!m && prec_m && consigne >  30.0) consigne -= 5.0;
73      rtos_signal(sem_cons);
74      prec_p = p;                              // échantillonnage à 50 ms :
75      prec_m = m;                              // filtre les rebonds
76  }
77
78  #task(rate=100ms, max=1ms)
79  void Mesure()
80  {
81      int16 somme = 0;
82      int8 i;
83      for (i = 0; i < 4; i++) {                // moyenne de 4 conversions
84          delay_us(10);
85          somme += read_adc();
86      }
87      rtos_wait(sem_etat);
88      mesure = (float)somme * (500.0 / 1023.0 / 4.0);  // 10 mV/°C, Vref = 5 V
89      rtos_signal(sem_etat);
90  }
91
92  #task(rate=500ms, max=2ms)
93  void PID()
94  {
95      static float I = 0.0, D = 0.0, y_prec = 25.0;
96      float w, y, e, v, u;
97      rtos_wait(sem_cons);  w = consigne;  rtos_signal(sem_cons);
98      rtos_wait(sem_etat);  y = mesure;    rtos_signal(sem_etat);
99      e = w - y;
100     D = (TD / (TD + NF * TE)) * D - (KP * TD * NF / (TD + NF * TE)) * (y - y_prec);
101     v = KP * e + I + D;                      // commande non saturée (%)
102     u = v;
103     if (u > 100.0) u = 100.0;
104     if (u < 0.0)   u = 0.0;
105     // anti-windup : intégration conditionnelle
106     if ((u == v) || (u >= 100.0 && e < 0.0) || (u <= 0.0 && e > 0.0))
107         I += KP * TE / TI * e;
108     y_prec = y;
109     set_pwm1_duty((int16)(u * 10.0));        // 0..1000 <=> 0..100 %
110     rtos_wait(sem_etat);  commande = u;  rtos_signal(sem_etat);
111 }
112
113 #task(rate=100ms, max=1ms)
114 void Securite()
115 {
116     float y;
117     rtos_wait(sem_etat);  y = mesure;  rtos_signal(sem_etat);
118     if (!alarme && y > T_MAX) {
119         alarme = 1;
120         rtos_disable(PID);                   // plus de régulation...
121         set_pwm1_duty(0);                    // ...et chauffe coupée
122         output_high(LED_ALARME);
123     }
124 }
125
126 #task(rate=250ms, max=5ms)
```

```
127 void Afficheur()                              // une ligne par activation
128 {
129     static int1 ligne = 0;
130     float w, y, u;
131     rtos_wait(sem_cons);  w = consigne;  rtos_signal(sem_cons);
132     rtos_wait(sem_etat);  y = mesure;  u = commande;  rtos_signal(sem_etat);
133     if (ligne == 0) {
134         lcd_gotoxy(1, 1);
135         printf(lcd_putc, "Cons:%5.1f%cC   ", w, 0xDF);
136     } else {
137         lcd_gotoxy(1, 2);
138         if (alarme) printf(lcd_putc, "ALARME T>%3.0fC  ", T_MAX);
139         else        printf(lcd_putc, "T:%5.1f U:%3.0f%% ", y, u);
140     }
141     ligne = !ligne;
142 }
143
144 #task(rate=10ms, max=1ms)
145 void Terminal()
146 {
147     static char cmd[6];
148     static int8 n = 0, compteur = 0;
149     static int1 cmd_prete = 0;
150     char c;
151     int16 val;
152     float y, u, w;
153
154     // 1) caractères reçus par l'ISR : commandes "S150<CR>" (consigne) et "R<CR>" (réarmement)
155     while (rx_r != rx_w) {
156         c = rx_buf[rx_r];
157         rx_r = (rx_r + 1) % RXN;
158         if (c == '\r') { cmd[n] = 0; n = 0; cmd_prete = 1; }
159         else if (n < 5) cmd[n++] = c;
160     }
161     if (cmd_prete) {
162         cmd_prete = 0;
163         if (cmd[0] == 'S') {
164             val = atol(cmd + 1);
165             if (val >= 30 && val <= 250) {
166                 rtos_wait(sem_cons);  consigne = val;  rtos_signal(sem_cons);
167             }
168         } else if (cmd[0] == 'R' && alarme && mesure < T_MAX - 20.0) {
169             alarme = 0;                          // réarmement manuel
170             output_low(LED_ALARME);
171             rtos_enable(PID);
172         }
173     }
174     // 2) émission d'un seul caractère par activation (1,04 ms à 9600 bauds)
175     if (tx_i < tx_n) {
176         putc(tx_buf[tx_i++]);
177     } else if (++compteur >= 100) {          // 3) nouvelle trame chaque seconde
178         compteur = 0;
179         rtos_wait(sem_etat);  y = mesure;  u = commande;  rtos_signal(sem_etat);
180         rtos_wait(sem_cons);  w = consigne;  rtos_signal(sem_cons);
181         sprintf(tx_buf, "W=%5.1f T=%5.1f U=%3.0f\r\n", w, y, u);
182         tx_n = strlen(tx_buf);
183         tx_i = 0;
184     }
185 }
186
187 // ====================================================================
188 void main()
189 {
190     setup_adc_ports(sAN0);                   // capteur sur AN0
191     setup_adc(ADC_CLOCK_DIV_64);             // Tad = 1 µs à 64 MHz
```

```
    set_adc_channel(0);
    setup_timer_2(T2_DIV_BY_16, 249, 1);     // PWM : 64 MHz / (4 x 16 x 250) = 4 kHz
    setup_ccp1(CCP_PWM);                     // sortie RC2, résolution 4 x 250 = 1000 pas
    set_pwm1_duty(0);
    lcd_init();
    sem_cons = 1;                            // sémaphores initialisés AVANT rtos_run
    sem_etat = 1;
    enable_interrupts(INT_RDA);
    enable_interrupts(GLOBAL);
    rtos_run();
}
```

5.8 Validation

1. **Temporelle** : vérifier à l'analyseur logique de Proteus la période réelle de chaque tâche (basculer une broche libre à chaque exécution), relever les durées par `rtos_stats` et contrôler `rtos_overrun`. Comparer aux valeurs `max` déclarées.
2. **Fonctionnelle** : boutons, commandes `S` et `R`, affichage, trames série, LED de vie.
3. **Performances de la régulation** : relever la réponse à un échelon de consigne (trames série ou graphe Proteus), mesurer dépassement et temps de réponse, comparer à la simulation (figure 5.4) ; tester un échelon de perturbation (porte ouverte : modifier la température ambiante ou la résistance thermique du modèle).
4. **Sécurité** : forcer une consigne excessive (ou désactiver la mesure) et vérifier la coupure en moins de 200 ms et l'impossibilité de réarmer au-dessus de $T_{\max} - 20$ °C.

5.9 Mise en pratique (TP, séances 7 à 9)

Phase 1 (séances 7–8) : matériel, identification et premières tâches

1. Saisir le schéma sous Proteus : PIC18F45K22 (64 MHz), modèle `OVEN` et son adaptation, étage de puissance, LCD, boutons, LED, `VIRTUAL TERMINAL`.
2. Écrire et valider les tâches `Live`, `Mesure`, `Afficheur` et `Terminal` (trame de mesure chaque seconde).
3. Réaliser l'essai indiciel en boucle ouverte (u constant, par exemple 50 %), enregistrer la réponse, identifier K, τ, θ_r (méthode de Broïda), puis calculer les réglages PID (ZN et CHR) et simuler la boucle fermée sous Matlab/Simulink ou Python.

Phase 2 (séance 9) : régulation, sécurité et rapport

1. Ajouter les tâches `PID`, `Clavier` et `Securite` ; régler la période d'échantillonnage.
2. Valider les performances (dépassement, temps de réponse, erreur statique) et le comportement temporel (périodes, durées, dépassements).
3. Rédiger un rapport selon les règles de l'art : cahier des charges, schéma, identification, synthèse du correcteur, découpage en tâches et analyse temporelle, code commenté, résultats de simulation (captures d'écran), discussion des écarts.

À retenir

— Démarche : spécifier, architecturer, modéliser, synthétiser la commande, découper en tâches, valider temporellement, coder, tester.

— Procédé thermique ≈ premier ordre + retard ; identification par Broïda ; réglage ZN ou CHR ; $T_e \leq \tau/10$; toujours un anti-windup.

— Avec un RTOS coopératif : tâches courtes, opérations lentes découpées (un caractère, une ligne par activation), entrées rapides sous interruption, vérification $\sum C_i <$ `minor_cycle`.