Université de Douala — École Supérieure Technique La Salle

Systèmes Temps Réel

Diapositives du cours — chapitres 1 à 5

Pr Moffo Lonla Bertrand

IIA5 — Année académique 2026–2027

Chapitre 1

Introduction aux systèmes temps réel

Objectifs du cours

- Analyser un problème multitâche temps réel : **synchronisation** des tâches, **protection** des ressources partagées.
- Vérifier qu'un ensemble de tâches respecte ses contraintes temporelles (RM, DM, EDF, LLF).
- Programmer la solution en C avec un **noyau temps réel** (RTOS de CCS C, FreeRTOS, threads POSIX).
- Concevoir, simuler (Proteus) et valider un STR complet : **asservissement de température** d'un four (PIC18F45K22).

Organisation — 36 h

CM 20 h TD 8 h TP 8 h — 9 séances, du 21/11/2026 au 30/01/2027 — projet en TP (séances 7 à 9)

Plan du cours

1. Introduction aux systèmes temps réel
2. Modèle de tâches et ordonnancement temps réel
3. Ressources partagées, synchronisation et communication
4. Systèmes d'exploitation temps réel et multitâche (CCS C, FreeRTOS, Proteus)
5. Conception d'un STR : asservissement de température

Prérequis : langage C, microcontrôleurs PIC (ports, timers, CAN, UART, interruptions), notions de systèmes d'exploitation et d'automatique (1^{er} ordre, PID).

Qu'est-ce qu'un système temps réel ?

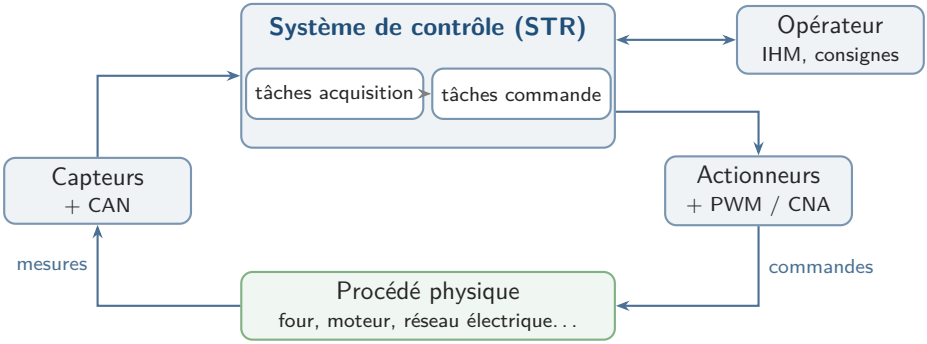
Un résultat juste mais tardif est un résultat faux

- Programme classique : correct s'il donne le **bon résultat**.
- Dans l'industrie, l'embarqué, le contrôle de procédés : il faut aussi le donner **à temps**.
  - airbag déployé 200 ms après le choc ;
  - injection calculée après le passage du piston ;
  - alarme de surchauffe traitée après la destruction de la charge.

Système temps réel (STR)

Système informatique dont la correction dépend non seulement de la justesse logique des résultats, mais aussi de la **date** à laquelle ils sont produits, en réponse aux événements de son environnement (le *procédé*).

Architecture d'un système de contrôle-commande



Le STR **acquiert** (capteurs), **calcule** (lois de commande), **agit** (actionneurs) — périodiquement ou sur événement — au rythme imposé par la **dynamique du procédé**.

Temps réel $\neq$ rapide

Contresens fréquent

Un STR doit être **prévisible** (*déterministe*) : on doit **garantir avant l'exécution** que chaque échéance sera respectée, *y compris dans le pire cas*.

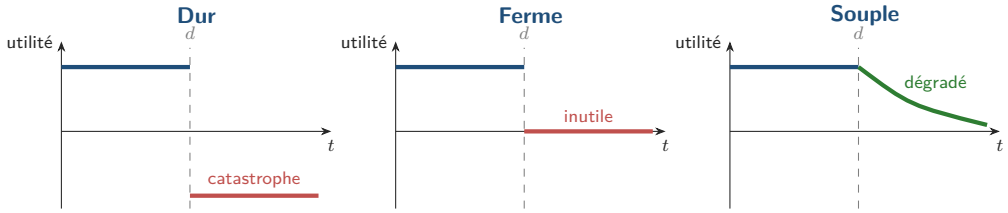
- Un système rapide *en moyenne* mais imprévisible (cache, mémoire virtuelle, ramasse-miettes) n'est **pas** temps réel.
- C'est la **maîtrise du pire cas** qui compte.

Procédé	échéance
courant d'onduleur	10–100 µs
moteur, ABS, airbag	1–10 ms
protection réseau	10–100 ms
four	0,1–10 s
IHM	0,1–1 s

Un PIC à 64 MHz est « temps réel » pour un four, pas pour un onduleur.

Classes de contraintes et caractéristiques

Temps réel dur, ferme, souple



- **Dur** : un seul dépassement peut être catastrophique — airbag, ABS, commande de vol, arrêt d'urgence.
- **Ferme** : un résultat tardif est *inutile* (jeté), dépassement occasionnel toléré — trame vidéo, mesure radar.
- **Souple** : un résultat tardif garde une valeur *dégradée* — streaming, IHM, réservation.

Caractéristiques d'un STR

- **Déterminisme / prévisibilité** : durées bornées et connues.
- **Réactivité** : latence maîtrisée.
- **Respect des échéances** garanti hors ligne.
- **Concurrence** : multitâche, synchronisation.
- **Fiabilité, sûreté**, fonctionnement continu.
- **Ressources limitées** (embarqué).

Système embarqué

Système électronique *et informatique* autonome, souvent temps réel, spécialisé dans une tâche précise et intégré au produit qu'il contrôle.

Tous les systèmes embarqués ne sont pas temps réel dur (calculatrice), tous les STR ne sont pas embarqués (supervision sur PC).

Expression des contraintes temporelles

Les contraintes s'expriment sur les **tâches** :

- une **période** T : acquisition toutes les 100 ms, régulation toutes les 0,5 s ;
- une **échéance** relative D : commande appliquée moins de 5 ms après la mesure ;
- une **date de réveil** r : appui sur un bouton, caractère reçu ;
- une **gigue** admissible ;
- un **temps de réponse** maximal de bout en bout.

Fil rouge : le four à micro-ondes (chapitre 5)

Mesurer toutes les 100 ms, calculer le PID toutes les 500 ms, rafraîchir le LCD, scruter le clavier, dialoguer avec un terminal, LED de vie. Régulation **dure** (surchauffe), affichage **souple**.

Exemples de systèmes temps réel

- **Automobile** : injection, ABS, ESP, airbag, bus CAN.
- **Énergie** : relais de protection, régulation tension/fréquence, compteurs intelligents.
- **Industrie** : automates, robots, régulation de procédés.
- **Aéronautique, spatial** : commandes de vol, pilote automatique.
- **Médical** : pacemaker, respirateur.
- **Vie courante** : four à micro-ondes, machine à laver, ascenseur, distributeur de billets.

Structures logicielles

De la boucle infinie au multitâche

1. **Boucle infinie** (*super-loop*) : activités appelées en séquence dans `while(1)`.
   $\rightarrow$ temps de réponse = durée de toute la boucle.
2. **Boucle + interruptions** (*foreground/background*) : l'urgent en interruption, le reste en fond.
   $\rightarrow$ partage de données délicat.
3. **Noyau temps réel multitâche (RTOS)** : tâches indépendantes, **ordonnanceur**, outils de synchronisation et de communication.
   $\rightarrow$ objet des chapitres 2 à 4.

TD 1, ex. 4

Bouton d'urgence (≤ 5 ms), mesure 50 µs, PID 1 ms, LCD 8 ms dans une boucle : pire délai $\approx 9{,}1$ ms > 5 ms. Solutions : interruption, découpage, RTOS.

À retenir

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), **ferme** (tardif = inutile), **souple** (tardif = dégradé).
- Contraintes : périodes, échéances, dates de réveil, temps de réponse — imposées par la dynamique du procédé.

TD 1 : définitions, QCM, classification de systèmes, boucle infinie ou RTOS.

Chapitre 2

Modèle de tâches et ordonnancement temps réel

Objectifs de la séance

- Modéliser une tâche périodique : r, C, D, T ; temps de réponse, laxité, charge, hyperpériode.
- Comparer les algorithmes classiques (FIFO, SJF, tourniquet, priorités) : rotation, attente, débit.
- Appliquer les algorithmes temps réel **RM, DM, EDF, LLF** et tracer leurs chronogrammes.
- Prouver l'ordonnançabilité : critère de **Liu et Layland**, **analyse du temps de réponse**, $U \leq 1$ pour EDF.

TD 2 : algorithmes classiques **TD 3** : RM, DM, EDF, LLF (séance 3)

Modèle de tâches

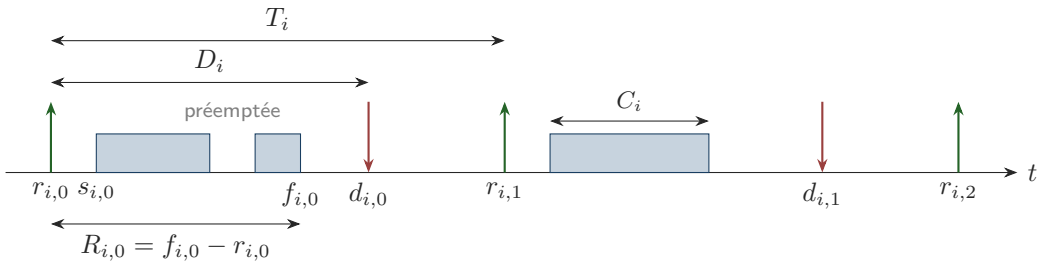
Tâche, processus, instance

Définitions

Tâche : entité d'exécution séquentielle réalisant une activité (mesurer, calculer, afficher). **Processus** : programme en cours d'exécution avec son propre espace mémoire (peut contenir plusieurs *threads*). Une tâche périodique τ_i réveillée toutes les T_i engendre des **instances** $\tau_{i,k}$.

- **Périodique** : réveil régulier (échantillonnage, régulation).
- **Apériodique** : réveil sur événement imprévisible.
- **Sporadique** : sur événement, mais délai minimal connu entre deux réveils.
- **Indépendantes** ou **dépendantes** (ressources, précédence, messages).

Paramètres temporels : $\tau_i(r_i, C_i, D_i, T_i)$



- $r_{i,k} = r_i + k\,T_i$; échéance absolue $d_{i,k} = r_{i,k} + D_i$.
- C_i : durée **pire cas** (WCET).
- $0 < C_i \leq D_i \leq T_i$.

- Temps de réponse $R_i = \max_k(f_{i,k} - r_{i,k})$.
- Échéances respectées $\iff$ $R_i \leq D_i$.

Laxité, charge, hyperpériode

Grandeurs fondamentales

- **Laxité** : $L(t) = d - t - C_{\text{restant}}(t)$ — marge avant l'échéance.
- **Facteur d'utilisation** : $u_i = C_i/T_i$; **charge** $U = \sum_i C_i/T_i$.
- **Hyperpériode** : $H = \text{ppcm}(T_1, \ldots, T_n)$ — la séquence se répète toutes les H (tâches synchrones).

Condition nécessaire

Si $U > 1$, **aucun** algorithme ne peut respecter toutes les échéances.

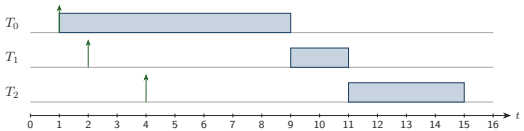
- **Ordonnanceur** : partie du noyau qui choisit la tâche élue ; sa règle = l'**algorithme**.
- Préemptif / non préemptif ; hors ligne / en ligne ; priorités fixes / dynamiques ; optimal ou non.

Algorithmes classiques

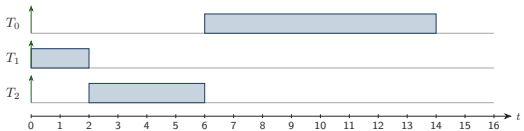
Critères de performance (tâches à exécution unique)

- **Rotation** $= f_i - a_i$ **Attente** $=$ rotation $- C_i$ **Réponse** $= s_i - a_i$
- **Débit** $= n$ / (fin de la dernière $-$ première arrivée)

FIFO (non préemptif) : ordre d'arrivée ; effet convoi.



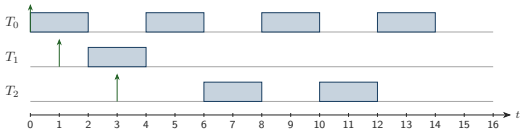
SJF (non préemptif) : plus court d'abord ; minimise l'attente moyenne.



Conventions : temps discret ; à priorité égale on garde la tâche en cours, sinon l'indice le plus faible.

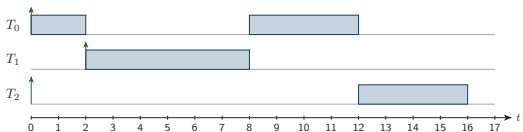
Tourniquet et priorités fixes

Tourniquet (quantum $q = 2$) : équitable, mais ignore l'urgence.



À un même instant, l'arrivée passe avant la tâche dont le quantum expire.

Priorités fixes préemptives : T_1 (prio 1) préempte T_0 à $t = 2$.



Question

L'ordonnancement par priorités est la base de tous les ordonnanceurs temps réel. Question : **comment choisir les priorités** pour garantir les échéances ?

Ordonnancement temps réel

Rate Monotonic (RM)

RM

Priorités **fixes**, inversement proportionnelles à la **période** ; préemptif.

- Optimal parmi les priorités fixes ($D = T$).
- **Liu et Layland (1973)** :

$$U \leq n\left(2^{1/n} - 1\right) \Rightarrow \text{ordonnançable}$$

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

Suffisante, mais pas nécessaire

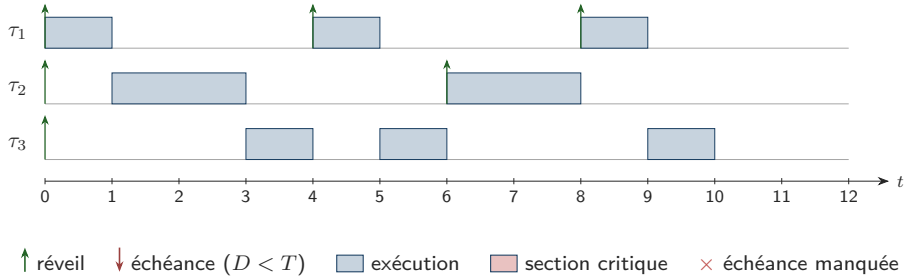
$U \leq U_{\text{LL}}$: ordonnançable. $U_{\text{LL}} < U \leq 1$: **on ne peut pas conclure** $\rightarrow$ test exact.
$U > 1$: impossible.

Analyse du temps de réponse (test exact)

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \qquad \text{(itérer à partir de } R_i^{(0)} = C_i + \textstyle\sum_{hp} C_j\text{)}$$

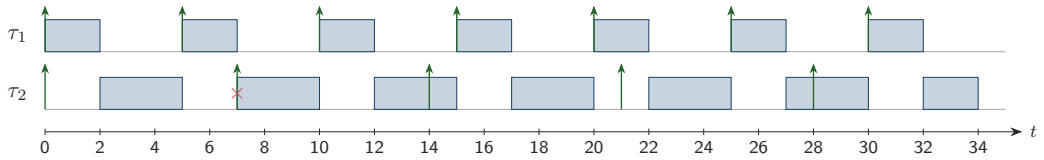
$\tau_1(1,4)$, $\tau_2(2,6)$, $\tau_3(3,12)$ — $U = 0{,}833 > 0{,}780$

$R_1 = 1$, $R_2 = 3$, $R_3 : 6 \to 7 \to 9 \to 10 \to 10 \leq 12$: **ordonnançable**.



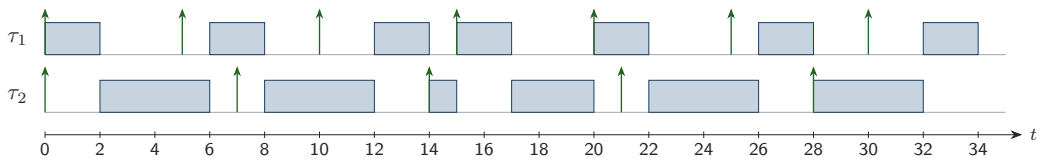
Échec de RM, réussite d'EDF

$\tau_1(2,5)$, $\tau_2(4,7)$, $U = 0{,}971$. RM : $R_2 : 6 \to 4 + \lceil 6/5 \rceil \cdot 2 = 8 > 7$: **échec**.



EDF — Earliest Deadline First

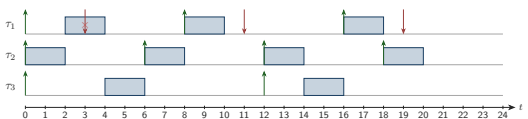
Priorité **dynamique** : l'instance dont l'**échéance absolue** est la plus proche. Pour $D = T$: ordonnançable $\iff U \leq 1$. **Optimal**.



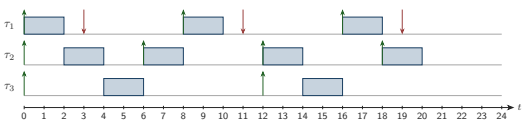
Deadline Monotonic (DM) : échéances contraintes $D < T$

$\tau_1(C{=}2, D{=}3, T{=}8)$, $\tau_2(2, 6, 6)$, $\tau_3(2, 12, 12)$, $U = 0{,}75$.

RM : τ_2 $(T = 6)$ avant $\tau_1 \rightarrow \tau_1$ finit à $4 > 3$.



DM : priorité à la plus petite **échéance relative** D_i.



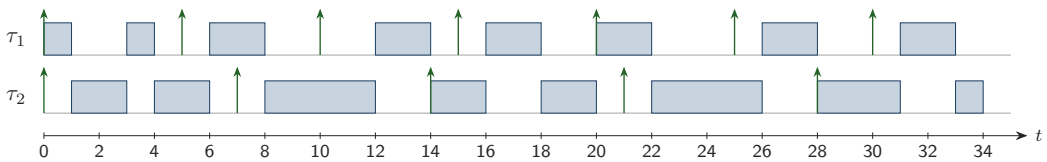
- DM est optimal parmi les priorités fixes pour $D \leq T$; si $D = T$, DM = RM.
- Test : analyse du temps de réponse avec l'ordre DM.

Least Laxity First (LLF)

LLF

Priorité **dynamique** à la plus petite laxité $L(t) = d - t - C_{\text{restant}}(t)$. Optimal ($U \leq 1$ si $D = T$).

Même jeu $\tau_1(2,5)$, $\tau_2(4,7)$: à $t = 0$, $L_1 = L_2 = 3$; à $t = 1$, $L_1 = 3$, $L_2 = 2 \rightarrow \tau_2$ préempte.



Coût

4 préemptions au lieu d'une avec EDF : les laxités se croisent sans cesse → changements de contexte coûteux.

Synthèse des algorithmes temps réel

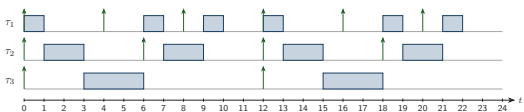
Algorithme	Priorité	Critère	Test	Optimalité
RM	fixe	T_i croissant	LL (suff.), RTA (exact)	prio. fixes, $D = T$
DM	fixe	D_i croissant	RTA (exact)	prio. fixes, $D \leq T$
EDF	dynamique	$d_{i,k}$ la plus proche	$U \leq 1$ $(D = T)$	optimal
LLF	dynamique	laxité minimale	$U \leq 1$ $(D = T)$	optimal, coûteux

- RTOS industriels (FreeRTOS, VxWorks, μC/OS) : **préemptif à priorités fixes** — le concepteur applique RM/DM.
- EDF : Linux `SCHED_DEADLINE`, ERIKA.
- RTOS de CCS : **coopératif** (non préemptif), cadencé par le temps.

Non préemptif, apériodique, hors ligne

Non préemptif (noyau coopératif) : une tâche urgente attend la fin de la tâche en cours :

$$B_i = \max_{j \in lp(i)} C_j$$



Apériodiques : pas de priorité RM possible.

- arrière-plan (priorité minimale) ;
- serveur périodique (scrutation, différé, sporadique) ;
- sporadique $\rightarrow$ périodique de période = délai minimal.

Hors ligne (cyclique) : table sur l'hyperpériode, cycles mineurs — déterministe mais rigide.

Avec CCS

Une tâche longue retarde toutes les autres : découper avec `rtos_yield()`.

À retenir

À retenir

- $\tau_i(r, C, D, T)$; $R = f - r$; $L = d - t - C_{\text{rest}}$; $U = \sum C_i/T_i$; $H = \text{ppcm}(T_i)$.
- $U > 1$: impossible. EDF, LLF : $U \leq 1$ suffit ($D = T$).
- RM : $U \leq n(2^{1/n} - 1)$ **suffisant, non nécessaire** ; sinon analyse du temps de réponse ou chronogramme sur H.
- DM généralise RM aux échéances contraintes ; LLF multiplie les préemptions.

TD 2 : FIFO, SJF, SRTF, tourniquet, priorités. RM/DM/EDF/LLF, C maximal, non préemptif.

TD 3 : Liu-Layland, RTA,

Chapitre 3

Ressources partagées, synchronisation et communication

Objectifs de la séance

- Identifier les **ressources partagées** et les **sections critiques** d'une application.
- Protéger une ressource : masquage, mutex, **sémaphores** (P/V).
- Résoudre le **producteur / consommateur** ; éviter l'**interblocage**.
- Comprendre l'**inversion de priorité** et ses remèdes (PIP, PCP/ICPP) ; l'intégrer dans l'analyse du temps de réponse.
- Expérimenter sur PC avec les **threads POSIX**.

TD 4 : condition de course, producteur/consommateur, PIP/ICPP, blocage, interblocage (séance 5)

Ressources partagées et section critique

La condition de course

`compteur = compteur + 1;` = LOAD, ADD, STORE.

	Tâche A	Tâche B	mém.
1	LOAD (5)		5
2	*préemptée*	LOAD (5)	5
3		ADD, STORE	6
4	ADD, STORE		**6**

Section critique

Code accédant à une ressource partagée (variable, LCD, UART) qui ne doit être exécuté que par **une seule tâche à la fois** (exclusion mutuelle).

Condition de course

Le résultat dépend de l'ordre d'accès, non maîtrisé : erreurs **intermittentes**.

Expérience sur PC : deux threads POSIX

```c
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;
}
```

Exécution :

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

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

POSIX

Norme IEEE 1003 de l'interface programme / système d'exploitation ; sa partie temps réel normalise threads, mutex, sémaphores, files de messages, ordonnancement à priorités. Linux, macOS, QNX, VxWorks… (Windows : WSL ou MSYS2).

Mécanismes d'exclusion mutuelle

- **Propriétés attendues** : exclusion mutuelle, progrès, attente **bornée** (et calculable), aucune hypothèse sur les vitesses.
- **Masquage des interruptions** : `disable_interrupts(GLOBAL)` ... `enable_interrupts(GLOBAL)` — sections très courtes.
- **Instructions atomiques** (*test-and-set*) : verrous à attente active, multiprocesseur.
- **Noyau coopératif** (CCS) : pas de préemption entre tâches tant qu'on ne rend pas la main — mais rien contre les *interruptions*.

Sémaphores et mutex

Le sémaphore de Dijkstra

Sémaphore S = compteur ≥ 0 + file d'attente

$P(S)$ (*wait*) : si $S > 0$, $S \leftarrow S - 1$; sinon la tâche se **bloque**.
$V(S)$ (*signal*) : réveille une tâche bloquée, sinon $S \leftarrow S+1$. (opérations **atomiques**)

```
// exclusion mutuelle : S := 1
P(S)
   section critique
V(S)
```

```
// synchronisation : E := 0
Mesure : lire CAN ; V(E)
PID    : P(E) ; calculer u
```

- Binaire (1) : exclusion ; à compteur (N) : N ressources ; à 0 : attendre un événement.
- **Mutex** : sémaphore binaire avec **propriétaire** et protocole anti-inversion.

Producteur / consommateur

```
places := N ; mesures := 0 ; mutex := 1
Producteur :          Consommateur :
  produire x            P(mesures)
  P(places)             P(mutex)
  P(mutex)                x := T[il]
    T[ie] := x          V(mutex)
  V(mutex)              V(places)
  V(mesures)            consommer x
```

Ordre des P

`P(mutex)` avant `P(places)` $\Rightarrow$ **interblocage** quand le tampon est plein.

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
```

Implémentation POSIX : `code/cours/prodcons.c` (sémaphore construit avec mutex + variable de condition).

Autres outils de communication

- **Files de messages** : FIFO gérée par le noyau, blocage si vide/pleine — `xQueueSend/Receive`, `rtos_msg_send/read`.
- **Boîte aux lettres** : capacité 1, dernière valeur (état).
- **Drapeaux d'événements** : bits positionnés / attendus (ET, OU).
- **Tubes** : flux d'octets.
- **Variable partagée protégée** ; sans protection si atomique et à **un seul écrivain**.

Notion	POSIX	FreeRTOS	CCS
mutex	`pthread_mutex_lock`	`xSemaphoreTake`	`rtos_wait`
sémaphore P/V	`sem_wait/post`	`Take/Give`	`rtos_wait/signal`
condition	`pthread_cond_wait`	notifications	`rtos_await`
messages	`mq_send/receive`	`xQueueSend/Receive`	`rtos_msg_send/read`

Interblocage et inversion de priorité

Interblocage

```
Tâche A :  P(LCD) ; P(UART) ; ... ; V(UART) ; V(LCD)
Tâche B :  P(UART) ; P(LCD) ; ... ; V(LCD) ; V(UART)
```

Conditions de Coffman (1971) — toutes nécessaires

(1) exclusion mutuelle (2) détention et attente (3) pas de réquisition (4) attente circulaire

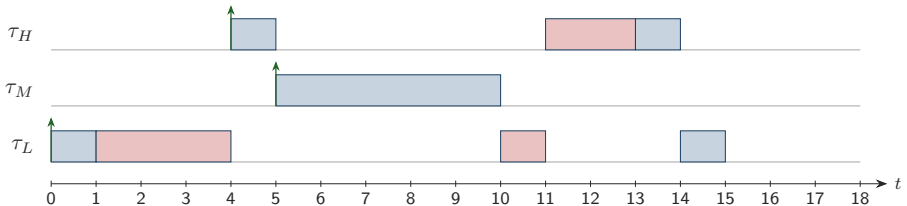
À retenir

Casser une condition : **prendre les ressources dans un ordre global unique** (toujours LCD avant UART), ne détenir qu'une ressource à la fois, ou utiliser le plafond de priorité.

Famine : attente indéfinie d'une tâche jamais servie.

Inversion de priorité

τ_H (prio 1, réveil 4, $C = 4$ dont 2 sur R), τ_M (prio 2, réveil 5, $C = 5$), τ_L (prio 3, réveil 0, $C = 6$ dont 4 sur R).



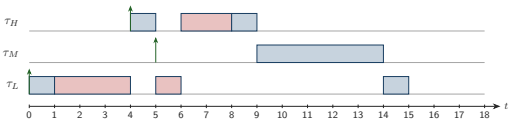
Sans protocole

τ_H, bloquée sur R détenue par τ_L, attend **toute** l'exécution de τ_M : $R_H = 10$. Mars Pathfinder (1997) : redémarrages en boucle, corrigé à distance en activant l'héritage de priorité.

↑ réveil ↓ échéance ($D < T$) exécution section critique × échéance manquée

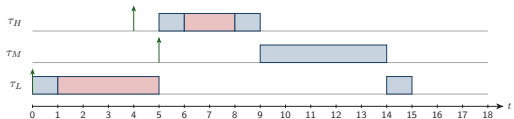
Héritage de priorité (PIP) et plafond immédiat (ICPP)

PIP : τ_L **hérite** de la priorité de τ_H tant qu'elle la bloque.



$R_H = 5$. Borne le blocage ; n'empêche ni interblocage ni blocage en chaîne.

ICPP : en prenant R, τ_L monte **immédiatement** au **plafond** de R.



$R_H = 5$. Blocage **une seule fois**, **aucun interblocage**.

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \qquad B_i = \text{plus longue section critique bloquante}$$

À retenir

- Toute donnée partagée modifiée par plusieurs tâches doit être protégée (exclusion mutuelle).
- Sémaphore = compteur + file, P/V atomiques ; mutex = sémaphore binaire à propriétaire.
- Producteur/consommateur : `places`$= N$, `mesures`$= 0$, `mutex`$= 1$; P(ressource) *avant* P(mutex).
- Interblocage : 4 conditions de Coffman ; ordre global d'acquisition.
- Inversion de priorité : PIP la borne, PCP/ICPP la supprime avec les interblocages ; B_i dans la RTA.

Chapitre 4

Systèmes d'exploitation temps réel et multitâche

Objectifs des séances

- Passer de la machine à états au **noyau temps réel** : ordonnanceur, contexte, états des tâches.
- Lire et écrire un programme **FreeRTOS** (préemptif) et un programme **CCS C** (coopératif).
- Maîtriser `#use rtos`, `#task` et les fonctions `rtos_*` : messages, sémaphores, attentes.
- Dimensionner `minor_cycle` et `max` ; mesurer les durées avec `statistics`.
- Câbler et simuler les montages sous **Proteus**.

TD 5 : RTOS de CCS et Proteus (séance 5) **TD 6** : études de cas (séance 6)

De la machine à états au RTOS

La machine à états

```
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);
}
```

- Simple, totalement déterministe.
- Ordonnancement **coopératif** primitif.
- Aucune priorité, aucune préemption.
- Temps de réponse = durée de toute la séquence.

RTOS

Noyau multitâche qui alloue le processeur aux tâches selon une politique **déterministe** et fournit des services (temps, synchronisation, communication) à durée **bornée** : **ordonnanceur** + **services** + **outils de synchronisation**.

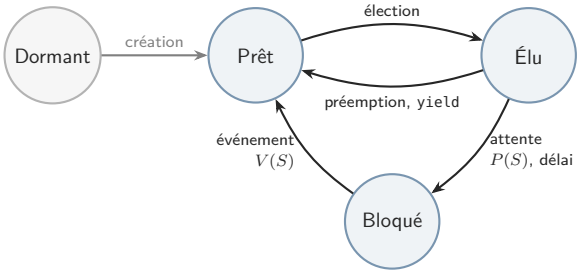
Contexte et familles d'ordonnanceurs

- **Contexte** : compteur ordinal, registres, pointeur de pile, état → rangé dans le **TCB** (*Task Control Block*).
- **Changement de contexte** : sauvegarde du TCB sortant, restauration du TCB élu (quelques µs).

Famille	Principe	Exemple
Coopératif	la tâche garde le CPU jusqu'à ce qu'elle rende la main	machine à états, **CCS**
Tourniquet	tranche de temps égale pour chaque tâche	temps partagé
Préemptif à priorités	la tâche prête la plus prioritaire obtient le CPU	**FreeRTOS**, VxWorks

Priorités 0–255 : 0 est la plus forte (μC/OS) ou la plus faible (FreeRTOS) selon le noyau.

États d'une tâche



Transitions

Seule une tâche **élue** peut se bloquer. Une tâche débloquée redevient **prête** ; elle n'est élue que si elle est la plus prioritaire.

Exemple valide : Élu → Bloqué → Prêt → Élu.

Un RTOS préemptif : FreeRTOS

```c
void tache_mesure(void *p) {                        // période 100 ms, priorité 3
    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 prochain réveil
    }
}
void tache_pid(void *p) {                           // priorité 2
    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);
    }
}
```

Préemptif à priorités fixes : priorités choisies par RM/DM, ordonnançabilité vérifiée par RTA (avec blocages).

Le RTOS du compilateur CCS C

Principe du RTOS de CCS

- Compilateur **CCS C** (PCWH) : PIC10/12/16/18, PIC24, dsPIC ; types `int1`, `int8`, `int16`, `int32`.
- Un **timer** matériel cadence le RTOS ; à chaque **cycle mineur**, le **répartiteur** appelé par `rtos_run()` exécute *à tour de rôle* les tâches échues.
- **Coopératif, sans interruptions** : une tâche garde le CPU jusqu'à sa fin ou `rtos_yield` / `rtos_wait` / `rtos_await`.

`#use rtos(timer=0, minor_cycle=10ms)`

`minor_cycle` $\geq$ plus longue tâche ; toutes les périodes en sont des **multiples**. Option `statistics`.

`#task(rate=20ms, max=1ms, queue=2)`

Fonction sans argument ni retour. `rate` : période ; `max` : durée prévue ($\leq$ cycle mineur) ; `queue` : octets de file.

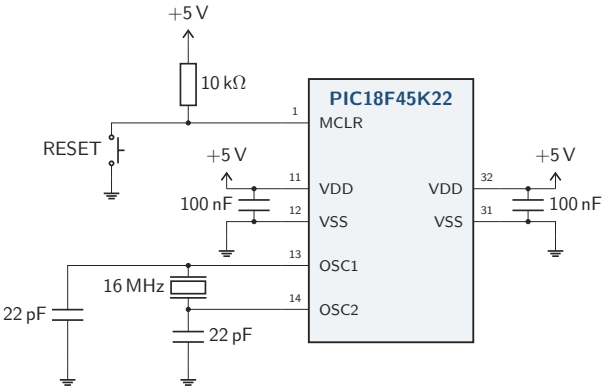
Les fonctions du RTOS de CCS

Fonction	Rôle	Fonction	Rôle
`rtos_run()`	démarre le répartiteur	`rtos_msg_send(t,b)`	envoie l'octet b à la tâche t
`rtos_terminate()`	arrête le RTOS	`rtos_msg_poll()`	nb d'octets dans sa file
`rtos_enable(t)`	active la tâche t	`rtos_msg_read()`	lit un octet de sa file
`rtos_disable(t)`	désactive t	`rtos_wait(s)`	$P(s)$: attend $s > 0$ puis $s{-}{-}$
`rtos_yield()`	rend la main, reprise après	`rtos_signal(s)`	$V(s)$: $s{+}{+}$
`rtos_await(e)`	rend la main tant que e est faux	`rtos_overrun(t)`	t a dépassé max ?
		`rtos_stats(t,&st)`	durées min, max, totale

Règles d'or

`rtos_yield/wait/await/msg_poll/msg_read` : **seulement dans une tâche**. Sémaphores initialisés **avant** `rtos_run()`. **Jamais** de boucle infinie ni de `delay_ms(500)` dans une tâche.

Montage de base du PIC18F45K22



- 5 V, découplage 100 nF par paire VDD/VSS.
- Quartz 16 MHz + 2 × 22 pF ; PLL $\times 4 \rightarrow$ 64 MHz.
- MCLR : 10 kΩ + bouton RESET.
- Numéros de broches : boîtier PDIP-40.
- Sous Proteus : alimentation implicite, fréquence dans les propriétés du PIC.

Exemple 1 : quatre LED indépendantes

```
#use rtos(timer=0, minor_cycle=10ms)

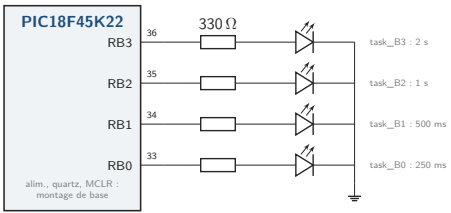
#task(rate=250ms, max=10ms)
void task_B0() { output_toggle(PIN_B0); }

#task(rate=500ms, max=10ms)
void task_B1() { output_toggle(PIN_B1); }

#task(rate=1s, max=10ms)
void task_B2() { output_toggle(PIN_B2); }

#task(rate=2s, max=10ms)
void task_B3() { output_toggle(PIN_B3); }
```

`main()` : `set_tris_b(0x00); rtos_run();`

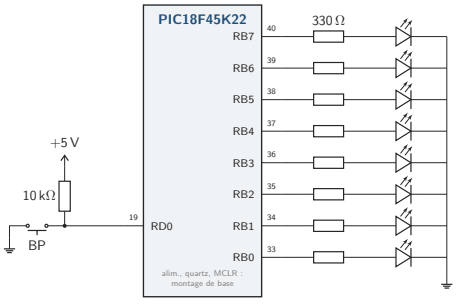


Chaque LED : période $= 2\times$ `rate` (500 ms, 1 s, 2 s, 4 s). Toutes les périodes sont multiples de 10 ms.

Exemple 2 : communication par messages

```
#task(rate=1ms, max=1ms)
void Generator()
{
    count++;
    if (input(PIN_D0) == 0) {
        rtos_msg_send(Display, count);
        rtos_await(input(PIN_D0) == 1);
    }
}

#task(rate=10ms, max=1ms, queue=2)
void Display()
{
    if (rtos_msg_poll() > 0)
        output_b(rtos_msg_read());
}
```

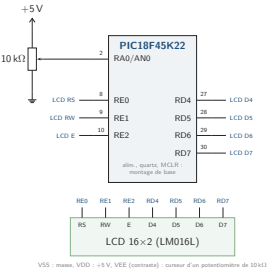


`Generator` compte toutes les 1 ms ; l'appui envoie la valeur à `Display` (nombre « aléatoire ») ; `rtos_await` évite les envois répétés.

Exemple 3 : exclusion mutuelle sur le LCD

```
#task(rate=1s, max=5ms)
void Horloge()
{
    secondes++;
    rtos_wait(sem_lcd);
    lcd_gotoxy(1, 1);
    printf(lcd_putc, "t = %5lu s   ", secondes);
    rtos_signal(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);
```



Coopératif : pas de préemption entre tâches, mais le sémaphore devient indispensable dès qu'une section critique contient `rtos_yield`/`rtos_await`, ou si la ressource est utilisée sous interruption. `sem_lcd = 1` dans `main()`.

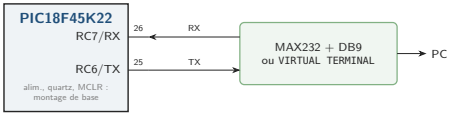
Mesurer les durées : option statistics

```
#use rs232(baud=115200, UART1, bits=8, parity=N)
#use rtos(timer=0, minor_cycle=5ms, statistics)

struct rtos_stats st;

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

#task(rate=1s, max=5ms)
void Superviseur()
{
    if (rtos_overrun(Calcul))
        printf("DEPASSEMENT\r\n");
    rtos_stats(Calcul, &st);
    printf("max=%lu t x%lu00ns\r\n", st.
     task_max_ticks, st.hns_per_tick);
}
```



Liaison série

À 9600 bauds, un caractère dure 1,04 ms : une trame de 50 caractères bloquerait 52 ms ! D'où 115 200 bauds et des trames courtes, ou un caractère par activation.

Ordonnançabilité avec le RTOS de CCS

1. Chaque tâche dure moins que `max`, et `max` $\leq$ `minor_cycle`.
2. Somme des durées des tâches échues **au même cycle mineur** $<$ cycle mineur (sinon décalage, gigue).
3. Charge $U = \sum C_i/T_i$ nettement inférieure à 1.

TD 5, ex. 4

Périodes 5, 10, 20, 100, 200 ms ; durées 0,12 + 0,8 + 1,5 + 2 + 3,2 = 7,62 ms à $t = 200$ ms $>$ 5 ms : débordement, gigue de 2,6 ms sur la tâche d'acquisition. $\rightarrow$ découper `Affichage` et `Liaison`.

Pire retard = modèle **non préemptif** (chapitre 2) ; découper les traitements longs avec `rtos_yield()`.

Proteus et études de cas

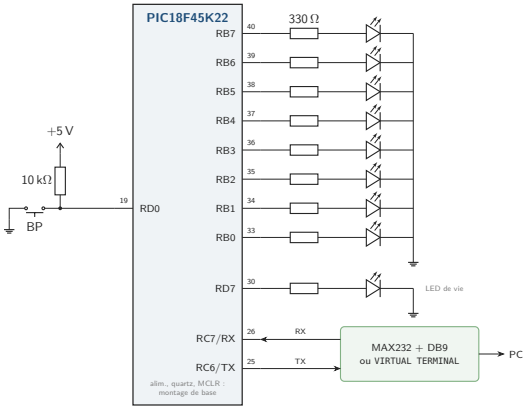
Chaîne de développement : CCS + Proteus

1. **CCS (PCWH)** : projet PIC18F45K22, horloge 16 MHz + PLL ; *Build* → `.hex` et `.cof`.
2. **Proteus (ISIS)** : PIC18F45K22, LED, boutons, `LM016L`, `VIRTUAL TERMINAL` / `COMPIM`, `LM35`, `OVEN`.
3. **Propriétés du PIC** : *Program File* = `.cof` ; *Processor Clock Frequency* = **64 MHz**.
4. **Simulation** : oscilloscope, analyseur logique (broche basculée en entrée/sortie de tâche → période, durée, gigue), pas à pas.

Attention

La fréquence saisie dans Proteus est la fréquence *après* PLL ; sinon toutes les durées sont fausses d'un facteur 4.

Étude de cas 1 : générateur de nombres aléatoires

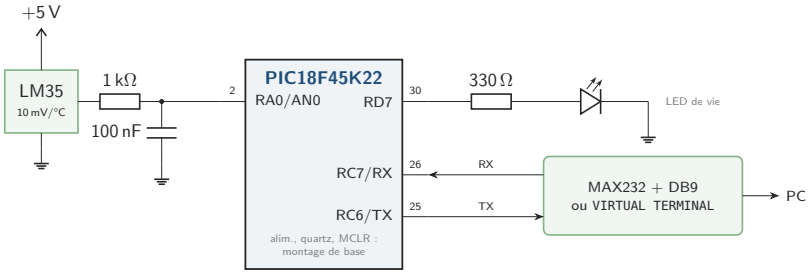


- `Live` (200 ms) : LED RD7.
- `Generator` (1 ms) : compte 0–255 ; sur appui ou `'N'` reçu, envoie la valeur à `Display` et `Terminal`.
- `Display` : octet reçu → PORTB.
- `Terminal` : caractères de l'UART → `Generator` ; nombres → PC.

Attention

`printf` de 5 caractères = 5,2 ms > cycle mineur (1 ms) : un caractère par activation.

Étude de cas 2 : voltmètre / thermomètre RS232



- `Live` (200 ms) ; `Get_voltage` (2 s) : CAN AN0 → mV et °C (LM35, 10 mV/°C) ; `To_RS232` (1 s) : envoi au PC.
- Données partagées `tension_mV`, `temperature` protégées par un **sémaphore**.
- Trame ≈ 26 caractères à 9600 bauds ≈ 27 ms ⇒ `max = 30ms`, `minor_cycle = 50ms`.

À retenir

À retenir

- RTOS = ordonnanceur + services + outils de synchronisation, à durées bornées.
- États : prêt, élu, bloqué (+ dormant) ; seule une tâche élue peut se bloquer.
- FreeRTOS : préemptif à priorités fixes ; CCS : coopératif, cadencé par `minor_cycle`.
- CCS : `max` $\leq$ `minor_cycle`, `rate` multiple de `minor_cycle`, aucune attente active, opérations lentes découpées, sémaphores initialisés avant `rtos_run()`.

TD 5 : analyse, recherche d'erreurs, chenillard, dimensionnement, Proteus. **TD 6** : études de cas.

Chapitre 5

Conception d'un STR : asservissement de température

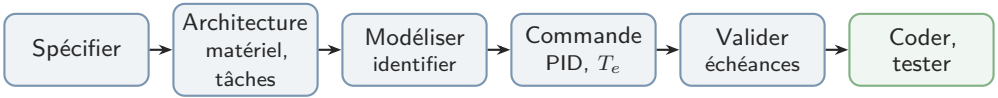
Objectifs de la séance

- Appliquer la **démarche de conception** d'un STR à un cas complet : le four à micro-ondes (modèle `0VEN` de Proteus) piloté par un PIC18F45K22.
- Identifier le procédé (**Broïda**) et régler un **PID numérique** avec anti-windup.
- Découper l'application en **tâches**, choisir les périodes et le cycle mineur, vérifier la **charge** et le **pire cycle**.
- Protéger les données partagées ; traiter l'UART sans bloquer les tâches.

TD 7 : conception **TP 1 à 3** (séances 7–9) : réalisation et validation sous Proteus

Démarche et cahier des charges

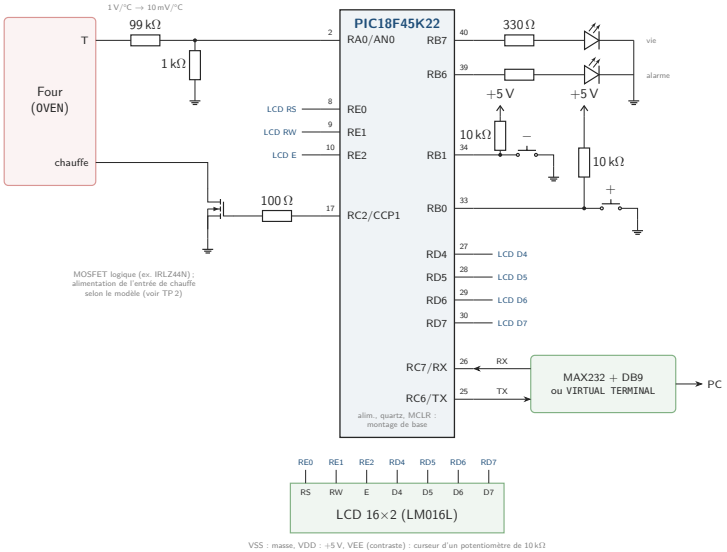
Démarche de conception d'un STR



Cahier des charges du four

- Consigne 30–250 °C (pas de 5 °C, boutons RB0/RB1 ou commande série `S150`).
- Erreur statique nulle, dépassement $\leq 10\ \%$, $t_{5\%} < 4$ min pour 25 $\rightarrow$ 150 °C.
- LCD : consigne / température et commande ; trame série chaque seconde.
- **Sécurité** : $T > 260$ °C $\Rightarrow$ chauffe coupée, LED d'alarme, réarmement `R` sous 240 °C ; réaction ≤ 200 ms.

Schéma électronique



Mesure : sortie T du four $\rightarrow$ pont $1/100$ $\rightarrow$ 10 mV/°C $\rightarrow$ AN0.

$$\theta = N \cdot \frac{500}{1023} \approx 0{,}489\, N \text{ °C}$$

résolution $\approx 0{,}49$ °C ; moyenne de 4 conversions.

Vérifier l'échelle réelle de la sortie T (TP 2).

Action : PWM CCP1 (RC2) + Timer2 $\rightarrow$ MOSFET logique.

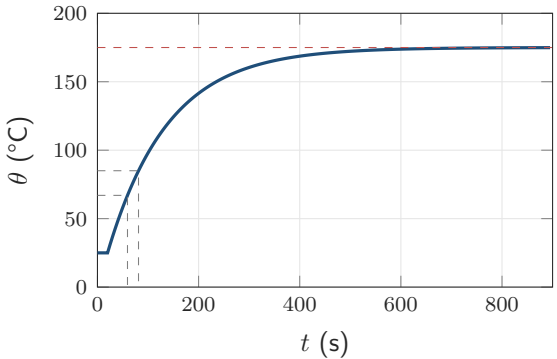
$$f = \frac{64 \text{ MHz}}{4 \times 16 \times 250} = 4 \text{ kHz}$$

résolution $4(PR2+1) = 1000$ pas : `set_pwm1_duty((int16)(u*10))`.

Prédiviseur Timer2 du K22 : 1, 4 ou 16 seulement ($f_{\min} \approx 3{,}9$ kHz).

Identification et commande

Identification : méthode de Broïda



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

Échelon u : 0 → 50 %.

- $K = \Delta\theta_\infty / \Delta u = 150/50 = 3$ °C/%
- $\tau = 5{,}5\,(t_{40} - t_{28}) \approx 120$ s
- $\theta_r = 2{,}8\,t_{28} - 1{,}8\,t_{40} \approx 20$ s

$t_{28} \approx 59{,}4$ s (67 °C), $t_{40} \approx 81{,}2$ s (85 °C) ; outil : `scripts/identif_broida.py`.

Réglage du PID et période d'échantillonnage

Méthode	K_p	T_i	T_d
Ziegler-Nichols (BO)	$1{,}2\,\tau/(K\theta_r) = 2{,}4$	$2\theta_r = 40$ s	$0{,}5\theta_r = 10$ s
Chien-Hrones-Reswick 0 %	$0{,}6\,\tau/(K\theta_r) = 1{,}2$	$\tau = 120$ s	$0{,}5\theta_r = 10$ s

Choix de T_e

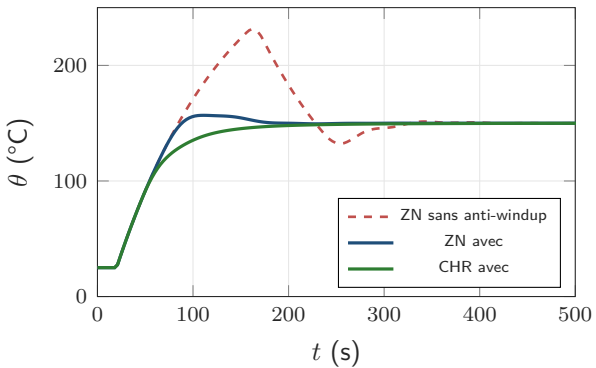
$T_e \leq \tau/10$ et $T_e \leq \theta_r/4 \Rightarrow T_e \leq 5$ s. On prend $T_e = 0{,}5$ **s** = période de la tâche PID.

$$I_k = I_{k-1} + \frac{K_p T_e}{T_i} e_k$$

$$D_k = \frac{T_d}{T_d + N T_e} D_{k-1} - \frac{K_p T_d N}{T_d + N T_e} (\theta_k - \theta_{k-1})$$

$$u_k = \mathrm{sat}_{[0,100]}(K_p e_k + I_k + D_k)$$

L'anti-windup est indispensable



Réglage	D_1	$t_{5\%}$
ZN sans AW	65 %	284 s
ZN avec AW	5,5 %	132 s
CHR avec AW	0 %	137 s

Windup

Commande saturée à 100 % au démarrage : sans protection, l'intégrale « gonfle » et provoque un fort dépassement. **Intégration conditionnelle**.

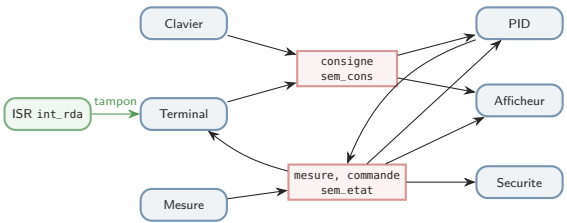
Conception multitâche

Découpage en tâches et charge

Tâche	Rôle	T_i	C_i	C_i/T_i
`Live`	LED de vie RB7	200 ms	10 µs	0,005 %
`Clavier`	fronts RB0/RB1 $\rightarrow$ consigne	50 ms	40 µs	0,08 %
`Mesure`	4 conversions, moyenne, °C	100 ms	200 µs	0,2 %
`PID`	loi de commande, PWM	500 ms	900 µs	0,18 %
`Securite`	$T > T_{\max} \Rightarrow$ coupure	100 ms	20 µs	0,02 %
`Afficheur`	*une* ligne du LCD	250 ms	2,5 ms	1 %
`Terminal`	commandes + *un* caractère	10 ms	60 µs	0,6 %
			charge U	$\approx 2{,}1$ %

- `minor_cycle = 10ms` (périodes multiples) ; $H = 1$ s.
- Pire cycle (toutes échues à $t = k$ s) : $\sum C_i = 3{,}73$ ms < 10 ms $\Rightarrow$ gigue $\leq 3{,}7$ ms.
- Sécurité : 100 + 100 + gigue ≈ 204 ms > 200 ms $\rightarrow$ `Securite` à 50 ms ou lecture directe du CAN.

Communication et synchronisation



- `consigne` : 2 écrivains → `sem_cons`.
- `mesure`, `commande` : 4 lecteurs → `sem_etat`.
- ISR → `Terminal` : tampon circulaire, un seul écrivain par index d'un octet (atomique) : pas de masquage.
- `Securite` coupe la PWM et `rtos_disable(PID)`.

La tâche PID

```
#task(rate=500ms, max=2ms)
void PID()
{
    static float I = 0.0, D = 0.0, y_prec = 25.0;
    float w, y, e, v, u;
    rtos_wait(sem_cons);  w = consigne;  rtos_signal(sem_cons);
    rtos_wait(sem_etat);  y = mesure;    rtos_signal(sem_etat);
    e = w - y;
    D = (TD / (TD + NF * TE)) * D - (KP * TD * NF / (TD + NF * TE)) * (y - y_prec);
    v = KP * e + I + D;
    u = v;
    if (u > 100.0) u = 100.0;
    if (u < 0.0)   u = 0.0;
    // anti-windup : intégration conditionnelle
    if ((u == v) || (u >= 100.0 && e < 0.0) || (u <= 0.0 && e > 0.0))
        I += KP * TE / TI * e;
    y_prec = y;
    set_pwm1_duty((int16)(u * 10.0));
    rtos_wait(sem_etat);  commande = u;  rtos_signal(sem_etat);
}
```

Ne jamais bloquer : réception sous interruption, émission par caractère

```
// ---- réception UART par
     interruption : tampon circulaire
     ISR -> Terminal
#define RXN 8
char  rx_buf[RXN];
int8  rx_w = 0, rx_r = 0;

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

À 9600 bauds, un caractère toutes les 1,04 ms : une scrutation à 10 ms en perdrait.

```
    // 2) émission d'un seul caractère par
     activation (1,04 ms à 9600 bauds)
    if (tx_i < tx_n) {
        putc(tx_buf[tx_i++]);
    } else if (++compteur >= 100) {
        compteur = 0;
        rtos_wait(sem_etat);  y = mesure;  u =
     commande;  rtos_signal(sem_etat);
        rtos_wait(sem_cons);  w = consigne;
     rtos_signal(sem_cons);
        sprintf(tx_buf, "W=%5.1f T=%5.1f U=%3.0f\r\n
     ", w, y, u);
        tx_n = strlen(tx_buf);
        tx_i = 0;
    }
```

Une trame de 23 caractères = 24 ms > cycle mineur : on émet **un caractère par activation** (10 ms).

Validation et mise en pratique

Validation

- **Temporelle** : analyseur logique, `rtos_stats`, `rtos_overrun`.
- **Fonctionnelle** : boutons, S, R, LCD, trames.
- **Performances** : échelon de consigne vs simulation.
- **Sécurité** : coupure ≤ 200 ms, réarmement.

TP (séances 7 à 9)

- TP 1 : RTOS de CCS, mesure des tâches.
- TP 2 : schéma, boucle ouverte, identification, réglage.
- TP 3 : régulation multitâche, sécurité, validation.
- Rapport + démonstration Proteus.

À retenir

À retenir

- Démarche : spécifier, architecturer, modéliser, commander, découper en tâches, valider, coder, tester.
- Procédé thermique $\approx 1^{\text{er}}$ ordre $+$ retard ; Broïda ; ZN ou CHR ; $T_e \leq \tau/10$; **anti-windup**.
- RTOS coopératif : tâches courtes, opérations lentes découpées (un caractère, une ligne), entrées rapides sous interruption, $\sum C_i <$ `minor_cycle`.
- Sécurité : chemin de détection le plus court, coupure directe de l'actionneur.