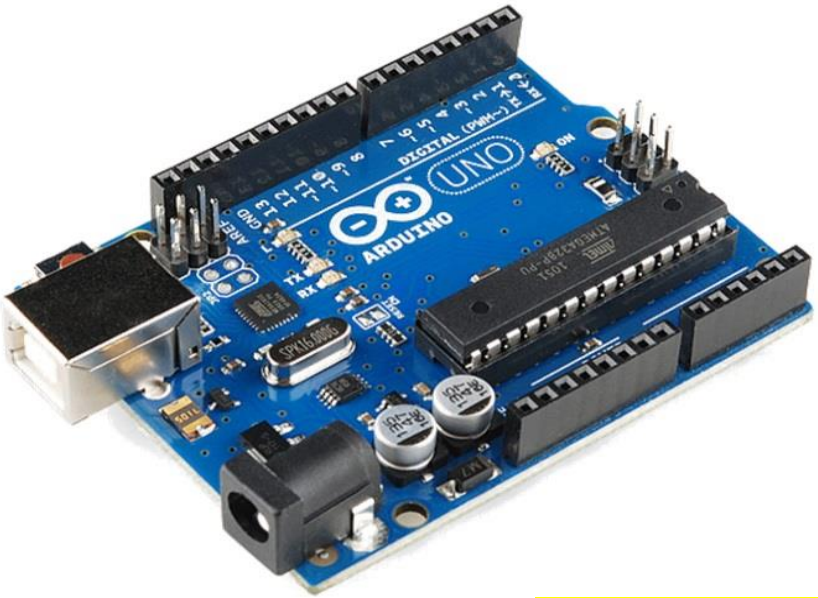
Arduino Real Time Operating System

Introduction aux systèmes temps réel



Clin d'œil à ROLF qui a présenté ce sujet en 2014
https://microclub.ch/wp-content/uploads/2014/05/Multit%C3%A2che-Systeme-Embarqu%C3%A9-pdf1.pdf

Cet exposé ne se prétend pas, ni ne veut, être un document exhaustif traitant d'un sujet gigantesque. Mais une incitation, un survol, à découvrir ou redécouvrir les systèmes en temps réel idoines…

Plan de l'exposé

Qu'est-ce que le temps réel ?
Exemples d'application temps réel
Interactions scrutations et interruptions
Exemple concret et vécu
RTOS avec Arduino (FreeRTOS)
Exemples pratiques avec Arduino Uno

Qu’est-ce que le temps réel ?

En informatique industrielle, on parle d'un **système temps réel** lorsque ce système informatique contrôle (ou pilote) un procédé physique à une vitesse adaptée à l'évolution du procédé, du processus, contrôlé.

De façon informelle, on définit un système temps réel comme un système dont le comportement dépend, non seulement de l’exactitude des traitements effectuées, mais également du temps où les résultats de ces traitements sont produits. En d’autres termes, un retard (le fait de dépasser une échéance) est considéré comme une erreur qui peut entraîner de graves conséquences.

Un système temps réel n’est pas un système « qui va vite / rapide» mais un système qui satisfait des contraintes temporelles (les contraintes de temps dépendent de l’application et de l’environnement alors que la rapidité dépend de la technologie utilisée, celle du processeur par exemple).

Une réponse, même bonne, si elle arrive trop tard est une mauvais réponse !

Exemples d'application

Exemple 1: domaine de l'avionique
Dans un Airbus A340 :

Il y a 115 équipements avec :
- 3 calculateurs qui élaborent les paramètres inertiels
- 2 calculateurs qui implémentent les lois de guidage
- 5 calculateurs qui implémentent les lois de pilotage
- 2 calculateurs d'alarmes, etc.
- Environ 200 000 données sont échangées

Exemple 2: multimédia sur le Web

Système temps réel souple:

-Contraintes temporelles: gigue, délais de bout en bout, temps de réponse, etc.

-Synchronisations: intra et inter-flux

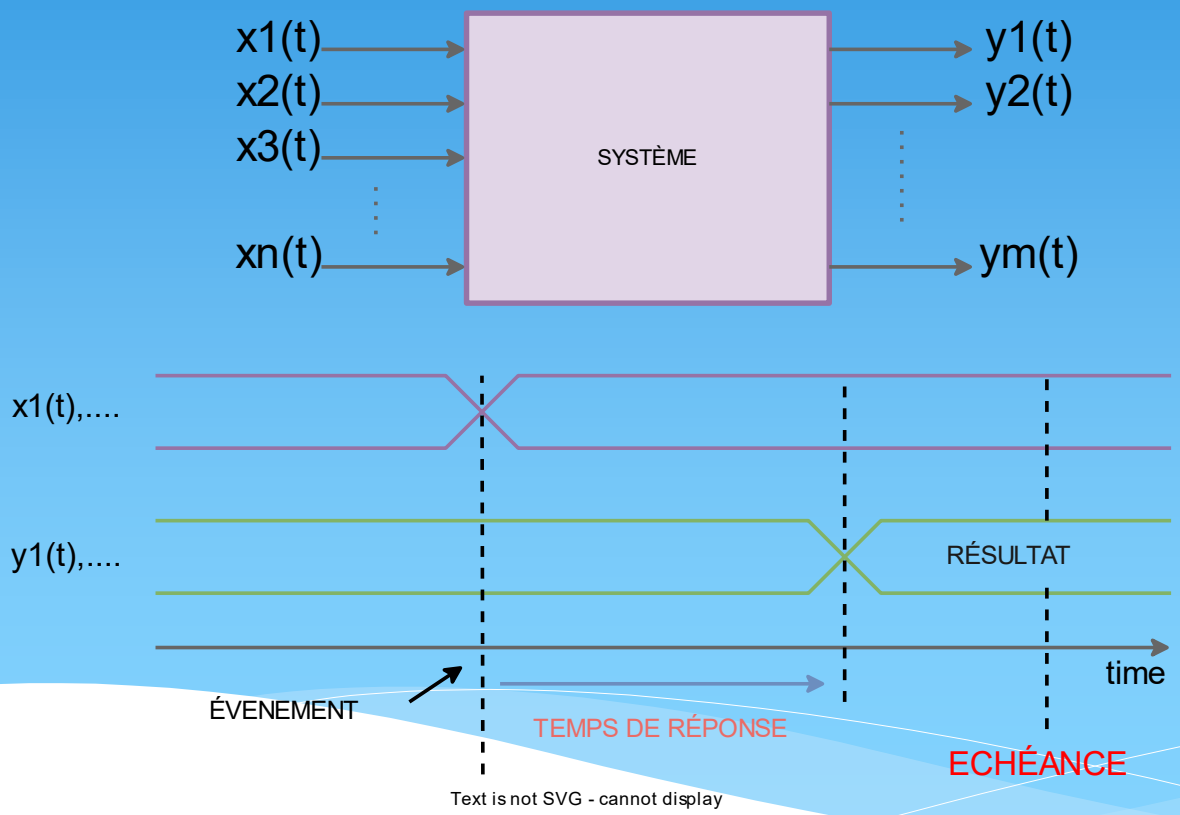
-Plateforme généraliste (ex. PC + Windows)

-Application interactive

-Débits variables et difficiles à estimer hors ligne.

Un système est qualifié de temps-réel lorsque son exactitude logique est conditionnée par :

- L'exactitude temporelle (établissement des sorties)
- L'exactitude des résultats (valeur des sorties)



Question : que se passe-t-il si le temps de réponse dépasse l'échéance ?
cela dépend de la classification de la contrainte temps réel.

Classification des systèmes temps-réel
Le degré de tolérance au non respect de l'échéance caractérise le système :

- **Hard Real Time** : la réponse du système dans le temps imparti est vitale.
  L'absence de réponse est catastrophique et entraîne la faute du système.
  Exemples : système de conduite de missiles, airbag d'une voiture, …

- **Firm Real Time** : quelques échéances manquées sporadiquement sont tolérées.
  Exemple : décodage stream vidéo.

- **Soft Real Time** : un degré de tolérance concernant le respect des échéances est admis.
  La réponse du système après les délais réduit progressivement son intérêt.
  Exemple : mise à jour des places disponibles sur système de réservation de train.

Imaginons le cas le plus compliqué où un évènement peut survenir à n'importe quel moment et doit être pris en compte dans un délai maximum donné; nous rentrons de plein fouet dans la logique des systèmes temps réel. Le programme doit être à même de réagir aux évènements à des instants qu'il ne maitrise pas et durant lesquels il peut être en train d'effectuer une autre tâche.

Comment procéder ?

Deux modèles d'interaction avec le monde extérieur:
<<la scrutation cyclique (polling) et <<l'interaction par interruptions>>

Interaction par scrutation cyclique

Une première solution pour permettre à un programme de prendre en compte des événements extérieurs, est de le faire interroger (ou scruter) ses périphériques. Ceci est appelé <<interaction par scrutation cyclique>>.
Un système mettant en œuvre de type d'interaction se présente comme une boucle infinie. Par exemple, dans le cas d'un système recevant des données issues de capteurs et produisant des commandes idoines en retour.

Avantages:

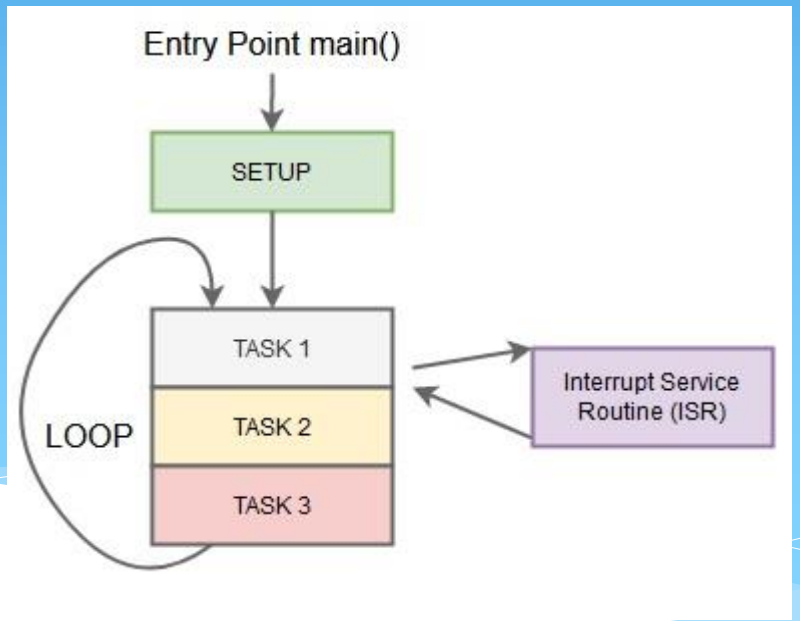
Le premier avantage est que le temps de réaction est simple à déterminer: il s'agit simplement du temps de la boucle. On peut donc savoir immédiatement si le système satisfait aux contraintes temporelles. Par ailleurs la programmation est simple, dans le cas d'une application gérant peu de périphériques.

Inconvénients:

Lorsque le nombre de périphériques est important, ou qu'ils ont besoin de temps de traitement trop différents, la programmation devient rapidement un casse-tête. Il devient alors nécessaire de rajouter à la boucle gérant les périphériques lents, des sous-boucles de traitements des périphériques rapides…

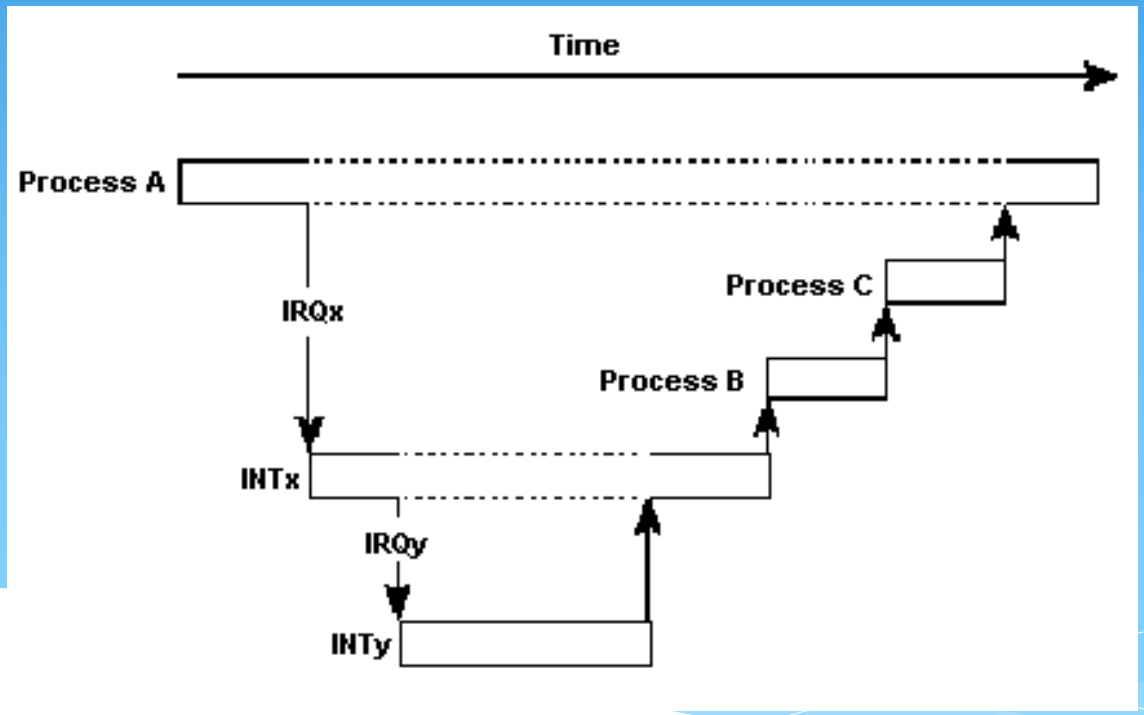
Interaction par interruptions

La seconde solution réside dans le fait que le processeur puisse être averti de la survenue d'un évènement. Il est alors nécessaire de mettre en œuvre un mécanisme qui interrompe le cours normal de l'exécution du programme pour l'avertir dudit événement, et qui permette de reprendre le programme là où il a été interrompu.

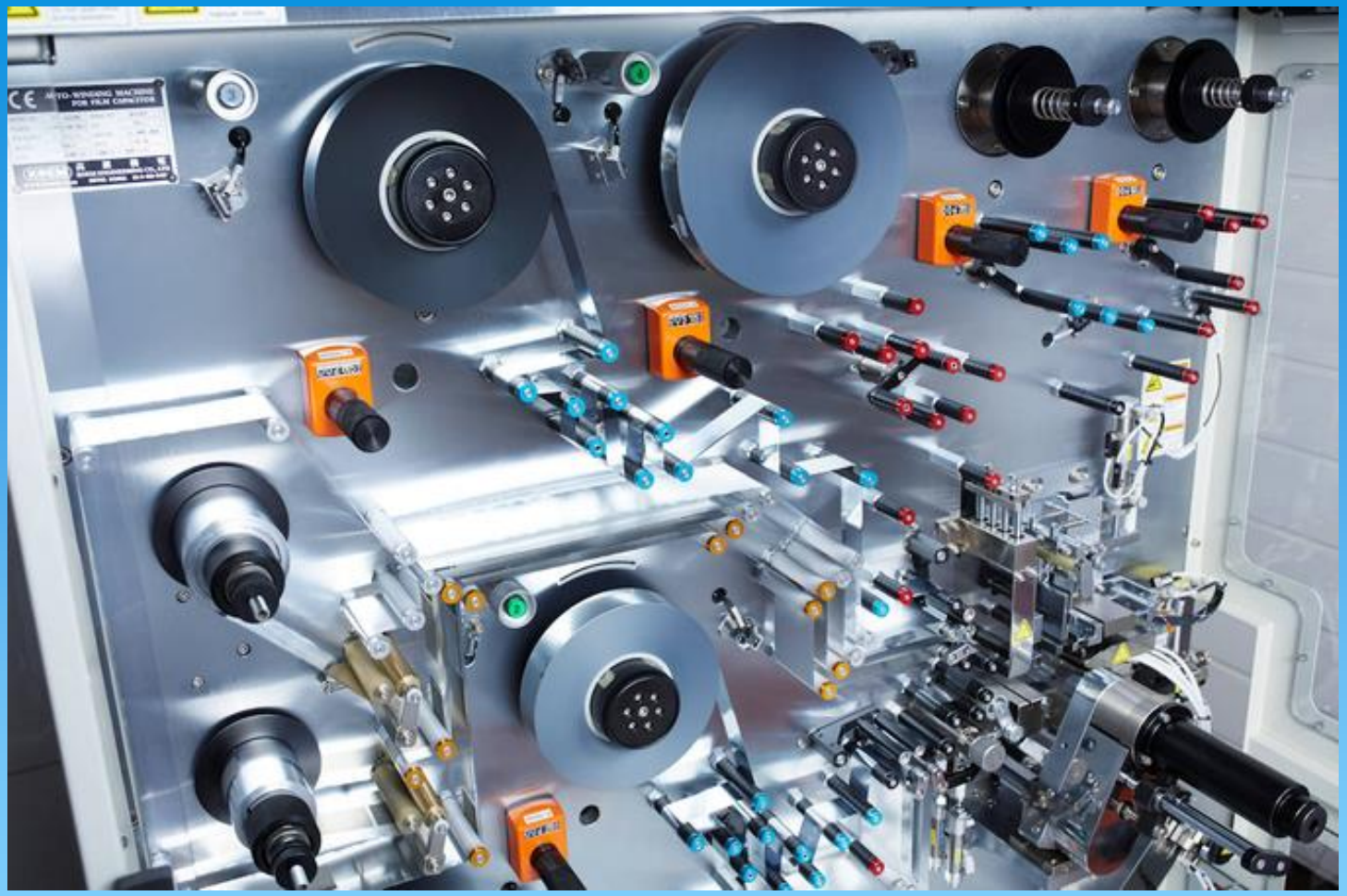


1 ms, 10 ms et 100 ms, par exemple

Lors de l'arrivée d'une interruption, le processeur doit arrêter l'exécution de la tâche courante. Afin d'être en mesure de reprendre ultérieurement la tâche interrompue, il doit sauvegarder son état. Cet état contient, en particulier, la valeur du compteur ordinal (program counter) qui indique la prochaine tâche à exécuter. Une fois ceci effectué, le processeur identifie une séquence spéciale d'instructions appelée «gestion d'interruptions» (interrupt handler), qui doit être exécutée suite à l'arrivée de l'interruption. Ces interruptions peuvent être générées extérieurement par un timer. Par exemple toutes les 1 ms, les 10 ms et les 100 ms.



Exemple vécu…
Pratique d'utilisation des interruptions
Bobineuse de condensateurs



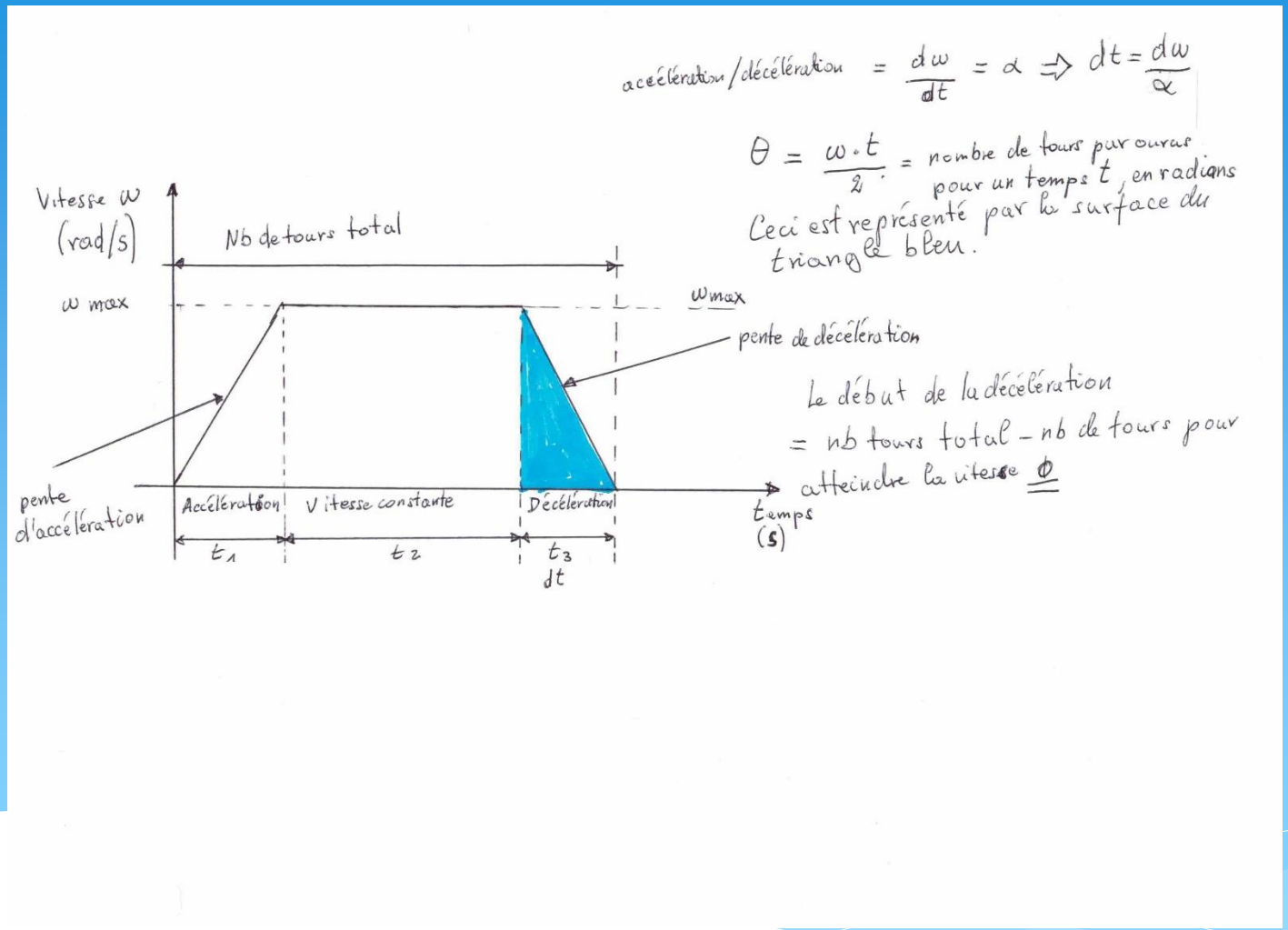
La qualité d'un condensateur bobiné est intimement liée à l'opération de bobinage. Lors de laquelle aucun à-coup, dû à une différence de vitesse ou d'accélération ou décélération, ne doit se produire.

Les pentes d'accélération et décélération doivent être constantes ! Le nombre de tours bobinés doit être également constant, dans une série, et ceci au centième de tour près pour chaque condensateur.

Si la pente d'accélération ($d\omega/dt$) est facile à maitriser, il en va tout autrement pour la décélération.

En effet, il est important, voire vital, de connaitre précisément le moment où l'on devra commencer à décélérer avec la pente fixe, choisie. Vital, l'est également, le nombre de tours, au centième près, nécessaire pour que le moteur passe de sa vitesse initiale à l'arrêt complet.

Allures du bobinage



accélération/décélération $= \frac{d\omega}{dt} = \alpha \Rightarrow dt = \frac{d\omega}{\alpha}$

$\theta = \frac{\omega \cdot t}{2}$ = nombre de tours parcourus pour un temps t en radians

Ceci est représenté par la surface du triangle bleu.

Le début de la décélération = nb tours total – nb de tours pour atteindre la vitesse Φ

```
;-> PROCEDURE DE SOUSTRACTION 6 DIGITS EN
BCD
;       BHL - CDE = BHL
;           ACDE non affecte

SOUBCD:     @ENTR TYP=E,SEG=C
            PUSH        PSW
            PUSH        D
            PUSH        B
            MVI         A, 9AH
            SUB         E
            ADI         0
            DAA
            MOV         E, A
            MVI         A, 99H
            ACI         0
            SUB         D
            ADI         0
            DAA
            MOV         D, A
            MVI         A, 99H
```

Outils mathématiques primaires à disposition d'où impossibilité de faire des calculs compliqués en temps réel !!!

Eléments à disposition

Une carte compteur (32 bits) détermine le nombre de tours effectués (au centième de tour)
Les pentes d'accélération et de décélération. La vitesse du moteur est donnée par une consigne issue d'un D/A 8 bits (255 valeurs) qui représente un nombre de radians/sec.

Comment faire alors ?

Le mouvement circulaire uniformément accéléré (MCUA) nous permet de calculer précisément (toujours au centième de tour) le nombre de tours nécessaires pour arrêter le moteur.

Et alors ?

De ce qui précède, nous connaissons la vitesse instantanée du moteur du moment que nous connaissons sa consigne (sous-entendu que les asservissements en vitesse et courant soit adéquatement dimensionnés). Partant, nous connaissons en temps réel le nombre de tours nécessaires pour arrêter le moteur. Donc il existe un couple de valeurs vitesse/nombre de tours pour arrêt du moteur.

Donc, nous créons une table (array) à deux dimensions, une pour la consigne vitesse (0-255) et l'autre pour le nombre de tours pour s'arrêter.

Ainsi, avec une périodicité de 1 ms, on compare le nombre de tours restant à bobiner avec celui du nombre de tours nécessaires pour s'arrêter (avec la pente de décélération choisie) et, si le nombre de tours restant est égal ou inférieur, on décrémente la consigne d'une unité.

La périodicité de 1ms (RST 6 du 8085) résulte de la programmation de l'UART (8253). Ce dernier activant toutes les 1 ms l'interruption idoine du 8085.

Exemple de tables de décélération pour une pente de 400 rad/ S²

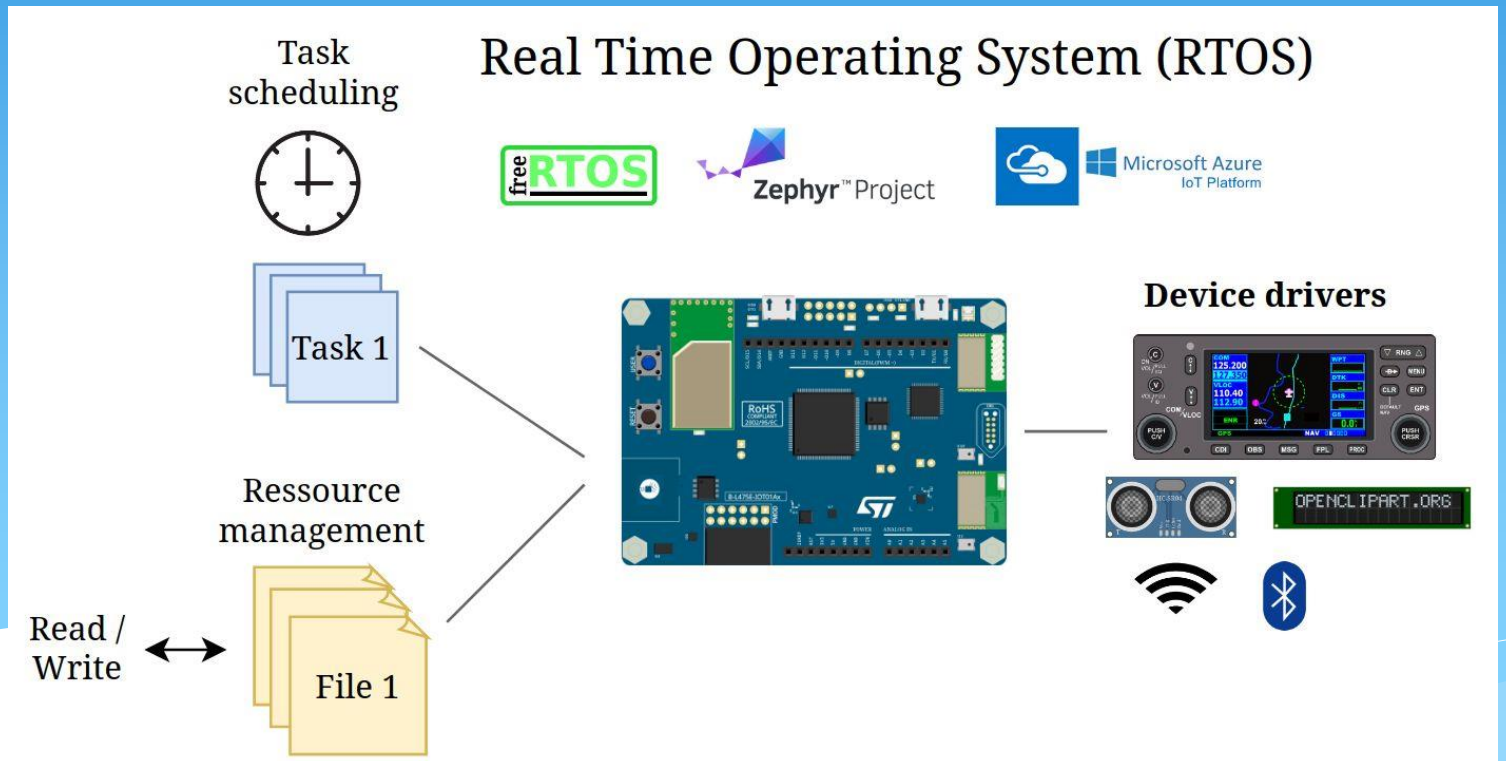
```
;-> TABLE DE PILOTAGE (DIXIEME ET CENTIEME)

TABL1:  @ENTR TYP=E

;=> TABLE DE DECELERATION DE :   400   RAD/S2

     DB 00H,00H,01H,02H,05H,08H,11H,15H
     DB 20H,26H,32H,38H,46H,54H,63H,72H
     DB 82H,93H,04H,16H,28H,42H,55H,70H
     DB 85H,01H,17H,34H,52H,70H,89H,09H
     DB 29H,50H,72H,94H,17H,40H,65H,89H
     DB 15H,41H,68H,95H,23H,52H,81H,11H
     DB 42H,73H,05H,37H,70H,04H,39H,74H
     DB 10H,46H,83H,21H,59H,98H,38H,78H
     DB 02H,27H,52H,78H,04H,31H,57H,85H
     DB 12H,40H,69H,98H,27H,56H,86H,17H
     DB 47H,78H,10H,42H,74H,07H,40H,73H
     DB 07H,41H,76H,11H,46H,82H,18H,55H
     DB 92H,29H,67H,05H,43H,82H,21H,61H
     DB 01H,41H,82H,16H,51H,86H,21H,57H
     DB 93H,29H,65H,02H,40H,77H,15H,53H
     DB 92H,30H,69H,09H,49H,89H,29H,70H
     DB 11H,52H,94H,36H,79H,21H,64H,08H
     DB 51H,95H,39H,84H,29H,74H,20H,66H
     DB 12H,58H,05H,53H,00H,36H,72H,08H
     DB 45H,82H,19H,56H,94H,32H,70H,08H
     DB 47H,85H,24H,64H,03H,43H,83H,23H
     DB 64H,04H,45H,86H,28H,69H,11H,54H
     DB 96H,39H,81H,25H,68H,12H,55H,99H
     DB 44H,88H,33H,78H,23H,69H,15H,61H
     DB 07H,54H,00H,47H,95H,42H,90H,38H
     DB 86H,34H,83H,32H,81H,30H,80H,30H
     DB 80H,30H,81H,32H,83H,34H,86H,38H
     DB 90H,42H,95H,47H,00H,54H,07H,61H
     DB 15H,69H,24H,78H,33H,89H,44H,00H
     DB 56H,12H,68H,25H,82H,39H,96H,54H
     DB 12H,70H,28H,87H,45H,05H,64H,23H
     DB 83H,43H,04H,64H,25H,86H,47H,08H

;-> TABLE DE PILOTAGE (DIZAINE ET UNITE

     DB 00H,00H,00H,00H,00H,00H,00H,00H
     DB 00H,00H,00H,00H,00H,00H,00H,00H
     DB 00H,00H,01H,01H,01H,01H,01H,01H
     DB 01H,02H,02H,02H,02H,02H,02H,03H
     DB 03H,03H,03H,03H,04H,04H,04H,04H
     DB 05H,05H,05H,05H,06H,06H,06H,07H
     DB 07H,07H,08H,08H,08H,09H,09H,09H
     DB 10H,10H,10H,11H,11H,11H,12H,12H
     DB 13H,13H,13H,13H,14H,14H,14H,14H
     DB 15H,15H,15H,15H,16H,16H,16H,17H
     DB 17H,17H,18H,18H,18H,19H,19H,19H
     DB 20H,20H,20H,21H,21H,21H,22H,22H
     DB 22H,23H,23H,24H,24H,24H,25H,25H
     DB 26H,26H,26H,27H,27H,27H,28H,28H
     DB 28H,29H,29H,30H,30H,30H,31H,31H
     DB 31H,32H,32H,33H,33H,33H,34H,34H
     DB 35H,35H,35H,36H,36H,37H,37H,38H
     DB 38H,38H,39H,39H,40H,40H,41H,41H
     DB 42H,42H,43H,43H,44H,44H,44H,45H
     DB 45H,45H,46H,46H,46H,47H,47H,48H
     DB 48H,48H,49H,49H,50H,50H,50H,51H
     DB 51H,52H,52H,52H,53H,53H,54H,54H
     DB 54H,55H,55H,56H,56H,57H,57H,57H
     DB 58H,58H,59H,59H,60H,60H,61H,61H
     DB 62H,62H,63H,63H,63H,64H,64H,65H
     DB 65H,66H,66H,67H,67H,68H,68H,69H
     DB 69H,70H,70H,71H,71H,72H,72H,73H
     DB 73H,74H,74H,75H,76H,76H,77H,77H
     DB 78H,78H,79H,79H,80H,80H,81H,82H
     DB 82H,83H,83H,84H,84H,85H,85H,86H
     DB 87H,87H,88H,88H,89H,90H,90H,91H
     DB 91H,92H,93H,93H,94H,94H,95H,96H
```

```
; Fichier    COSMOS.STM
02-02-84
;
heure: 10.10

;-> TIMER 1MS PERMANENT (RST 6)

REAL1:   @ENTR TYP=E,SAVE=ALL
     DI
     MVI A,40H
     OUT 0E5H
     MVI A,1
     OUT 0EDH

;-> MOTEUR M1
;-> COMPARAISON POUR DECREMENTATION

     @IF H,GT,L
     @OR H,EQ,L
     @AND B,GT,D
     @OR H,EQ,L
     @AND B,EQ,D
     @AND C,GE,E
     @THEN
     @IF UDA1,GT,2
     @THEN
     CHARGE FM1,4
     DECR UDA1
     MOTEUR 1,'M'
     @ELSE
     MOTEUR 1,0
     @BEND IF
     @BEND IF

     @BEND CASE

     @BEND IF

     EI
     @EXIT
     @END
```

Partie relative à la procédure de décrémentation de la consigne (UDA1). Le 8085 possède 2 registre 16bits H-L et D-E

Real Time Operating System (RTOS)

Un système d'exploitation en temps réel (RTOS) est un système d'exploitation qui permet la prévisibilité, le déterminisme du temps d'exécution d'une tâche plutôt que l'optimisation globale comme pour un GPOS. Un RTOS autorise la préemption d'une tâche en cours pour exécuter une tâche de niveau de priorité plus élevé.



Il existe de nombreux RTOS, certains sont certifiés pour des applications critiques (aviation, médical), d'autres spécialisés pour l'IoT, …
Open Source : FreeRTOS, Zephyr Project, …
Payant : Azure RTOS, VxWorks, QNX, …

Real Time Operating System

Le RTOS est un logiciel système qui fournit des services et gère les ressources du processeur pour les applications. Ces ressources incluent :les cycles du processeur, la mémoire, les périphériques et les interruptions.

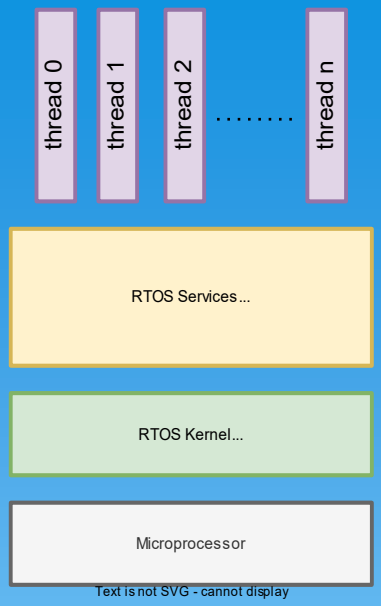
Le RTOS doit être capable d'assurer un fonctionnement en continu pendant des mois ou des années. Il dispose une interface permettant le développement d'applications (API : Application Programming Interface), doit posséder une faible empreinte mémoire et favoriser la portabilité du code entre différentes architectures de processeurs.

Le but principal d'un RTOS est d'allouer le temps de traitement entre diverses tâches que le logiciel embarqué doit effectuer. Cela implique généralement une division du programme en morceaux, communément appelés « tasks » (tâches) ou « threads ».

Le RTOS contrôle l'exécution des threads et la gestion associée de chaque thread : le contexte. Chaque thread se voit attribuer une priorité, pour contrôler quel thread doit s'exécuter si plus d'un est prêt à s'exécuter (c'est-à-dire : non bloqué).

Lorsqu'un thread de priorité supérieure (par rapport au thread en cours d'exécution) doit s'exécuter, le RTOS enregistre le contexte du thread en cours d'exécution dans la mémoire et restaure le contexte du nouveau thread. Le processus d'échange de contexte de threads est appelé commutation de contexte.

Les push et le pop… ça ne vous rappelle rien ?



Il est important de noter qu'un RTOS doit offrir une préemption. La préemption est l'action de passer instantanément et de manière transparente à un thread de priorité supérieure, sans avoir à attendre l'achèvement du thread de priorité inférieure. En plus de l'allocation du processeur, un bon RTOS fournit une communication supplémentaire, une synchronisation, et les services d'allocation de mémoire.
Les opérations d'un RTOS sont effectuées par le **KERNEL** (noyau)

Éléments de base du Kernel

Ordonnanceur

Elément principal du noyau. Implémente l'algorithme d'ordonnancement qui détermine quelle tâche obtient les ressources du processeur. On parle en anglais de Scheduler.

Objets du noyau

Accessibles au programmeur pour le développement d'applications. Exemple d'objets : tâche, mutex, sémaphore (objet de synchronisation), file de messages.

Services

Opérations effectuées par le noyau ; par exemple : gestion de la mémoire, traitement des interruptions, gestion du temps (cycles, délais, etc.)

L'ordonnanceur

Assure l'exécution d'entités ordonnançables
Par définition, les entités ordonnançables sont concurrentes : elles entrent en compétition pour obtenir du temps processeur.

Une tâche est une entité ordonnançable
La faculté, pour un noyau, de pouvoir gérer plusieurs tâches devant s'exécuter simultanément est exprimée par le terme multitâches.

La simultanéité d'exécution (sur un système monoprocesseur) n'est qu'apparente.
L'ordonnanceur donne la main à chaque tâche de façon séquentielle.

L'ordonnanceur doit exécuter la bonne tâche au bon moment
L'ordonnanceur détermine quelle tâche doit s'arrêter afin de donner les ressources processeur à une autre tâche.

Rappels ou découvertes:

Ces notions son très importantes dans FreeeRTOS

Déterminisme :

Un **système déterministe** est un système qui réagit toujours de la même façon à un événement, c'est-à-dire que, quoi qu'il se soit passé auparavant, à partir du moment où le système arrive dans un état donné, son évolution sera toujours identique.

Préemption :

Exemple: Deux tâches tournent sur un système. Si la tâche 2 peut prendre la main, car elle est prioritaire, on dit qu'elle préempte la tâche 1.

Ordonnanceur

L'Ordonnanceur est le composant du noyau responsable de la distribution du temps CPU entre les différentes tâches. Il est basé sur le principe de la priorité : chaque tâche possède une priorité dépendant de son importance (plus une tâche est importante, plus sa priorité est élevée). Il attribue le processeur à la tâche exécutable (non dormante et non bloquée) de plus haute priorité.

Sémaphore

Le concept de sémaphore est utilisé pour contrôler l'accès à une ressource. Lorsqu'une tâche accède à une ressource non partageable, le sémaphore à l'entrée de celle-ci devient bloqué et le reste tant que la tâche n'a pas relâché la ressource. Il empêche ainsi tout autre tâche à accéder à cette ressource.

Dans le trafic ferroviaire, il ne doit y avoir qu'un train au maximum par tronçon de voie ferrée. Lorsqu'un train entre dans ce tronçon, aucun autre train de peut y entrer tant que le premier train ne l'a pas quitté.

Mutex

Un **sémaphore** qui ne peut prendre que deux états (libre et bloqué) est appelé sémaphore binaire ou mutex. Dans ce cas, une seule tache peut avoir accès à la ressource.

Les files de communication (Queues)

Les files (*queue*) permettent de faire communiquer et synchroniser les tâches entre elles. Une *queue* (file) peut contenir un nombre fini d'éléments de données de taille fixe. La longueur (nombre maximal d'éléments) et la taille de chaque élément de données sont définies lors de la création de la file.

Les files sont normalement utilisées en tant que tampons FIFO (*First In First Out*) où les données sont écrites à la fin (*tail*) de la file et supprimées par l'avant (*head*) de la file.

Les files sont des objets auxquels peuvent accéder n'importe quelle tâche ou fonction d'interruption (ISR) qui connaît leur existence.

La commutation de contexte

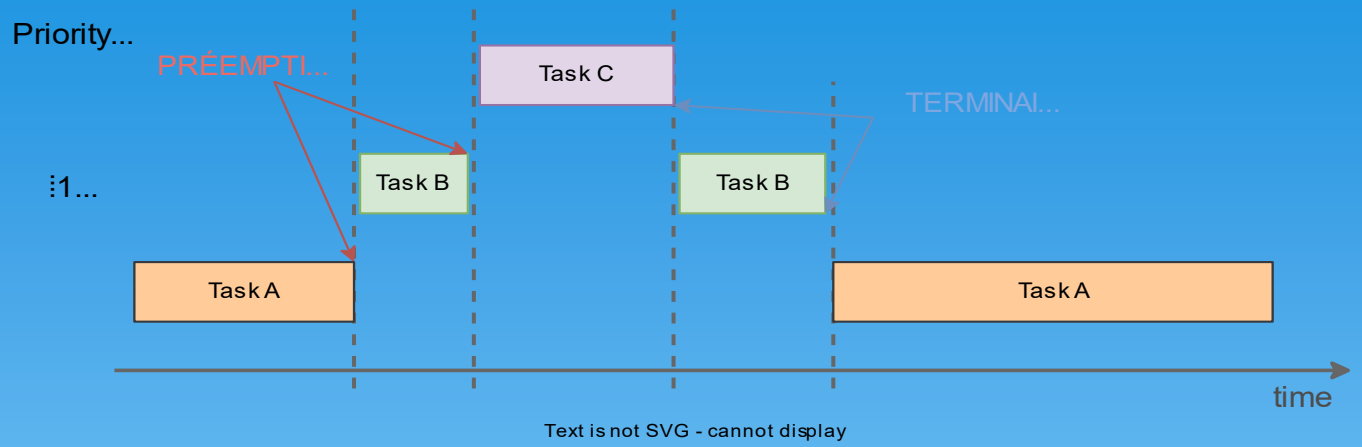
Le contexte de la tâche arrêtée doit être sauvegardé, celui de la tâche devant s'exécuter doit être restauré.

Le contexte inclut généralement ces informations :
Le contenu des registres du processeur,
La valeur du compteur de programme et du pointeur de pile.

Le module de l'ordonnanceur assurant la commutation du contexte est le répartiteur (dispatcher)

La commutation de contexte, si elle se produit trop fréquemment, peut consommer une part non négligeable du temps processeur.
Il existe différentes stratégies d'ordonnancement :
dont le préemptif avec priorités.

Ordonnancement préemptif avec priorités



Un niveau de priorité est affecté à chaque tâche

•Suivant le type d'ordonnanceur, la priorité peut être attribuée par le développeur (priorité statique) ou par l'ordonnanceur en cours d'exécution (priorité dynamique)

Plusieurs tâches peuvent avoir la même priorité

La tâche de plus haute priorité obtient toujours le temps processeur

•Lorsqu'une tâche de plus haute priorité est prête à s'exécuter, la tâche en cours d'exécution est préemptée :

•L'exécution de la tâche est interrompue puis l'ordonnanceur donne la main à une autre tâche par commutation de contexte

Une tâche ne peut pas être préemptée par une tâche de priorité identique ou inférieure

Exemple de RTOS: FreeRTOS

FreeRTOS Kernel Quick Start Guide
https://www.freertos.org/FreeRTOS-quick-start-guide.html

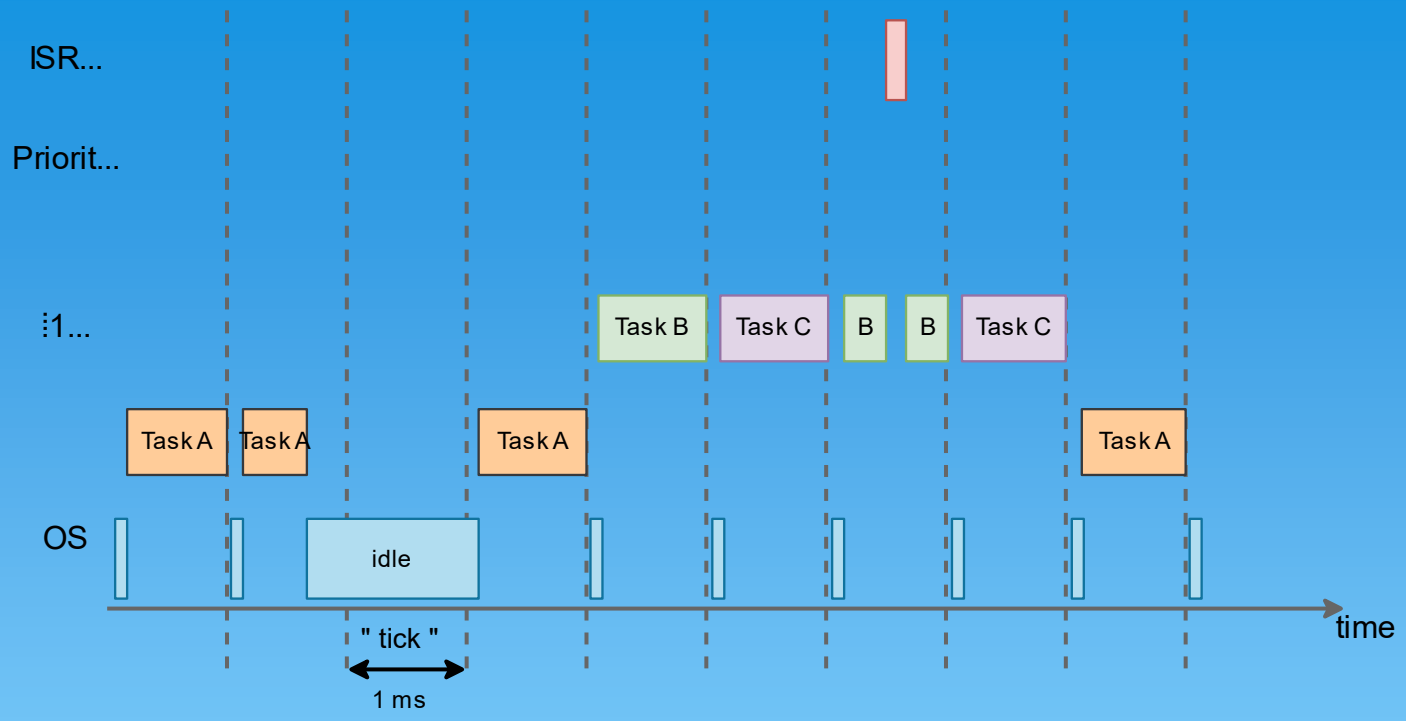
Caractéristiques de FreeRTOS

FreeRTOS implémente les composants noyau de base :
- **Ordonnanceur / répartiteur**
- **Objets noyaux : tâches, objets de synchronisation et de communication**
- **Services : gestion des interruptions et de la mémoire**
- **API spécifique, non standardisée**

FreeRTOS utilise un ordonnancement « round-robin » avec priorités
- **Les priorités sont définies de manière statique au moment de la conception**
- **Le nombre de niveaux de priorités supportés est configurable**
- **La valeur du quantum (Tick) est configurable**

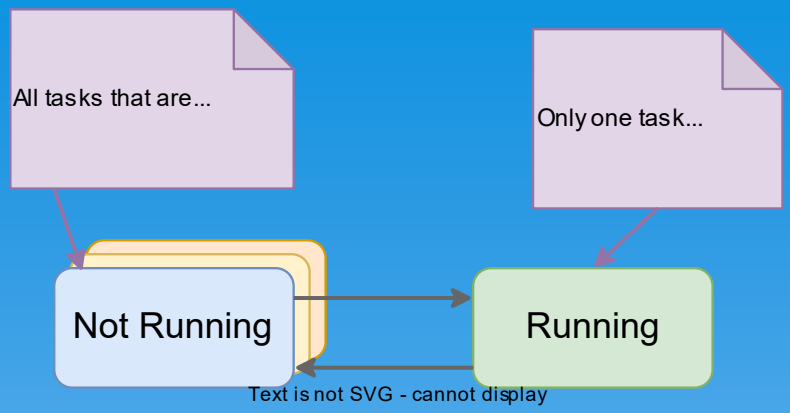
FreeRTOS est un système d'exploitation temps réel (RTOS) pour les microcontrôleurs. Développé en partenariat avec les principaux fabricants de puces au monde sur une période de 15 ans, et maintenant téléchargé toutes les 2 minutes, FreeRTOS est un système d'exploitation en temps réel (RTOS) leader du marché des microcontrôleurs et des petits microprocesseurs. Distribué gratuitement sous la licence open source du MIT, FreeRTOS comprend un noyau et un ensemble croissant de bibliothèques IoT adaptées à une utilisation dans tous les secteurs industriels. FreeRTOS est construit en mettant l'accent sur la fiabilité et la facilité d'utilisation. FreeRTOS est un système d'exploitation temps réel (RTOS) faible empreinte, portable, préemptif et Open source pour microcontrôleur. Il a été porté sur plus de 40 architectures différentes. Créé en 2003 par Richard Barry et la FreeRTOS Team, il est aujourd'hui parmi les plus utilisés dans le marché des systèmes d'exploitation temps réel. Les domaines d'applications sont assez larges, car les principaux avantages de FreeRTOS sont l'exécution temps réel, un code source ouvert et une taille très faible. Il est donc utilisé principalement pour les systèmes embarqués qui ont des contraintes d'espace pour le code, mais aussi pour des systèmes de traitement vidéo et des applications réseau qui ont des contraintes "temps réel".

FreeRTOS est un système d'exploitation embarqué multitâches temps réel.



Text is not SVG - cannot display

Les ISR sont des sections de code exécutées par le microcontrôleur et non par FreeRTOS. Ce qui amène à des comportements inattendus du noyau. Pour cette raison, il est nécessaire de réduire au maximum le temps d'exécution d'une ISR. FreeRTOS fournit des méthodes servant à la gestion de ces interruptions et peut également lancer des interruptions par appel à une instruction matérielle.



Modèle simplifié

Dans l'état **Running** le processeur exécute le code associé à la tâche.
Dans l'état **Not Running**, la tâche est dormante (en veille) son status a été sauvegardé comme 'prête' (ready) pour que l'exécution puisse être reprise quand le scheduler décidera de la replacer dans l'état *Running*.
Une tâche passant de l'état *Not Running* à *Runnig* est considérée comme étant switched in ou swapped in.
Au contraire, une tâche passant de l'état *Running* à *Not Running* est dite comme étant switched out ou swapped out.
L'ordonnanceur de FreeRTOS est la seule entité permettant de réaliser un switch in ou out d'une tâche.
Quand une tâche est en reprise d'exécution, elle reprend depuis la dernière instruction effectuée après avoir quitté le l'état de Running. Cf. la commutation de contexte.

L'état Running

Une tâche obtient le processeur lorsqu'elle passe dans l'état Running
Le code d'une tâche est exécuté uniquement lorsqu'elle est dans cet état
Une tâche sort de l'état élu dans les situations suivantes :

Passage dans l'état Prêt :
Préemption par une tâche de priorité supérieure

Passage dans l'état bloqué :
Accès à une ressource non disponible
Attente d'un événement

Une commutation de contexte est effectuée quand une tâche sort ou entre dans l'état élu.

L'état supprimé

Le dernier état que peut prendre une tâche est l'état "supprimé", cet état est nécessaire car une tâche supprimée ne libère pas ses ressources instantanément. Une fois dans l'état "supprimé", la tâche est ignorée par l'ordonnanceur et une autre tâche nommée "IDLE" est chargée de libérer les ressources allouées par les tâches étant en état "supprimé".

Exemple de programme simple avec FreeRTOS

Création d'une tâche avec xTaskCreate

Chaque tâche nécessite de la RAM utilisée pour contenir l'état de la tâche (le bloc de contrôle de la tâche, ou TCB), et est utilisée par la tâche associée à la pile. Si une tâche est créée en utilisant xTaskCreate() alors la RAM nécessaire est automatiquement allouée à partir de FreeRTOS. Si une tâche est créée en utilisant xTaskCreateStatic() alors la RAM est fournie par l'auteur de l'application, ce qui entraîne deux paramètres de fonction supplémentaires, mais permet à la RAM d'être allouée statiquement au moment de la compilation. Les tâches nouvellement créées sont initialement placées dans l'état Prêt, mais deviennent immédiatement des tâches en cours d'exécution s'il n'y a pas de tâches plus prioritaires capables de s'exécuter. Les tâches peuvent être créées aussi bien avant qu'après le démarrage de l'ordonnanceur.

Création d'une tâche

```
void Task1( void *pvParameters );
const char *pvParametersTask1 = "Passage d'un pointeur sur une chaîne de caractères !";

/**
* @fn main
* @brief main entry point
*/
int main(void){

  // Création d'une  tâche
  xTaskCreate( Task1,                  // Pointeur sur la fonction implémentant la tâche
      "Tache 1",                       // Chaîne de caractères associée à la tâche (debug).
      configMINIMAL_STACK_SIZE,        // Taille de la pile associée à la tâche
      pvParametersTask1,               // Paramètre passé à la tâche
      tskIDLE_PRIORITY + 1,            // Priorité associée à la tâche
      NULL);                           // "handle" pour la gestion de la tâche
  ...
}
```

xTaskCreate()

Les tâches sont créées à l'aide de la fonction API xTaskCreate() de FreeRTOS. C'est probablement la plus complexe de toutes les fonctions API, il est donc regrettable qu'elle soit ici abordée en premier, mais les tâches doivent être maîtrisées en premier car elles sont le composant le plus fondamental d'un système multitâche.

pvTaskCode

Les tâches sont des fonctions C, sans fin, qui sont normalement implémentées sous la forme d'une boucle infinie. Le paramètre pvTaskCode est simplement un pointeur vers la fonction.

(En fait, juste le nom de la fonction) qui implémente la tâche.

pcName

Un nom descriptif pour la tâche. Il n'est en aucun cas utilisé par FreeRTOS. Il s'agit purement d'une aide au débogage. Identifier une tâche par un nom lisible par l'humain est beaucoup plus simple que d'essayer de faire la même chose à partir de son handle.

La constante configMAX_TASK_NAME_LEN définie par l'application, définit la longueur maximale d'un nom de tâche. Longueur maximale d'un nom de tâche, y compris le terminateur NULL.

Si vous fournissez une chaîne de caractères plus longue que ce maximum, elle sera simplement tronquée en silence.

configMINIMAL_STACK_SIZE

Chaque tâche possède un état unique qui lui est alloué par le noyau lors de sa création. à la tâche lors de sa création. La valeur configMINIMAL_STACK_SIZE indique au noyau la taille de la pile.

Cette valeur indique le nombre de mots que la pile peut contenir, et non le nombre d'octets.

Par exemple, si la pile a une largeur de 32 bits et que la valeur configMINIMAL_STACK_SIZE est de

100, 400 octets d'espace de pile seront alloués (100 * 4 octets).

Si votre tâche utilise beaucoup d'espace de pile vous devrez assigner une valeur plus importante.

Attention avec les processeurs ne possédant que peu de RAM !

pvParameters

Les fonctions de tâches acceptent un paramètre de type pointeur sur void (void*). La valeur assignée à pvParameters sera la valeur transmise à la tâche.

uxPriority

Définit la priorité d'exécution de la tâche. Les priorités peuvent être attribuées

de 0, qui est la priorité la plus basse, à (configMAX_PRIORITIES - 1), qui est la priorité la plus élevée.

configMAX_PRIORITIES est une constante définie par l'utilisateur. Il n'y a pas de limite supérieure au nombre de priorités qui peuvent être disponibles (autre que la limite des types de données utilisés et la RAM disponible dans votre microcontrôleur).

VTaskDelay

Retarde une tâche d'un nombre donné de ticks. Le temps réel pendant lequel la tâche reste bloquée dépend de la fréquence des tics. La constante portTICK_PERIOD_MS peut être utilisée pour calculer le temps réel à partir du taux de tick - avec une résolution d'une période de tick.

A l'usage j'ai constaté que ce temps n'est pas très précis si plusieurs processus sont en cours.

Il vaut mieux utiliser VTaskDelayUntil…

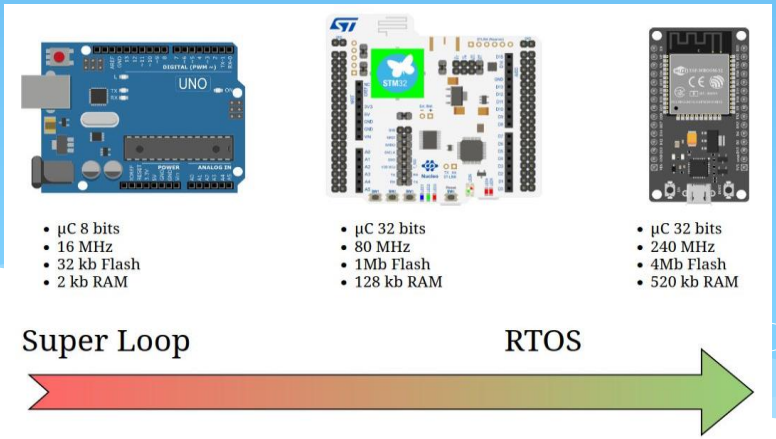
pxCreatedTask

Utilisé pour renvoyer un identifiant par lequel la tâche créée peut être référencée.

FreeRTOS est un système vivant en continuelle évolution et améliorations

Bilan

Selon une enquête UBM Embedded Developer, publiée en avril 2019, plus de 59 % des les projets nécessitent des capacités temps réel, plus d'un tiers utilisent une interface graphique et, par conséquent, plus de 67 % déclarent utiliser un RTOS ou scheduler de quelque sorte. Parmi les 33 % restants qui n'utilisaient pas de RTOS, la principale raison de ne pas en utiliser un (86 %) était que l'application n'en « avait pas besoin ». Parmi ceux qui ont choisi un RTOS commercial, 45 % ont cité en raison n° 1, la "capacité en temps réel". Les RTOS sont bien adaptés aux architectures 32 bits des microcontrôleurs STM32 (ST) ou ESP32 (Espressif) qui possèdent suffisamment de RAM pour exécuter en plus du RTOS, les tâches applicatives.



Deux exemples pratiques avec Arduino Uno et, enfin, Fin de l'exposé!

```
#include <Arduino_FreeRTOS.h>
void setup()
//Initialise le Serial Monitor avec un rate de
9600 baud
{
Serial.begin(9600);
//Serial.println(F("Dans le Setup !!!"));
//Force les pins digitales 3 à 5, en sorties
  pinMode(2,OUTPUT);
  pinMode(3,OUTPUT);
  pinMode(4,OUTPUT);
  pinMode(5,OUTPUT);
//Nous créons trois tâches intitulées Task1,
Task2 et Task3 et leur attribuons une priorité
de 1, 2 et 3 respectivement.
//Nous créons également une quatrième tâche
appelée IdelTask lorsqu'il n'y a pas de tâche
en cours d'exécution.
 xTaskCreate(MyTask1, "Task1", 100, NULL, 1,
NULL);
 xTaskCreate(MyTask2, "Task2", 100, NULL, 2,
NULL);
 xTaskCreate(MyTask3, "Task3", 100, NULL, 3,
NULL);
 xTaskCreate(MyIdleTask, "IdleTask", 100, NULL,
0, NULL);}
//Nous pouvons modifier la priorité des tâches
selon nos souhaits en changeant les chiffres
entre les textes NULL.
void loop()
{
//Il n'y a pas d'instruction dans la section
boucle du code.
// Parce que chaque tâche s'exécute sur
l'interruption après le temps spécifié
}
//La fonction suivante est Task1. Nous
affichons l'étiquette de la tâche sur le
moniteur de série.
static void MyTask1(void* pvParameters)
{
  while(1)
  {
    digitalWrite(2,HIGH);
    digitalWrite(3,LOW);
    digitalWrite(4,LOW);
    digitalWrite(5,LOW);
    //Serial.println(F("Task1"));
    Serial.println("Task1");
    vTaskDelay(200/portTICK_PERIOD_MS);
  }
}
//Similaire à la tâche 1
static void MyTask2(void* pvParameters)
{
while(1)
  { digitalWrite(2,LOW);
    digitalWrite(3,HIGH);
    digitalWrite(4,LOW);
    digitalWrite(5,LOW);
    //Serial.println(F("Task2"));
    Serial.println("Task2");
    vTaskDelay(50/portTICK_PERIOD_MS);
  }
}
static void MyTask3(void* pvParameters)
{
while(1)
  {
   digitalWrite(2,LOW);
   digitalWrite(3,LOW);
   digitalWrite(4,HIGH);
   digitalWrite(5,LOW);
   //Serial.println(F("Task3"));
   Serial.println("Task3");
   vTaskDelay(1000/portTICK_PERIOD_MS);
  }
}
//Il s'agit de la tâche "inactive" qui est
appelée lorsqu'aucune tâche n'est en cours
d'exécution.
static void MyIdleTask(void* pvParameters)
{
  while(1)
   {
    digitalWrite(2,LOW);
    digitalWrite(3,LOW);
    digitalWrite(4,LOW);
    digitalWrite(5,HIGH);
    //Serial.println(F("Etat inactif"));
    Serial.println("Etat inactif");
    delay(200);
```

```
//Une autre manière de faire clignoter des leds avec xTaskCreate !

#include <Arduino_FreeRTOS.h>
void TaskLoop1(void *parms);
void TaskLoop2(void *parms);
void TaskLoop3(void *parms);
void TaskLoop4(void *parms);

void setup()
{
  xTaskCreate(TaskLoop1, "Loop1", 100, NULL, 1, NULL);
  xTaskCreate(TaskLoop2, "Loop2", 100, NULL, 1, NULL);
  xTaskCreate(TaskLoop3, "Loop1", 100, NULL, 1, NULL);
  xTaskCreate(TaskLoop4, "Loop2", 100, NULL, 1, NULL);
}
void loop() {}

void TaskLoop1(void *parms)
{
  pinMode(2, OUTPUT);
  while(true)
  {
    digitalWrite(2, HIGH);
    vTaskDelay(100/portTICK_PERIOD_MS);
    digitalWrite(2, LOW);
    vTaskDelay(2000/portTICK_PERIOD_MS);
  }
}
void TaskLoop2(void *parms)
{
  pinMode(3, OUTPUT);
  while(true)
  {
    digitalWrite(3, HIGH);
    vTaskDelay(100/portTICK_PERIOD_MS);
    digitalWrite(3, LOW);
    vTaskDelay(500/portTICK_PERIOD_MS);
  }
}
  void TaskLoop3(void *parms)
{
  pinMode(4, OUTPUT);
  while(true)
  {
    digitalWrite(4, HIGH);
    vTaskDelay(50/portTICK_PERIOD_MS);
    digitalWrite(4, LOW);
    vTaskDelay(200/portTICK_PERIOD_MS);
  }
}

  void TaskLoop4(void *parms)
{
  pinMode(5, OUTPUT);
  while(true)
  {
    digitalWrite(5, HIGH);
    vTaskDelay(100/portTICK_PERIOD_MS);
    digitalWrite(5, LOW);
    vTaskDelay(100/portTICK_PERIOD_MS);
  }
}
```