Introduction aux RTOS

- Introduction aux systèmes embarqués I
- Introduction aux systèmes embarqués II

Introduction aux systèmes embarqués I

Définition

Un système embarqué est défini comme :

- un système électronique et informatique autonome,
- souvent temps réel,
- spécialisé dans une tâche précise.

En Anglais : **Embedded System**

Ses ressources sont généralement :

- limitées spatialement (encombrement réduit)
- énergétiquement (consommation restreinte).

Système embarqué vs PC

La distinction entre un système type PC et un système embarqué se fera essentiellement au niveau de la spécialisation dans une tâche précise:

- un smartphone est plutôt considéré comme un PC, la fonction téléphone est une parmi les applications possibles
- un automate Beckhoff, basé sur une architecture PC avec Win10 est considéré comme un système embarqué, prévu pour piloter une machine spécifique, le kernel Windows est préempté par le Runtime pour assurer les contraintes temporelles.

Embedded Systems



Computer like



Les systèmes embarqués sont omniprésents :

- grand public : robot ménager, four, drone ...
- médical : défibrillateur, pompe à insuline, pacemaker, ...
- industriel : automate, camera industrielle, calculateur abs, ...
- IoT : capteur d'analyse de vibrations connecté Cloud, mesure d'énergie connectée Cloud, ...
- ...

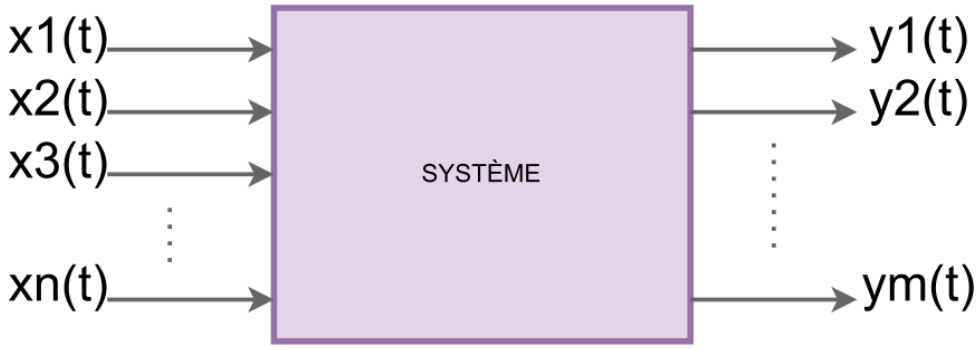
Le temps réel

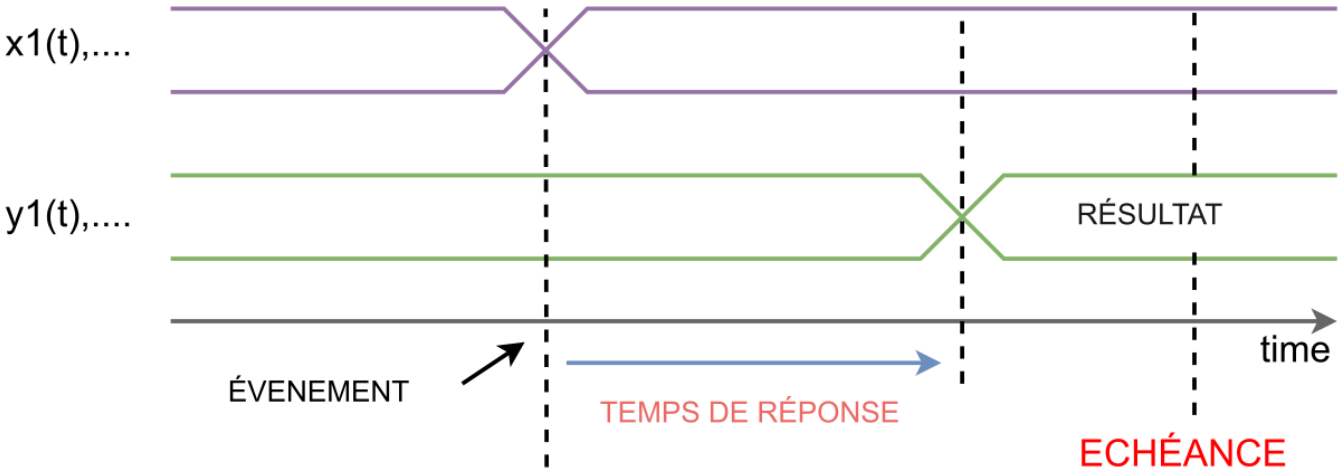
Contraintes temporelles

Le temps réel n'englobe pas la notion de vitesse mais garantit l'exécution d'une tâche à temps. La réaction du système est prévisible quel que soit sa charge processeur, les interruptions à traiter, ...

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)





L'échéance (deadline) fixe le délai maximal alloué au système temps-réel pour obtenir le résultat après un événement.

Question : que se passe-t-il si le temps de réponse dépasse l'échéance ?

- cela dépend de la classification de la contraite temps réel. cf ci-dessous

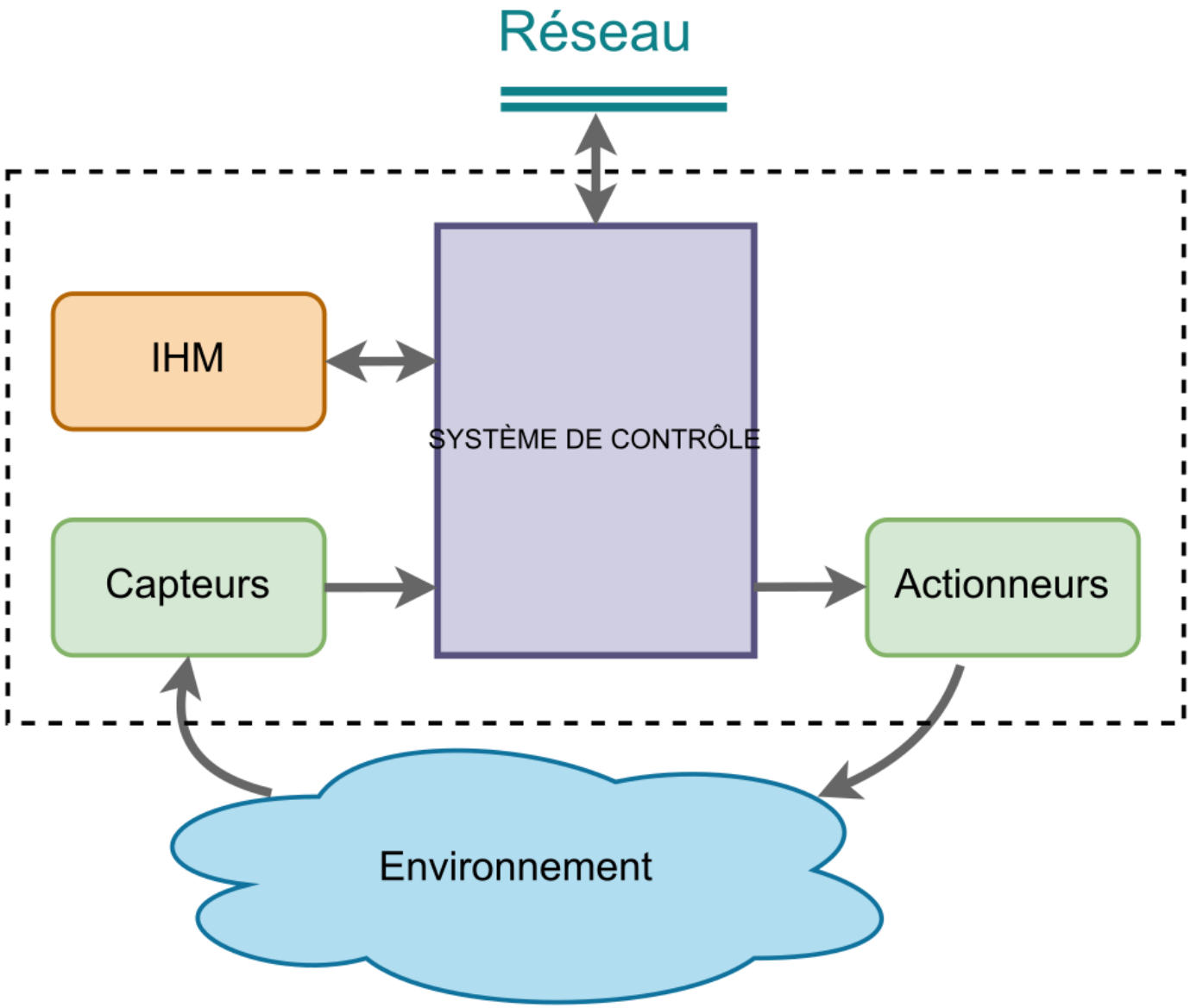
Classification des systèmes temps-réel

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.

Représentation d'un système embarqué

- Les capteurs vont mesurer les paramètres d'environnement
- Le système de contrôle traite les informations des capteurs et de l'IHM pour générer les consignes de pilotage des actionneurs
- un bus de communication permet la supervision ou le pilotage à distance



Faisons une analyse rapide pour un drone.

L'environnement est aérien, pouvons être soumis à des turbulences, des obstacles, des pertes de connectivités, etc. Les périphériques et l’environnement du système évoluent simultanément (en parallèle ou concurrence), la gravité agit toujours ;)









Capteurs	Actionneurs	IHM	Réseau
gyroscopes	moteurs brushless	boutons	télémétrie wifi
accélèromètre	servo-moteur	voyants lumineux état de marche	pilotage RF
gps	...	smartphone	...
magnétomètre		...	
encodeur de position moteurs			
camera			
lidar			
...			

Le système se décompose en plusieurs activités (tâches) qui doivent souvent être exécutées en parallèle :

- mesure d'orientation angulaire
- mesure d'altitude
- génération de signaux PWM pour les moteurs
- ...

Prévisibilité

Il faut être capable de prouver/démontrer/vérifier qu'un système temps réel va obéir au contraintes temps-réel telles que définies lors de la phase de spécification.

Eléments pris en compte :

- Identification des échéances à temps-réel hard/soft
- Charge du processeur
- Temps de traitement de chaque tâche
- Méthode d'ordonnancement des tâches
- Méthode «Worst Case ExecutionTime (WCET) »

Déterminisme

La maîtrise du déterminisme permet de garantir la prévisibilité du système.

Difficultés :

- Mise en oeuvre dans un environnement non déterministe
  - Evénements asynchrones : interruptions
- Non déterminisme lié au système
  - Mécanismes destinés à l'amélioration globale des performances de l'architecture matérielle (mémoire cache, prédiction de branchement, . . .)
  - Pannes matérielles

La notion de tâche

L'application s'exécutant sur le processeur est structurée en tâches (task)

- Une tâche effectue un ensemble d'opérations destinées à fournir un service à l'application.
- Une tâche constitue une unité élémentaire de traitement

La suite d'instructions composant une tâche est exécutée séquentiellement par le processeur

- Tâche implémentée sous forme de routine

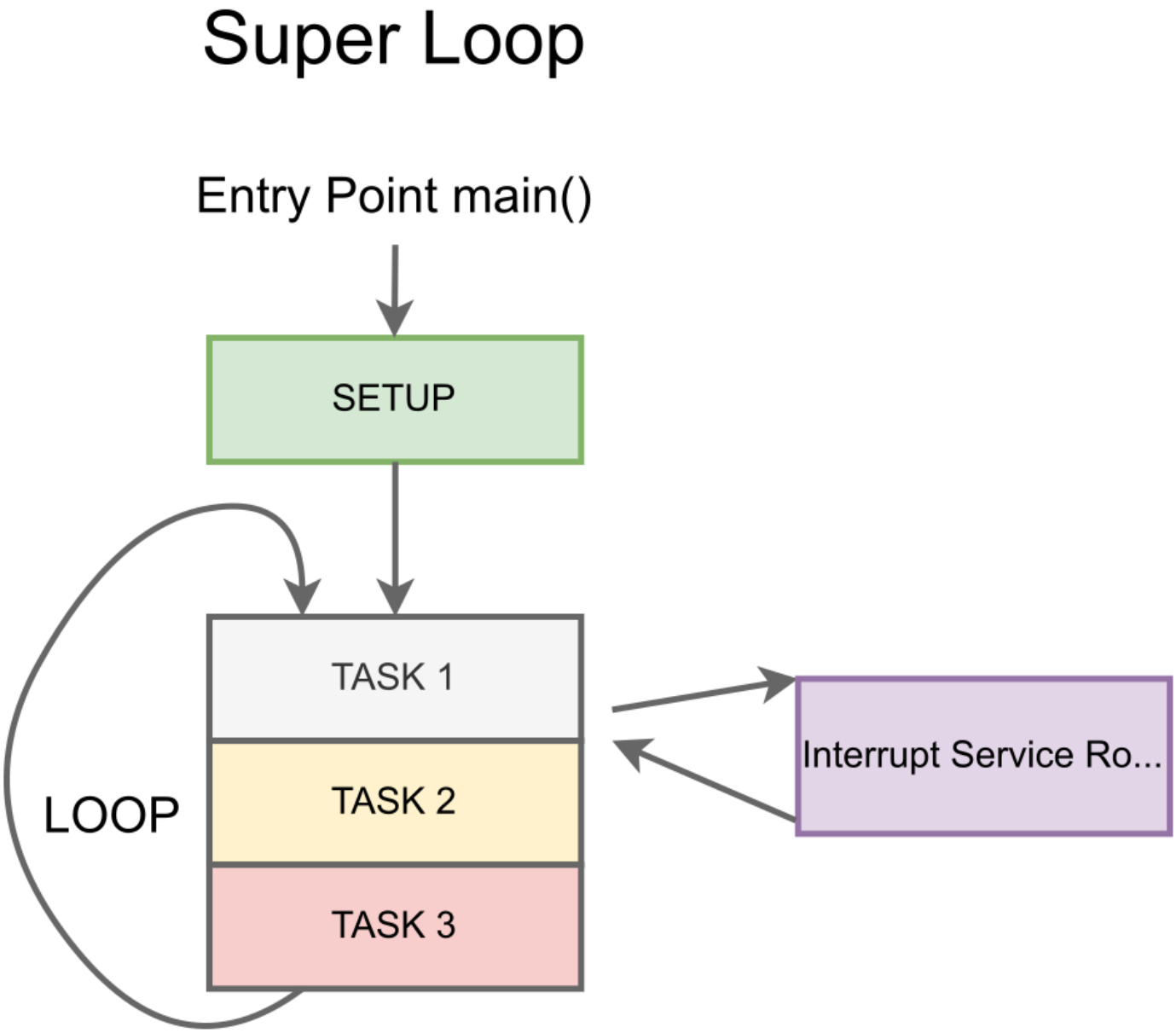
Décomposition d'une application en tâches

Reprenons l'exemple du drone et concentrons nous sur l'application permettant d'assurer un vol stationnaire :

Tâches associées aux E/S	Tâches assurant des traitements internes
lecture des gyroscopes	calcul des positions angulaire en fonction des données gyroscopes
lecture baromètre	calcul de l'altitude en fonction de la valeur du baromètre
lecture boussole	enregistrement de la position et des paramètres dans la carte SD
pilotage pwm des moteurs	cryptage de la communication
transmission des données en Wifi	...
...	...

Exécution cyclique ou Super Loop

Il s'agit de la méthode classique de programmation d'un microcontrôleur. Les tâches sont exécutées les unes à la suite des autres. Certaines tâches de niveau de priorité plus importantes génèrent une interruption qui permet de préempter la tâche en cours. Quand la routine d'interruption est traitée, on sort de l'ISR pour reprendre la tâche interrompue.



Avantages

- Bonne prévisibilité :
  - Absence de mécanismes de synchronisation inter-tâches
  - Validation relativement aisée
- L'ordonnancement ne consomme pas de ressources processeur
- Surdimensionnement minimal : le WCET détermine directement la période du cycle

Inconvénients

- Très peu flexible
- La complexité augmente fortement avec le nombre de tâches et leurs interdépendances

-> La Super Loop est à mettre en œuvre dans des systèmes simples, gérant peu de tâches.

Un drone est un système complexe avec de nombreuses tâches qui doîvent partager les ressources matérielles, s'assurer de l'échange des mesures, respecter les contraintes temporelles pour que l'asservissement des commandes de vol reste fonctionnel etc.

Système d'exploitation - Operating System (OS)

Dans cette section, nous allons introduire la notion de système d'exploitation (OS) et analyser les différences entre :

- GPOS : General Purpose OS
- RTOS : Real Time OS
- Bare Metal : système sans OS

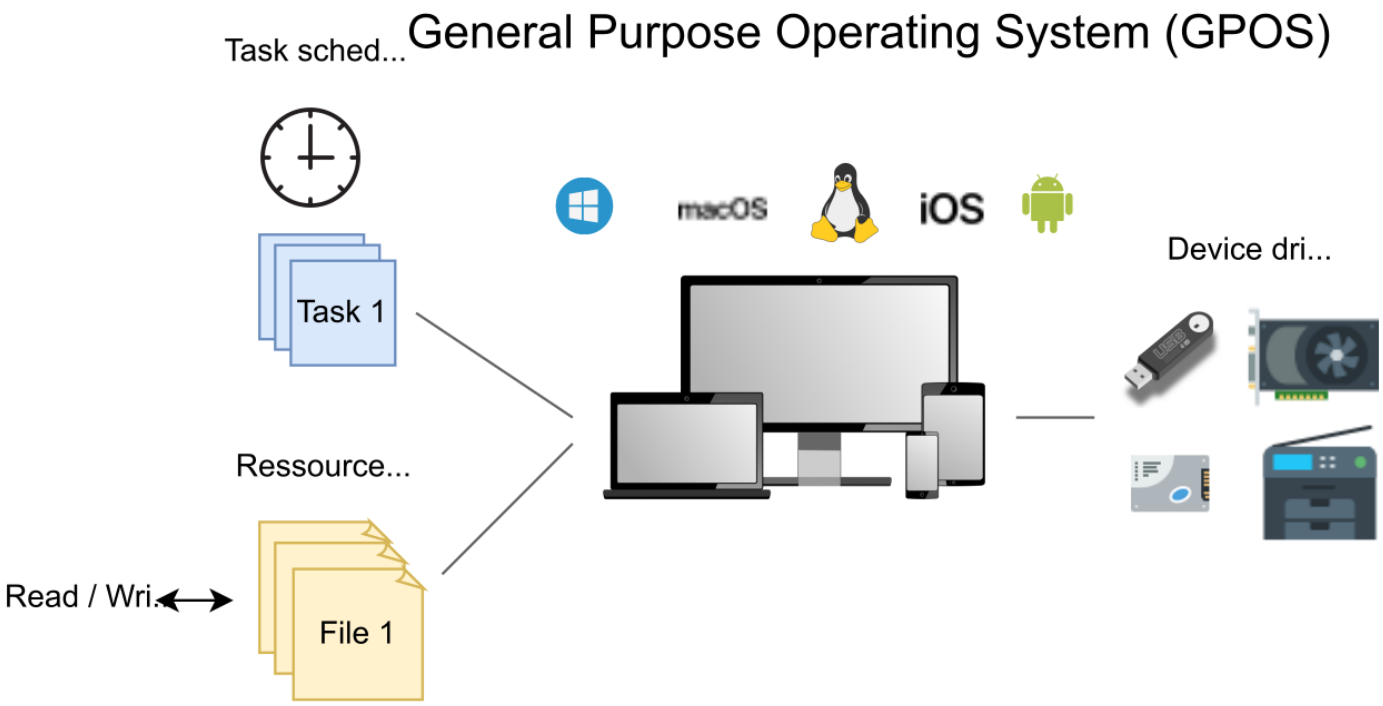
General Purpose Operating System (GPOS)

Un système d’exploitation (Operating System ou OS en anglais) est un programme réalisant les fonctions élémentaires suivantes :

- Le découpage en tâches d’une application
- Ordonnancement (Scheduling) des tâches à exécuter et l’allocation d’un temps processeur à ces différentes tâches.
- Protection de l’intégrité des données stockées et exécutées (en mémoire, sur les unités de stockage)
- Gestion des ressources physiques de l’ordinateur (temps cpu, mémoire, périphériques)
- Assurer un niveau d’abstraction du matériel pour l’utilisateur

Le Scheduler (Ordonnanceur) d'un General Purpose Operating System (GPOS) va optimiser l'exécution des tâches de manière à apporter le maximum de confort à l'utilisateur. Ce type d'ordonnancement ne permet pas de garantir l'exécution dans les délais d'une tâche parmi d'autres et ne sont pas adapté aux système embarqués temps réel.

Dans la famille des GPOS, on trouve Windows, Linux, macOS, Android, iOS.



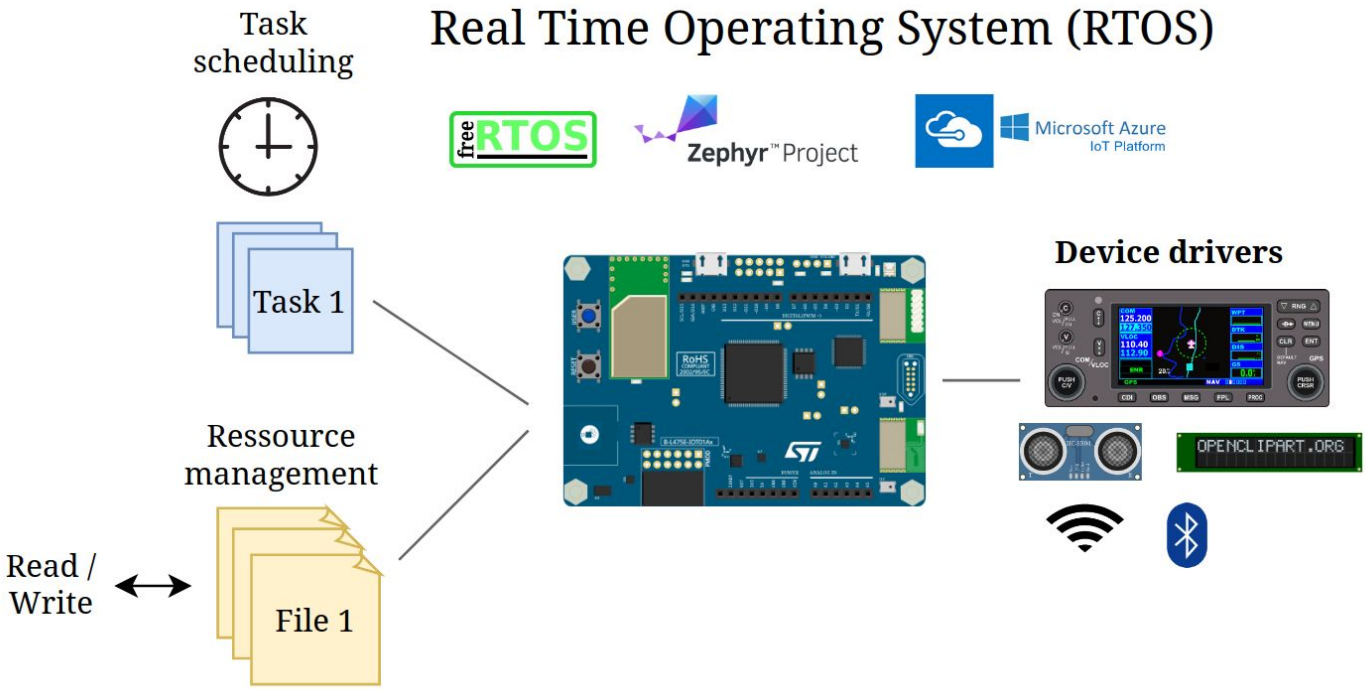
On retrouve fréquement Linux dans l'embarqué :

- Raspberry Pi,
- Jetson Nano,
- Odroid,

Linux est compatible avec de nombreuses architectures matérielles et apporte un excellent support matériel, mais Linux reste de base un GPOS en ce qui concerne les contraintes temps réel. On peut améliorer les choses avec le patch PREEMPT_RT, ou alors, avec le co-noyau Xenomai.

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, ...



Bare metal vs RTOS

On parle de programmation Bare metal lorsqu'on programme un microcontrôleur sans OS. Les RTOS sont prévus pour fonctionner sur des microcontrôleurs avec de faibles ressources matériel (RAM, Flash), néanmoins ils restent plus à l'aise sur des architectures 32 bits que sur les microntrôleurs 8 bits comme les PIC16F. Un RTOS consomme de la ressource, et sur un processeur 8 bits qui en possède très peu, il est plus efficace de rester en programmation bare metal avec une Super Loop.

Passons en revue les avantages et inconvénients de l'utilisation d'un RTOS:

Bénefices

- garantie les performances temps réel

- facilite le développement et peut réduire les coûts
- facilite l'ajout de nouvelles fonctionnalités
- simplifie la portabilité de l'application
- simplifie les certifications de sécurité

Coûts

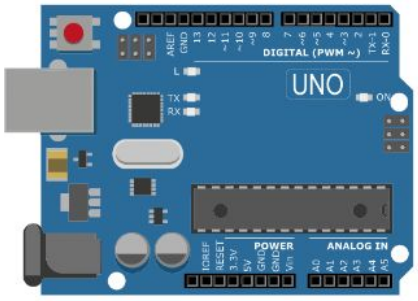
- engendre des coûts (achat de Licence, coût de formation)
- consomme de la mémoire et des cycles processeurs
- peut être "Overkill" pour l'application développée

En programmation Bare Metal, quand les contraintes se multiplient, il est possible de se retrouver à programmer les mécanismes d'un RTOS au risque de réinventer la roue et d'avoir des perfomances moindres. La question du passage vers un RTOS se pose alors.

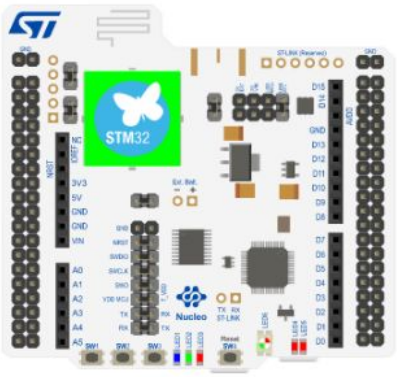
Bilan

Selon la dernière 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.



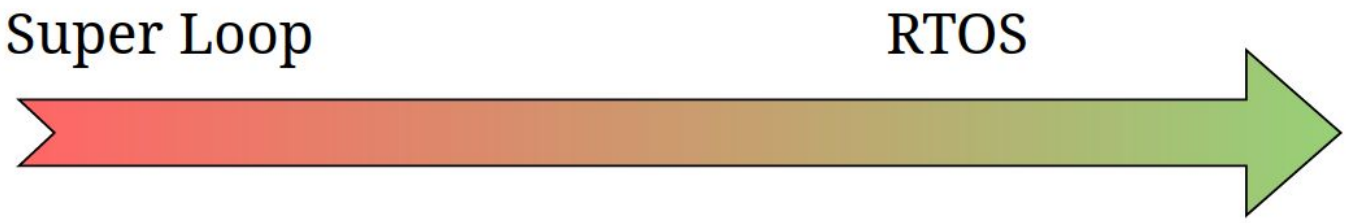
- µC 8 bits
- 16 MHz
- 32 kb Flash
- 2 kb RAM



- µC 32 bits
- 80 MHz
- 1Mb Flash
- 128 kb RAM



- µC 32 bits
- 240 MHz
- 4Mb Flash
- 520 kb RAM



Avec la migration continue vers les microprocesseurs 32 bits et les milliards de nouveaux appareils IoT qui arrivent sur le marché dans les années à venir, il existe de solides arguments en faveur de l'utilisation d'un RTOS.

Real Time Operating System

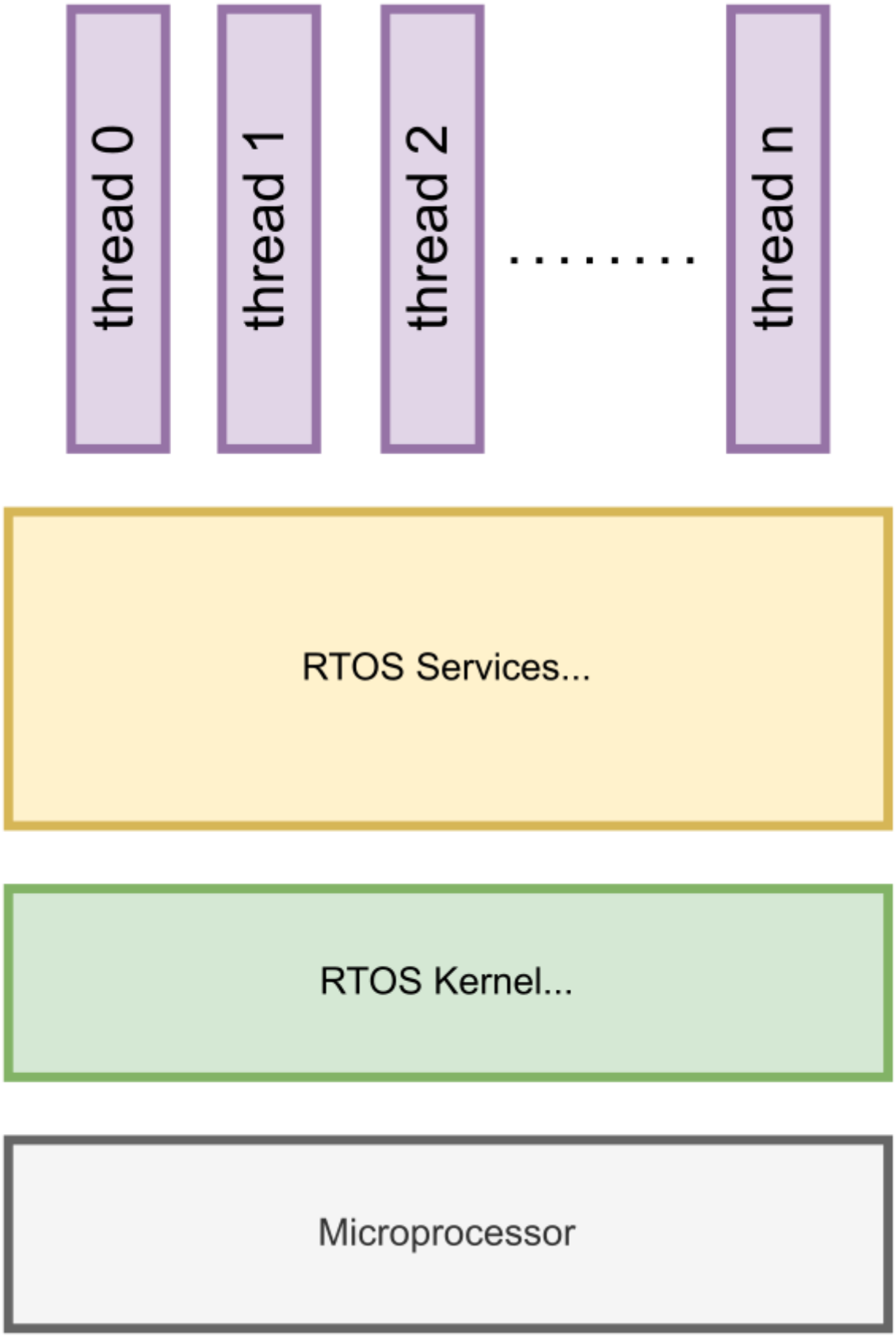
Le RTOS est un logiciel système qui fournit des services et gère les ressources du processeur pour applications. Ces ressources incluent :

- les cycles du processeur,
- la mémoire,
- les périphériques
- les interruptions.

Le RTOS doit être capable d'assurer un fonctionnement en continu pendant des mois ou des années. Il expose 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 logiciel 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 fonctionner (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.

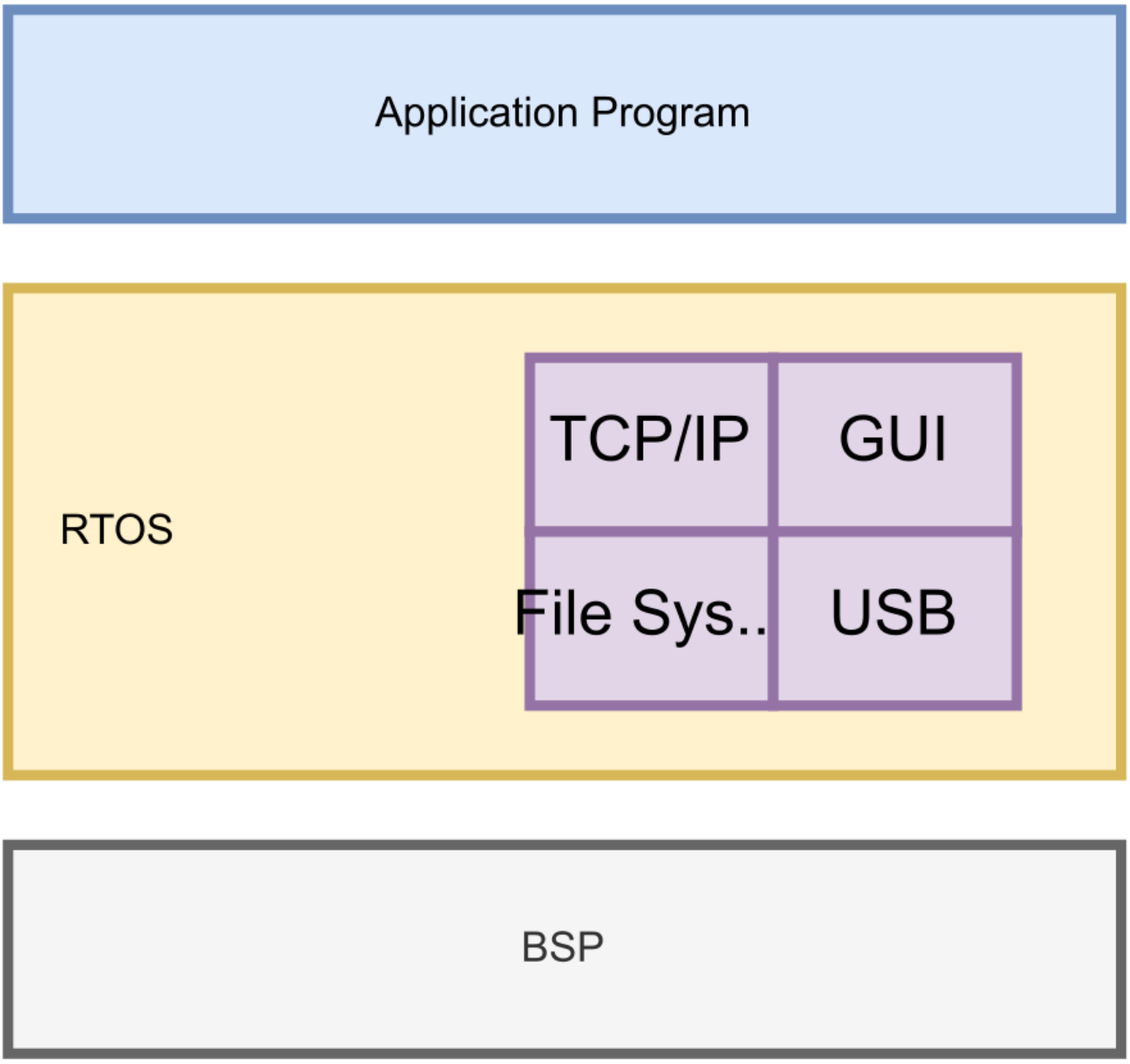


Il est important de noter qu'un RTOS doit offrir une préemption. La préemption est l'action de passer instantanément et de manière transparente à un thread de priorité supérieure, sans avoir à attendre l'achèvement du thread de priorité inférieure. En plus de l'allocation du processeur, un bon RTOS fournit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

Les opérations d'un RTOS sont éffectuées par le **KERNEL** (noyau)

D’autres services peuvent éventuellement être fournis :

- Couches de communication (TCP/IP)
- Graphical User Interface (GUI)
- Pilotes de périphériques
- Gestion de système de fichiers



Dans les systèmes embarqués, une couche supplémentaire peut exister, le board support package(BSP) contient le micrologiciel de démarrage spécifique au matériel, les pilotes de périphérique et d'autres routines qui permettent à un système d'exploitation embarqué donné, de fonctionner dans un environnement matériel donné (carte mère):

- Initialise la carte mère (horloges, registres processeur)
- Initialise la RAM
- Charge et lance l'OS depuis la flash

Élements de base du Kernel

Ordonnanceur	Objets du noyau	Services
Elément principal du noyau. Implémente l’algorithme d’ordonnancement qui détermine quelle tâche obtient les ressources du processeur. On parle en anglais de Scheduler.	Accessibles au programmeur pour le développement d’applications. Exemple d’objets : tâche, mutex, sémaphore (objet de synchronisation), file de messages.	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 multitâche

- 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.

La commutation de contexte

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

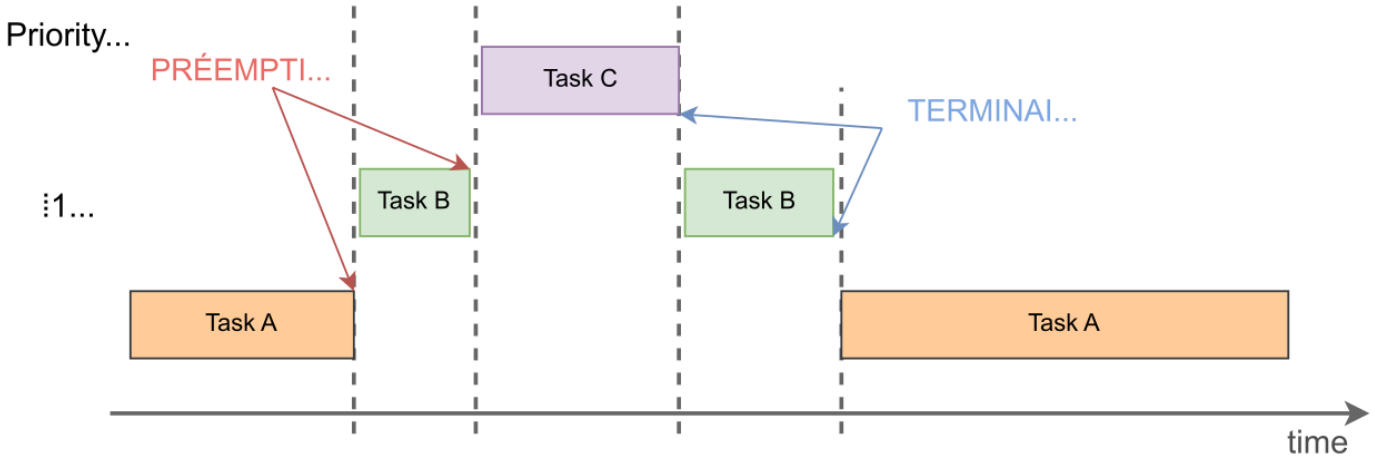
- Le contexte inclut généralement ces informations :
  - Le contenu des registres du processeur
  - La valeur du compteur de programme et du pointeur de pile
- Le module de l’ordonnanceur assurant la commutation du contexte est le répartiteur (dispatcher)

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 :

- préemptif avec priorités
- round-robin 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)

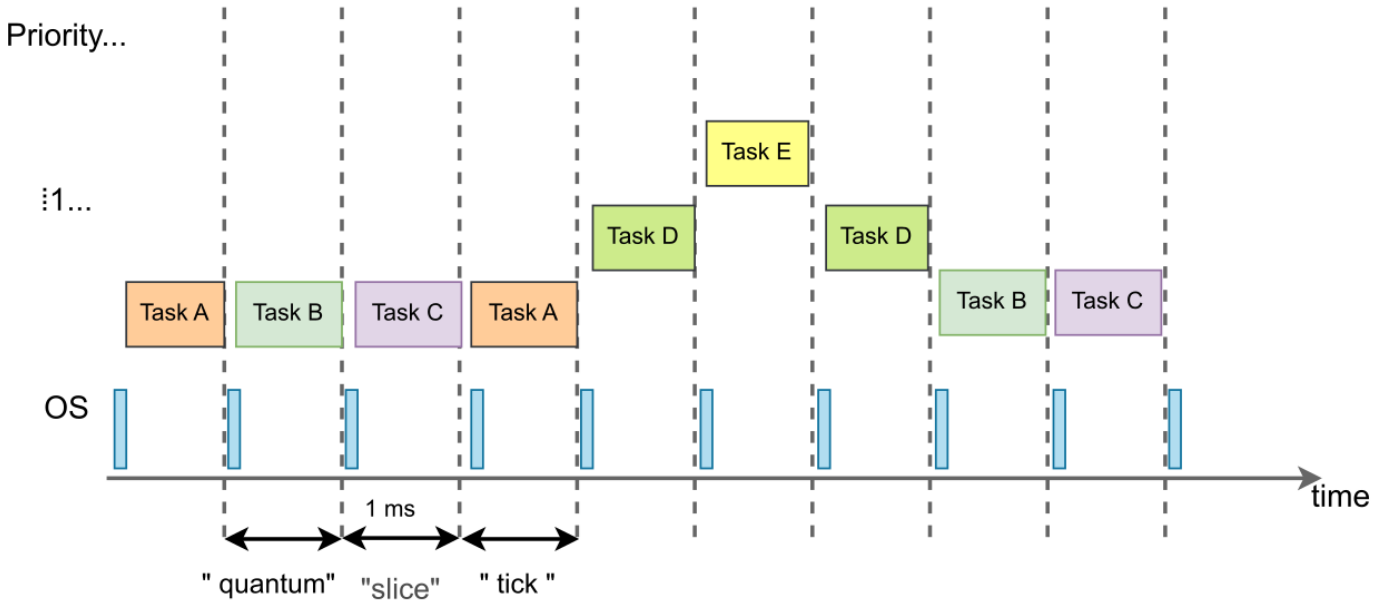
Plusieurs tâches peuvent avoir la même priorité

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

Ordonnancement « round-robin » avec priorités



La tâche de plus haute priorité prête à s'exécuter obtient toujours le temps processeur

- Les tâches de même priorité sont ordonnancées suivant la méthode du **tourniquet** (« round-robin ») :
- La durée d'exécution continue d'une tâche est au maximum la valeur du quantum (time quantum ou time slice)
- Lorsque la durée du quantum est atteinte, la tâche est préemptée pour donner la main à une tâche de même priorité

Les tâches de même priorité obtiennent ainsi un temps processeur équitable

Exemple de RTOS: FreeRTOS



http://www.freertos.org

Présentation de FreeRTOS

Système d’exploitation temps-réel, faible empreinte, portable et préemptif dont le code source est ouvert, sous licence MIT

- Versions commerciales :
  - OpenRTOS™ : ajout de services (pile TCP/IP, gestion USB, ...)
  - SafeRTOS™ : version certifiée pour utilisation dans des applications critiques (dans le domaine médical par ex.)

En 2017, Amazon fait l'acquisition de l'équipe et développe une version spécifique de FreeRTOS, *Amazon FreeRTOS* incluant les librairies facilitant l'intégration au Cloud Amazon AWS.

Il s'agit d'un système minimaliste, l'image binaire du noyau fait entre 4 Ko et 9 Ko ce qui le rend principalement destiné à être utilisé avec des microcontrôleurs ayant des performances modestes et une quantité mémoire limitée (32 – 256 Ko de Flash)

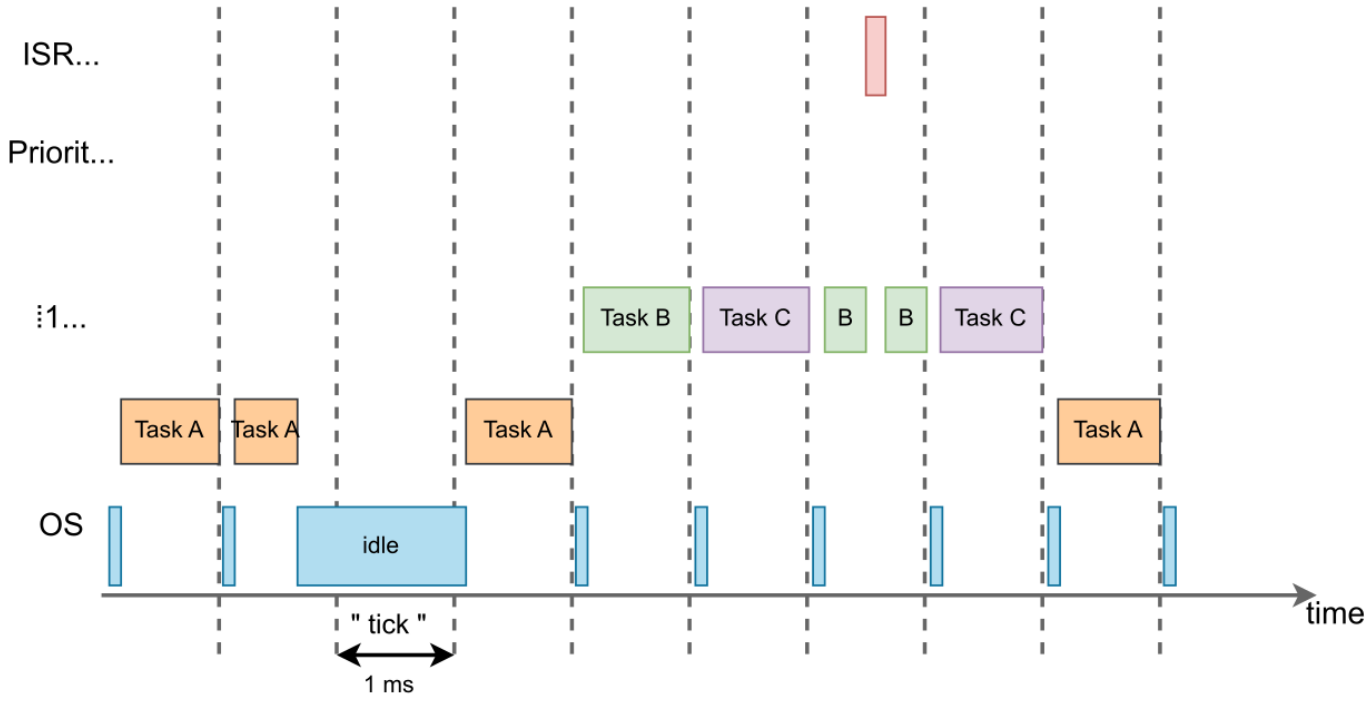
Caractéristiques de FreeRTOS

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



Les ISR sont des sections de code exécutées par le micro-contrôleur 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 des interruptions et peut également lancer des interruptions par appel à une instruction matérielle.

Les fichiers sources de FreeRTOS

La structure minimale du code source de FreeRTOS est contenue dans deux fichiers C qui sont communs à l'ensemble des portages de FreeRTOS (port sur une autre architecture processeur). Ces deux fichiers sont

- `tasks.c` : scheduler, traitement des tâches, ...
- `list.c` : liste de tâches ayant les propriétés ready, les mêmes niveau de priorité, ...

En plus de ces deux fichiers, nous pouvons associer :

- `queue.c` : permet les services de queues et de semaphore (quasiment toujours nécessaires)
- `timers.c` : permet la fonctionnalité de timer software. A utiliser uniquement si l'utilisation de timers softwares sont nécessaires.
- `event_groups.c` : procure la fonctionnalité d'évenements de groupe. A utiliser uniquement si l'utilisation des évenements de groupes sont nécessaires
- `croutine.c` implémente la fonctionnalité de co-routine pour FreeRTOS. Prévu au départ pour les microcontroleurs à très faibles capacités, son utilisation n'est plus d'actualité.

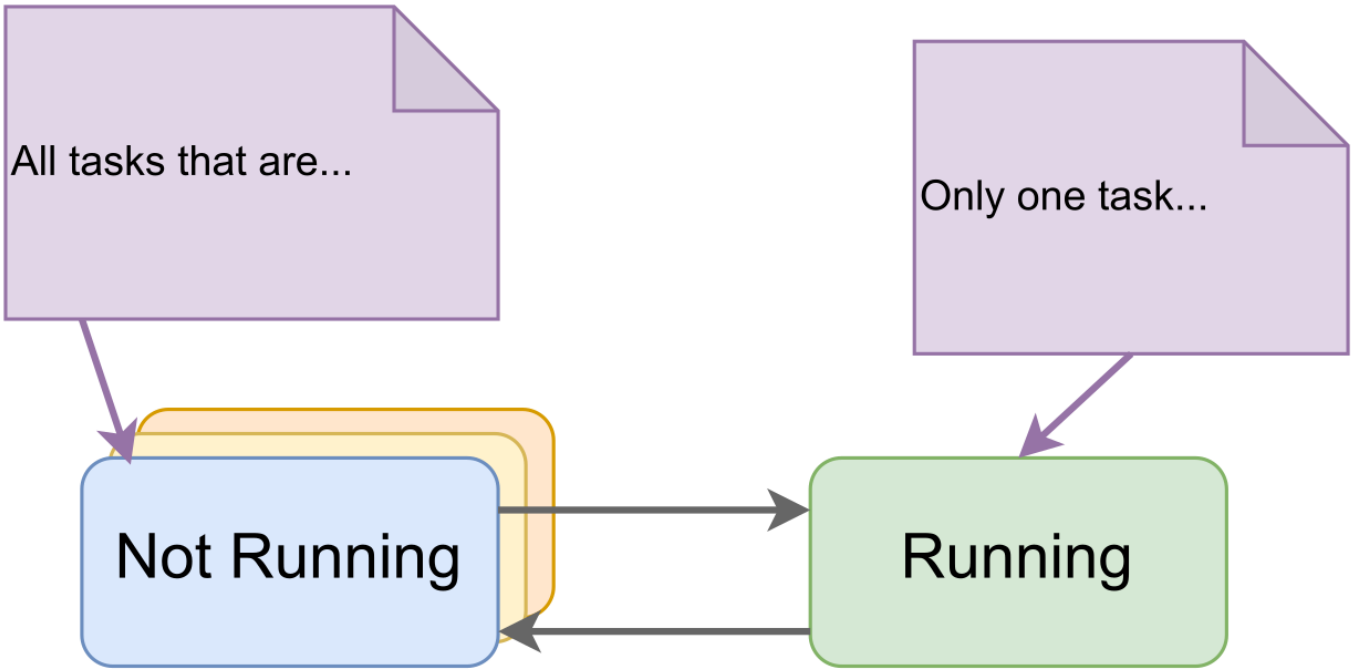
Introduction aux systèmes embarqués II

Le livre Mastering the FreeRTOS™ Real Time Kernel, a été utilisé pour cet article. Je reprendrais certains anglicisme par cohérence avec les sources et la documentation officielle.

FreeRTOS

Une application peut être consistuée de plusieurs tâches. Si le microcontrôleur est de type monoprocesseur, seule une tâche peut être exécutée pour un temps donné. En première approche, nous pouvons classer les tâches en deux états :

- **Running**
- **Not Running**



Ce modèle simplifié va être utilisé dans un premier temps.

- 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. C'est la commutation de contexte.

Problématique

Que se passe-t-il si une tâche A (task A) n'a rien à faire car elle est en attente d'une donnée / signal ... ? et imaginons que la tâche A est de même priorité que les tâches B et C.

L'ordonnanceur Round Robin va placer Task A en *Running*

- attendre la fin du quantum pour Task A
- placer Task B en *Running* durant un quantum
- placer Task C en *Running* durant un quantum
- placer Task A en *Running* qui aura peut-être l'information nécessaire pour faire autre chose qu'attendre, ou pas.

On remarque que l'on est pas très efficace à placer en *Running* une tâche qui ne fait rien mais le système reste opérationnel.

Que se passe-t-il si une tâche A (task A) n'a rien à faire car elle est en attente d'une donnée / signal et que cette tâche A est de priorité supérieure aux tâches B et C?

L'ordonnanceur Round Robin va placer Task A en *Running*

- attendre la fin du quantum pour Task A
- comme Task A est toujours ready et de priorité la plus élévée,
  - l'ordonnanceur replace Task A en *Running* pour un quantum

Task A empêche les tâches de priorité inférieures de s'exécuter.

Pour rendre les tâches utiles, elles doivent être réécrites pour être pilotées par des événements. Une tâche événementielle a un traitement à effectuer uniquement après l'occurrence de l'événement qui la déclenche. La tâche ne peut pas passer à l'état *Running* avant que cet événement ne se produise.

Utiliser des tâches événementielles signifie que les tâches peuvent être créées à différentes priorités sans que les tâches les plus prioritaires privent toutes les tâches de priorité inférieure du temps de traitement.

L'état bloqué

Une tâche qui attend un événement est dite à l'état "Bloqué"

Les tâches peuvent passer à l'état Bloqué pour attendre deux types d'événements différents :

1. Événements temporels (Délai)
2. Événements de synchronisation — lorsque les événements proviennent d'une autre tâche ou interruption

Les événements de synchronisation FreeRTOS : sémaphores, mutex, files de messagerie, groupes d'événements

La possibilité de blocage des tâches est fondamentale : *C’est le seul moyen pour des tâches de priorités inférieures d’être élues*

Une tâche peut se retrouver dans l'état "bloqué" lors de l’accès à une file en lecture/écriture dans le cas où la file est vide/pleine. Chaque opération d'accès à une file est paramétrée avec un timeout. Si ce timeout vaut 0 alors la tâche ne se bloque pas et l'opération d'accès à la file est considérée comme échouée. Dans le cas où le timeout n'est pas nul, la tâche se met dans l'état 'bloqué' jusqu'à ce qu'il y ait une modification de la file (par une autre tâche par exemple). Une fois l'opération d'accès à la file possible, la tâche vérifie que son timeout n'est pas expiré et termine avec succès son opération.

L'état suspendu

Une tâche peut être volontairement placée dans l'état "suspendu" par appel à la fonction API `vTaskSuspend()`. Elle sera alors totalement ignorée par l’ordonnanceur et ne consommera plus aucune ressource jusqu'à ce qu'elle soit retirée de l'état avec `vTaskResume()` et remise dans un état "prêt".

La plupart des applications n'utilisent pas l'état Suspendu.

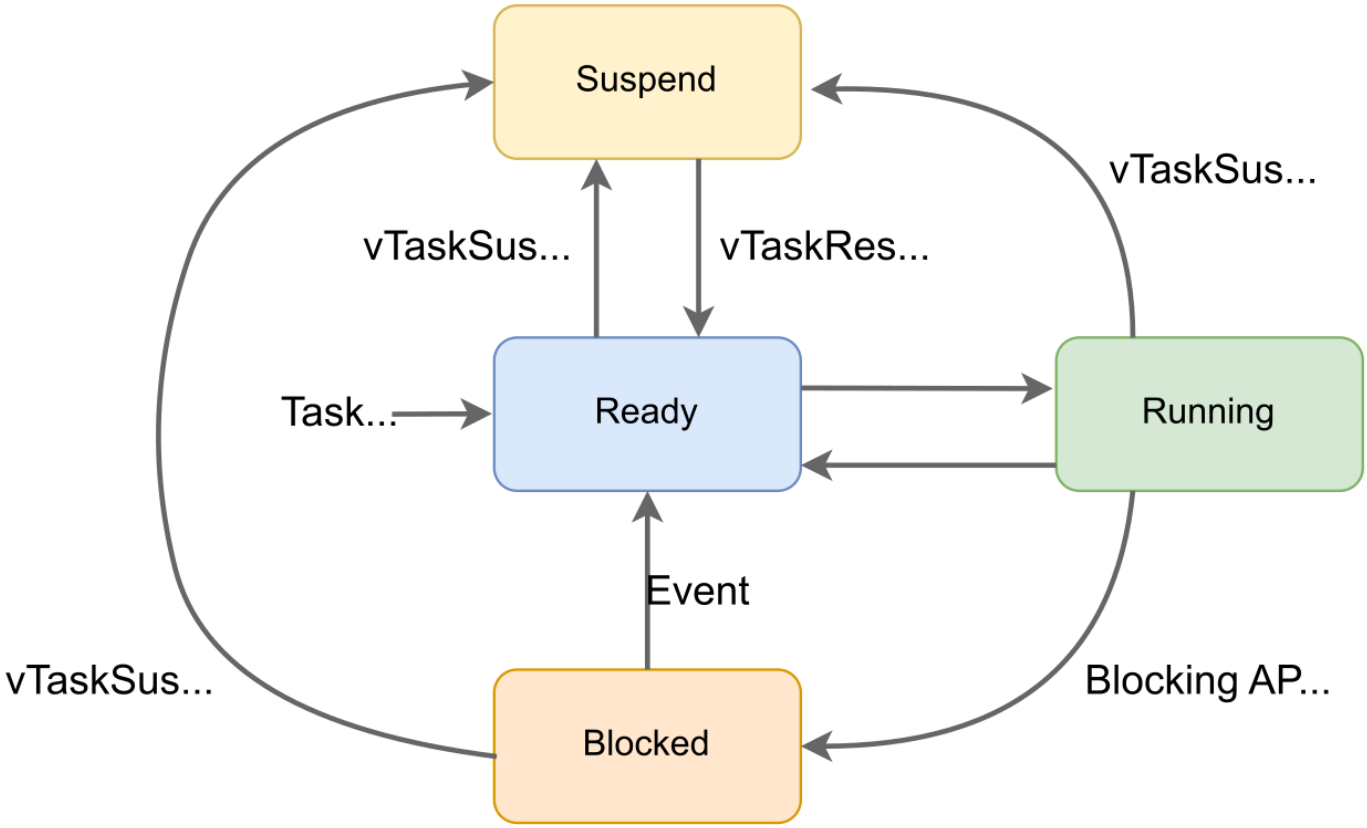
L'état prêt

Les tâches qui sont à l'état *Not Running* mais qui ne sont :

- ni bloquées
- ni suspendues

sont dites à l'état Prêt -> READY.

- Elles ont obtenu toutes les ressources nécessaires pour s’exécuter, sauf le processeur, et donc "prêtes" à fonctionner
- Une tâche est initialement placée dans l’état Ready après sa création
- Une tâche Ready passe à l’état Running suivant la politique d’ordonnancement (priorité, round-robin)



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
    - La durée du quantum est atteinte (ordonnancement « round-robin »)
  - Passage dans l’état bloqué :
    - Accès à une ressource non disponible
    - Attente d’un événement
    - Suspension de l’exécution par introduction d’un délai

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".

La tâche IDLE

La tâche 'IDLE' est créée lors du démarrage de l'ordonnanceur et se voit assigner la plus petite priorité possible.

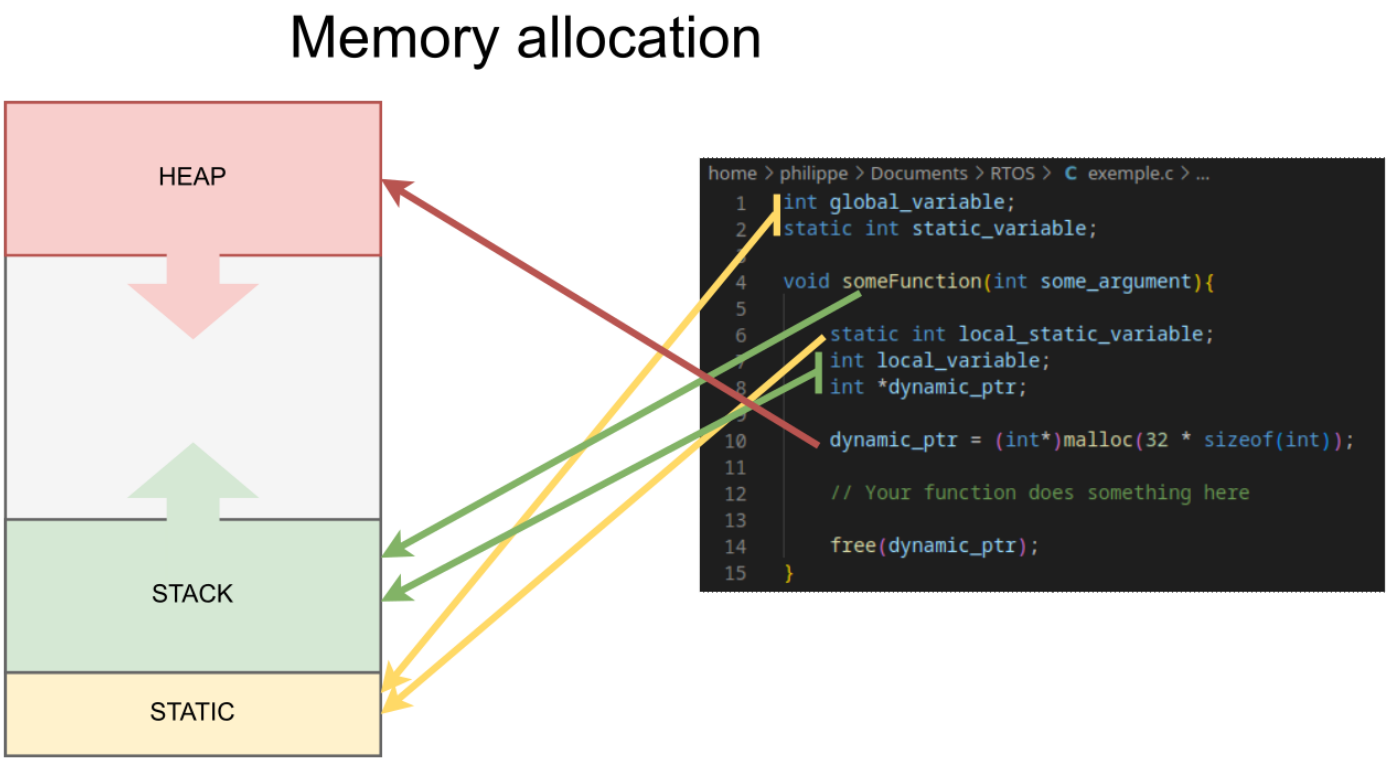
La tâche Idle peut avoir plusieurs fonctions à remplir dont :

- Libérer l'espace occupé par une tâche supprimée.
- Placer le micro-contrôleur en veille afin d'économiser l'énergie du système lorsqu'aucune tâche applicative n'est en exécution.
- Mesurer le taux d'utilisation du processeur.

Allocation mémoire

Rappel allocation mémoire pour une fonction

- Les constantes, les variables globales et variables statiques sont stokées en STATIC
- les variables locales sont stockées dans la STACK (pile)
- les variable dynamiques sont stockées dans la HEAP (tas)



Allocation mémoire dans FreeRTOS

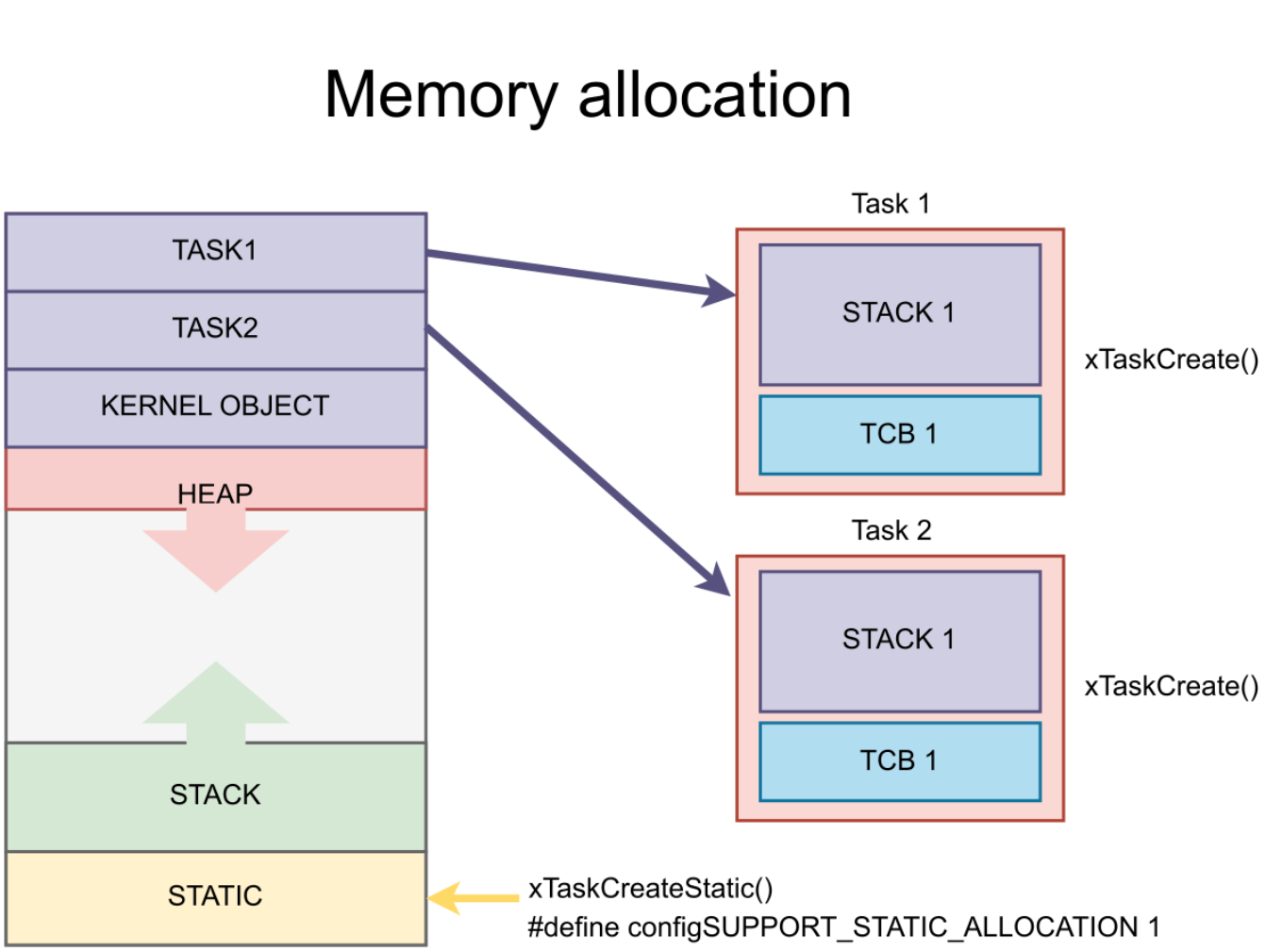
Les tâches et objets Kernel sont placées dans la HEAP

À la création d'une tâche, FreeRTOS crée et remplit la TCB correspondant à la tâche (Task control Block) qui contient toutes les informations nécessaires afin de spécifier et de représenter une tâche. Les Éléments associés à une tâche sont :

- La routine
  - Instructions (code) exécutées par la tâche
- Le niveau de priorité
  - Paramètre pris en compte par l'ordonnanceur
- L'identifiant
  - Attribué par le système
  - Utilisé comme paramètre dans les fonctions de gestions des tâches
- La STACK (pile)
  - Zone mémoire privée (stockage des données locales de la routine)

Chaque tâche possède sa propre STACK

FreeRTOS propose différentes stratégies de gestion de la HEAP, -> se référer à la documentation



Gestion des tâches

Le système d'exploitation, via l'API, fournit des fonctions permettant de gérer les tâches

- Durée de vie
  - Création
  - Destruction
- Contrôle de l'ordonnancement
  - Introduction d'un délai d'attente
  - Modification/relecture de la priorité
  - Suspension de l'exécution

Création d'une tâche

`xTaskCreate()`

Paramètres	
pvTaskCode	Pointeur vers la routine de la tâche
pcName	Chaîne de caractère spécifiant le nom de la tâche (optionnelle, utilisée pour le débogage)

Paramètres	
usStackDepth	Taille de la pile associée à la tâche
pvParameters	Pointeur vers une structure de paramètres passés à la routine de la tâche
uxPriority	Priorité initiale de la tâche
pvCreatedTask	Pointeur vers une variable contenant l’identifiant de la tâche après création, au retour de la fonction

Exemple de création de tâche en FreeRTOS Vanilla

L'exemple ci-dessous est codé suivant les fonctions FreeRtos sans modifications tierces, qu'on désigne sous le jargon "Vanilla". FreeRTOS pour l'ESP32 est en partie modifié spécifiquement pour cette cible, de même pour le STM32 avec la surcouche CMSIS

```
int main(  void )
{
    xTaskHandle hTask1;

    xTaskCreate( Task1, "Sample Task",
    1024, NULL, 2, &hTask1);        // Création de la tâche
  ...                               // taille de Stack 1024 , priorité 2
                                    // Pas de passge de structures de paramètres par pointeur
                                    // pointeur vers variable contenant l'identifiant de la
tâche

    vTaskStartScheduler();          // Démarrage de l'ordonnanceur

    return 0;
}

static void Task1( void * pvParameter)   // routine de la tâche
{
    int i;        // variable déclarée sur la pile de la tâche
    for(;;)       // boucle infinie
    {
      ...
    }
}
```

Destruction d’une tâche

`vTaskDelete()`

Paramètres	
pxTask	Identifiant de la tâche à détruire

Rappel : la libération des ressources n'est pas immédiate, la tâche passera dans l'état supprimée et quand la tâche de plus faible priorité IDLE sera exécutée, la memoire sera libérée.

Exemple de destruction d'une tâche FreeRTOS Vanilla

```
int main( void )
{
    ...
    xTaskCreate( Task1, "Task 1",
    1024, NULL, 2, NULL);               // Création de la tâche 1
    ...
}
static void Task1( void * pvParameter)
{
    xTaskHandle hTask2;
    xTaskCreate( Task2, "Task 2",
    1024, NULL, 3, &hTask2);           // création de la tâche 2
    // Do some work
    ...
    vTaskDelete( hTask2);              // Destruction de la tâche 2
    // Do some work
    ...
    vTaskDelete( NULL);                // Destruction de la tâche 1
}

static void Task2( void * pvParameter)
{
    for(;;)
    {
    ...
    }
}
```

Gestion du temps

Une tâche peut se placer dans l'état bloqué pendant une durée spécifiée en appelant `vTaskDelay`

Paramètres	
xTicksToDelay	Délai

Tous les délais spécifiés dans les fonctions FreeRTOS sont exprimés en « tick »

- Un « tick » correspond à la période de l'horloge système
  - Cette horloge, servant de base de temps à l'ordonnanceur, est générée à partir d'un périphérique de type timer.
  - L'horloge système émet des « system tick ».
- La valeur du « tick » est dépendante du matériel et du contexte de mise en œuvre du système temps-réel