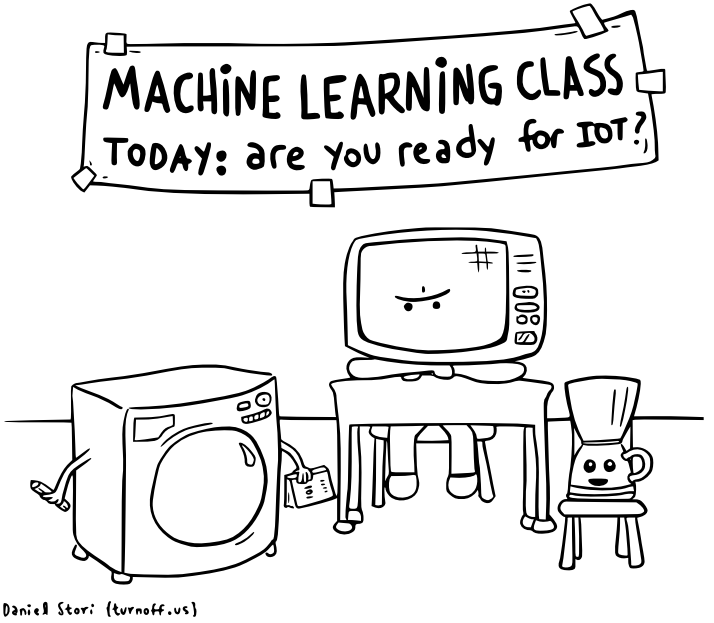
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE VAL DE LOIRE

4A STI



Mobilité - OS Embarqué — P-F. Bonnefoi

Version du 20 mars 2022

Qu’est-ce qu’un système embarqué ?

Quels périphériques sont présents ?

CPU

- exécution du code ;
- tout le reste est externe : mémoire RAM d'exécution, mémoire contenant le programme ;

Micro Contrôleur

- CPU ;
- périphériques intégrés : un peu de mémoire RAM, un contrôleur d'interruptions, un timer, de l'EPROM pour contenir le programme ;

SoC, «*System-on-a-Chip*» : un «*Core*» (CPU nouvelle génération) et de **nombreux** périphériques.

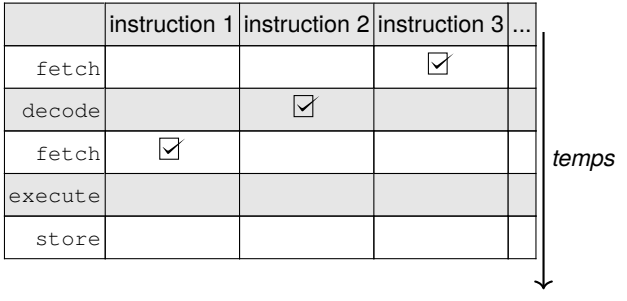
CPU Core	Unité programmable
MMU	Gestion de la mémoire virtuelle nécessaire pour un OS «*High End*»
DSP	Analyse de signal
Power Consumption	Batterie, génération de chaleur
Peripherals	A/D, UART, MAC, USB, Bluetooth, WiFi
Built-in RAM	vitesse et simplicité
Built-in cache	vitesse
Built-in EEPROM or FLASH	mise à jour en exploitaiton, «Field upgradeable»
JTAG Debug Suppot	Deboguage matériel
Tool-Chain	Compilateur, débogueur, ...

Différents usages

- **Application** : processeurs 32 ou 64 bits, permet de faire des calculs poussés (présence de DSPs), du multi-média, peuvent faire tourner des OS comme Linux.
- **Temps réel** : contrôle de moteurs, robotique : latence basse et sûreté de fonctionnement élevée. Adaptés à des routeurs réseau, des lecteurs multimédias où les données doivent être disponible à un instant donné ;
- **Micro-contrôleur** : gestion de matériel (fournis comme «*softcore*» dans des FPGAs), dépourvu de MMU (pas de Linux) mais intègrent de la mémoire et des périphériques.

- «**von Neumann**» : données et programme sont accédés par le même bus de données et le même bus d'adresse :
  - le CPU nécessite moins de broches d'E/S et est plus facile à programmer ;
  - le programme peut être mis à jour après déploiement⟹problème de sécurité ;
- «**Harvard**» : les données et le programme utilisent des bus différents :
  - très populaire avec les DSPs, «Digital Signal Processor», utilisant des instructions «Multiply-accumulate» où les opérandes de la multiplication sont au choix :
    - ⋆ une constante en provenance du programme ;
    - ⋆ une valeur fournie par le calcul précédent ou bien en provenance d'un convertisseur A/D ;

    L'architecture Harvard permet de récupérer cette constante et cette valeur **simultanément**.
- **RISC**, «*Reduced Instruction Set Computing*», *vs* **CISC**, «*Complex Instruction Set Computing*» :
  - CISC : une instruction peut réaliser une opération complexe mais elle nécessite plus de cycles d'horloge ;
  - RISC : une instruction peut être plus simple et s'exécuter plus rapidement mais il en faut plusieurs pour réaliser la même opération complexe ;
  - la programmation en assembleur est plus complexe en RISC, mais l'utilisation de compilateur rend la programmation directe en assembleur inutile dans la plupart des cas.
- exploitation de l'effet «**pipeline**» :
  - une instruction se décompose en plusieurs étapes : chercher l'instruction, la décoder, chercher les opérandes, exécuter l'instruction, stocker le résultat.
  - pour utiliser les bus de manière plus efficace, le CPU peut réaliser les différentes étapes sur différentes instructions :



- «***endianness***» : «little-endian» *vs* «big-endian» : exemple sur 32bits, soient 4 octets :

@	+1	+2	+3
78	56	34	12

vs

@	+1	+2	+3
12	34	56	78

- **Interrupt Controller** : gérer les différentes interruptions et leur priorités ;
- **DMA**, «*Direct Memory Access*» : bouger des zones mémoires indépendamment du processeur :
  - «*burst-mode*» : le circuit DMA prend le contrôle complet du bus aux dépends du CPU ;
  - «*cycle-stealing*» : négociation entre le DMA et le CPU ;
  - «*transparent*» : le DMA n'utilise le bus que lorsque le CPU ne l'utilise pas ;
- **MAC**, «*Medium Access Control*» : contrôle la couche 2 d'une interface réseau ;
- convertisseur **A/D** : numérise une valeur analogique en une valeur numérique suivant une résolution de 10 à 12 bits (avec un taux bas d'échantillonnage et un fort *jitter*.
- **UART**, «*Universal Asynchronous Receive/Transmit*» : liaison série de faible vitesse (par exemple RS232), en général de 9600 baud à 115200 baud/s avec des données sur 8bits, pas de contrôle hardware et 1 bit stop : «57600 N 8 1».
  3 fils pour relier deux appareils : le GND partagé, la broche TX de l'un reliée à la broche RX de l'autre et vice-versa.
- **USB** : liaison série haut débit, offrant différents «*Device Classes*» : périphérique HID, «*Human Interface Device*» : clavier/souris, tunnel TCP/IP, mémoire de masse, son *etc.* USB OTG, «*On The Go*», permet dávoir le rôle de maître ou de périphérique.
- **CAN**, «Controller Area Network» : bus inventé par Bosch pour les communications entre les différents circuits dans une voiture et utilisé dans les usines, entre des capteurs, *etc.*
- **WiFi** : échange continue d'information : débit élevé et données de taille quelconque mais consommateur d'énergie. l'antenne peut être externe ou incorporée dans le PCB, «printed circuit board» du circuit ;
- **Bluetooth**, BLE, «*Bluetooth Low Energy*» : échange intermittent d'information : faible débit de données réduites mais avec une très faible consommation.

- **bus** :
  - I^2C, SPI communication intelligente de données entre composants électroniques : associés à de la mémoire locale sur le périphérique : décharge le CPU de la gestion d'interruptions de composants disposant de leur propre rythme de fonctionnement (mesure de température, écran, autre CPU *etc.*) ;
  - GPIOs, «*General Purpose I/O*» : PWM, «*Pulse Width Modulation*» : contrôle de périphérique/moteur/radio (télécommande en 433Mhz, ou IR), , «*bit-banging*» : émulation de bus exotique ou connexion directe de composant (détecteur PIR de mouvement, interrupteurs *etc.*)

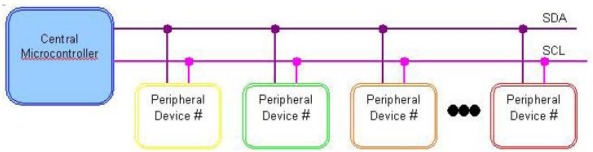
- **RTC**, «*Real Time Clock*» : maintenir l'heure et la date (utilisation d'une batterie séparée).
  Si le composant est «connecté» il peut utiliser un serveur NTP, «*Network Time Protocol*».

- **Timers** : compteur incrémentés ou décrémentés en fonction du temps gérés de manière indépendante du CPU
  - «*watchdog timer*» : un compteur qui doit être réinitialisé, «*kicked*», de manière logicielle avant qu'il n'atteigne zéro $\Longrightarrow$ s'il atteint zéro, le CPU subit un reset : l'idée est qu'il est dans une boucle infinie ou bien dans un interblocage ;
  - «*fast timers*» : mesurer la longueur d'impulsion ou pour les générer (PWM) ;

- **Memory controller** : obligatoire pour la DRAM, «dynamic RAM» : rafraîchissement de la mémoire de manière régulière (souvent intégré au CPU). Gérer la mémoire FLASH persistente.

- **co-processeur cryptographique** : réaliser des opérations de chiffrement/déchiffrement et signature avec des algorithmes symétriques et surtout asymétriques (coutêux pour le CPU).
  Embarque des clés de chiffrement qui peuvent être figées dans sa mémoire
  (exemple ATECC608 : propose du chiffrement sur courbe elliptique).

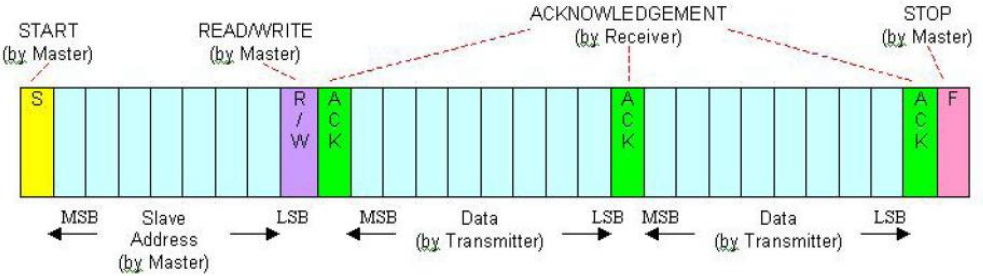
- système de **localisation satellitaire** : GPS américain, Glonass russe, Beidou chinois et Galileo européen.
  Permet de disposer de la position et de l'heure à une précision de 50 ns.



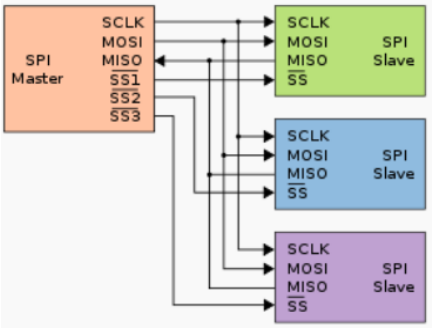
Le bus I2C, «*Inter-Integrated Circuit*» :

- ⋆ un bus générique proposé par Philips dans les années 80, beaucoup utilisé dans les télévisions ;
- ⋆ synchrone ;
- ⋆ **débit** : jusqu'à 400 Kbps ;

- ⋆ seulement 2 signaux :
  - ◇ `SCL`, «*Signal Clock*» : le contrôleur «Master» génère l'horloge ;
  - ◇ `SDA`, «*Signal Data*» : le «Master» transmets les informations et le «Slave» transmets l'acquittement : si aucun acquittement n'est reçu la communication peut être stoppée ou réinitialisée.



- ▷ plusieurs «*Slaves*» peuvent être connectés au même bus ;
- ▷ chaque *Slave* doit disposer d'une **adresse** sur 8bits, composée de :
  - ◇ une partie fixe qui dépend du constructeur ;
  - ◇ une partie configurable ;
  - ◇ le dernier bit qui définit le sens de la communication : 0 pour écrire et 1 pour lire ;
  - ◇ les communications commencent par un bit de début, «*start bit*», suivi de l'adresse sur 8 bits, le bit d'acquittement, un octet de donnée, un autre bit d'acquittement and à la fin un bit d'arrêt



- un bus générique proposé par Motorola dans les années 80 ;
- communications :
  - full duplex ;
  - synchrone ;
  - lien «Master/Slave» : c'est le master qui initie le transfer des trames de données ;
  - plusieurs liens simultanés possibles : un fil par slave permet de sélectionner celui avec lequel on veut communiquer ;
- **Débit** : quelques dizaines de Mbps ;

- 4 signaux :
  - `SCLK`, «*Clock*» : l'horloge obligatoire pour la transmission synchrone ;
  - `MOSI`, «*Master Out Slave Input*» : communication Master ⟹ Slave ;
  - `MISO`, «*Master Input Slave Output*» : communication Slave ⟹ Master ;
  - `SS`, «*Slave Select*» : un fil par Slave pour pouvoir le sélectionner ;

«SPI *vs* I2C» : Quel bus choisir ?

- le bus SPI permet des débits plus rapides ;
- le bus I2C ne nécessite que 2 fils **mais** nécessite un protocole de communication plus complexe : adressage, définition de trames, gestion de l'acquittement.

	SPI	I2C
Application	Better suited for data streams between processors	Occasional data transfers. Generally used for slave configuration
Data rates	>10 Mb/s	< 400 kb/s
Complexity	3 bus lines More wires more complex wiring More pins on a chip	Simple, only 2 wires Complexity does not scale up with number of devices
Addressing	Hardware (chip selection)	Built-in addressing scheme
Communication	No acknowledgment mechanism, Only for short distances	Better data integrity with collision detection, acknowledgment mechanism, spike rejection
Specification	No official specification	Existing official specifications
Licensing	free	free

Les différents types de mémoire

- la mémoire **non volatile** où des données peuvent être stockées et où elles resteront mémorisées malgré les resets et extinction du module :
  - Flash : l'espace programme où le firmware est stocké ;
  - EEPROM : utilisé pour des données utilisateurs
- SRAM ou «*Static Random Access Memory*» : où les les variables sont créées et manipulées lors de l'exécution du programme, «*at runtime*» ;

La quantité mémoire disponible suivant la «board» utilisée

ATMega328 (UNO)	⋆ Flash 32Ko (0,5Ko utilisé par le «*bootloader*») ; ⋆ SRAM 2Ko ; ⋆ EEPROM 1Ko.
ATMega2560 (MEGA)	⋆ Flash 256Ko (8Ko utilisé par le «*bootloader*») ; ⋆ SRAM 8Ko ; ⋆ EEPROM 4Ko.

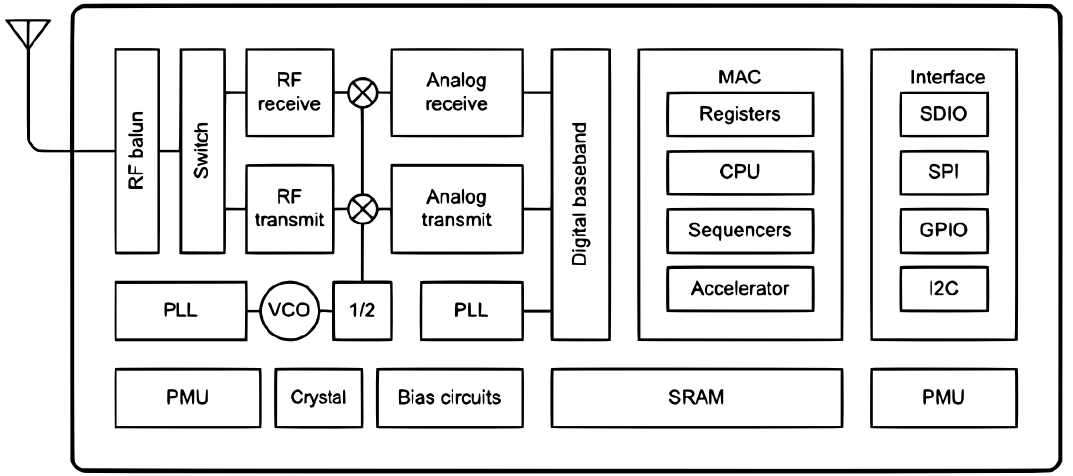
Consommation de mémoire et programmation

```
1 char *mon_texte = "Super la programmation sur Arduino !";
```

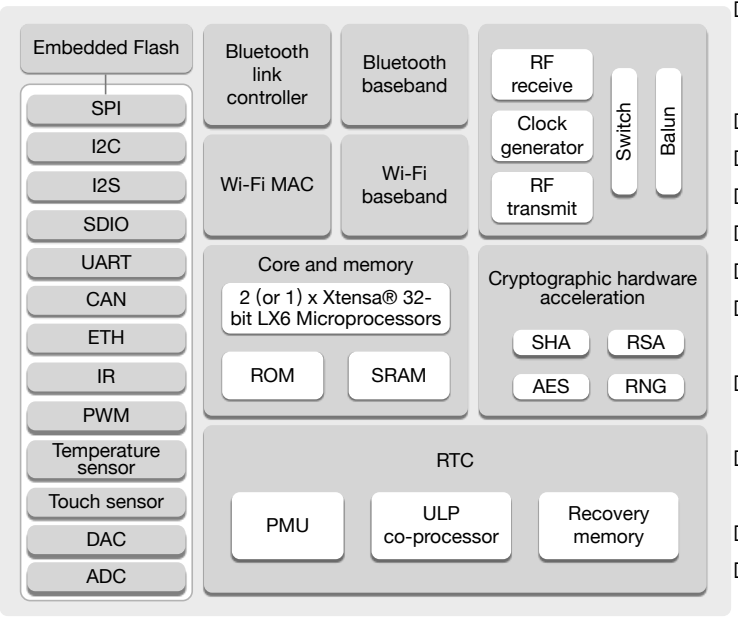
Cette déclaration de $36 + 1$ caractères consomme de la SRAM qui est limitée à 2048 octets sur un Arduino Uno.

Il est possible de loger cette chaîne de caractères dans la **mémoire Flash** grâce à l'utilisation du mot-clé `PROGMEM`.

La documentation est disponible à l'adresse : `http://arduino.cc/en/Reference/PROGMEM`



- ESP8266 SoC by Shanghai-based Chinese manufacturer, Espressif Systems ;
- CPU : Tensilica Xtensa L106 : 32bits à 80/160MHz, architecture Harvard modifiée ;
- 64Ko instruction, 96Ko données, FLASH externe de 512ko à 4Mo ;
- consommation : 3,3v 215mA ;
- WiFi b/g/n mode STA ou AP ;
- timers, deep sleep mode, JTAG debugging ;
- GPIO (16), PWM (3), A/DC 10 bits (1), UART, I^2C, SPI, PMU «*Power management unit*».



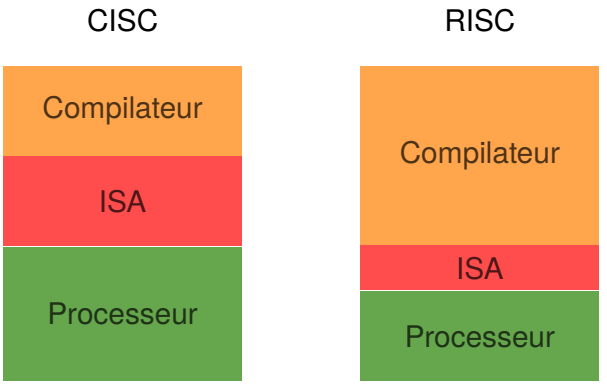
- Xtensa® single/**dual-core** 32-bit LX6 microprocessor(s), up to 600 DMIPS
  (200 DMIPS for single-core mi- croprocessor)
- **448 kB** ROM, **520 kB** SRAM, **16 kB SRAM** in RTC
- QSPI flash/SRAM, up to 4 x 16 MB
- Power supply: 2.3V to 3.6V
- Internal 8 MHz oscillator with calibration
- Internal RC oscillator with calibration
- External 2 MHz to 60 MHz crystal oscillator
  (40 MHz only for Wi-Fi/BT functionality)
- External 32 kHz crystal oscillator for RTC with calibration
- Two timer groups, including 2 x 64-bit timers and 1 x main watchdog in each group
- RTC timer with sub-second accuracy
- RTC **watchdog**

- 12-bit SAR ADC up to 18 channels
- 2 × 8-bit DAC
- 10 × touch sensors
- Temperature sensor
- 4 × **SPI**, 2 × I2S, 2 × **I2C**, 3×**UART**
- 1 host (SD/eMMC/SDIO), 1 slave (SDIO/SPI)
- **Ethernet MAC** with DMA and IEEE 1588 support

- **CAN 2.0**
- IR (TX/RX)
- Motor PWM
- LED PWM up to 16 channels
- **Hall sensor**
- Ultra-low-noise analog pre-amplifier

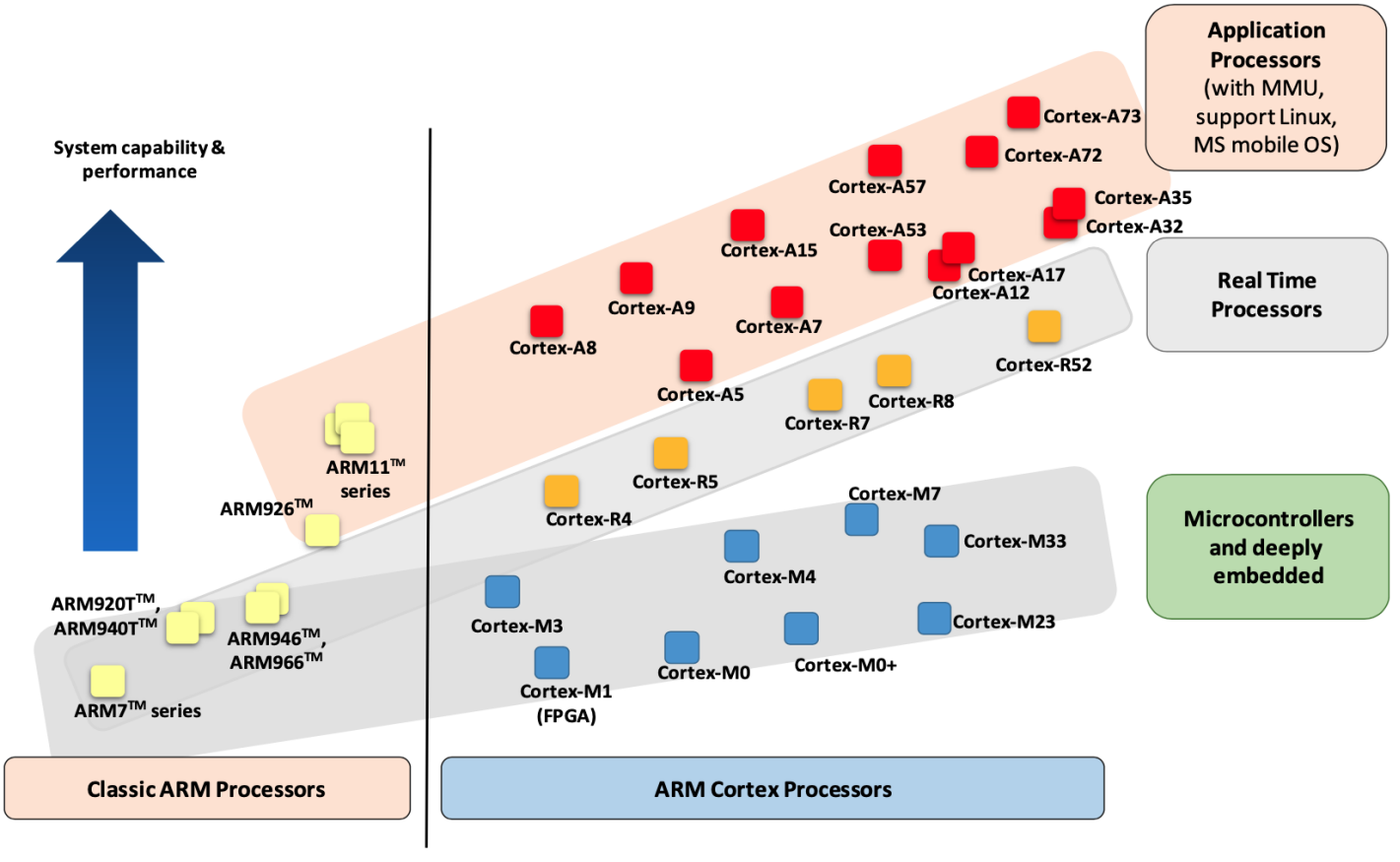
Qu'est-ce que «*RISC-V*» ?

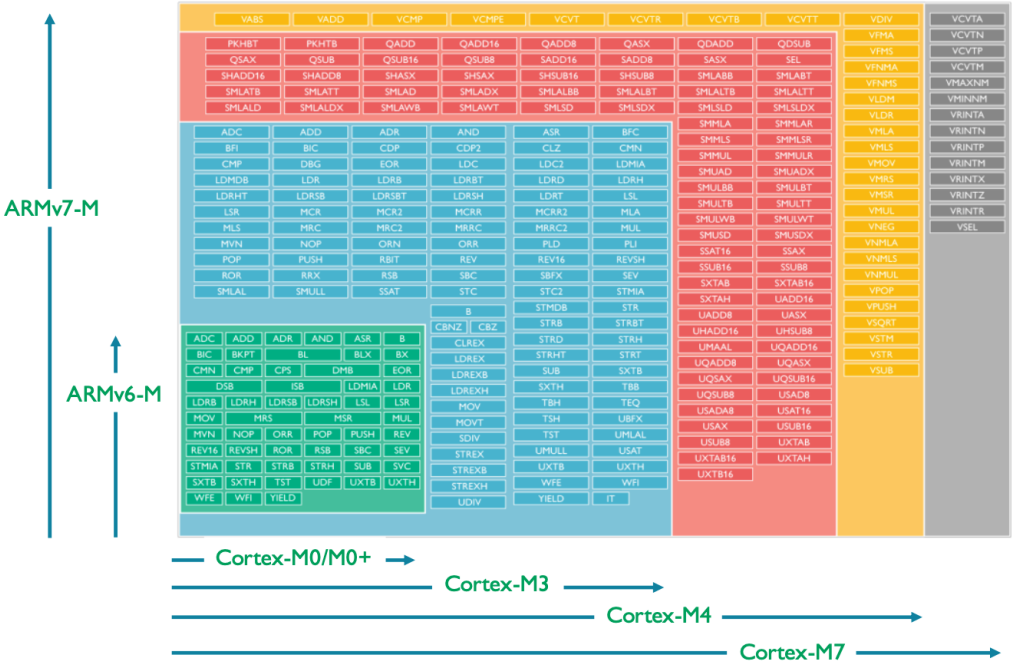
- «*Open Standard Instruction Set*», ISA ;
- basé sur les principes RISC, «*Reduced Instruction Set Computer*» ;



- licence «*Open Source*» sans royalties, mais des extensions payantes... ;
- supporté par :
  - différentes entreprises au niveau de l'offre hardware : sifive ;
  - différents OS : Linux, FreeRTOS ;
  - différents «*toolchains*» : compilateur, linkeur, constructeur de firmware ;
  - sous formes de différentes «*IPs*» pour FPGA, «*Field Programmable Gate Array*».

ARM Cortex et ISA





Floating Point

DSP (SIMD, fast MAC)

Advanced data processing
bit field manipulations

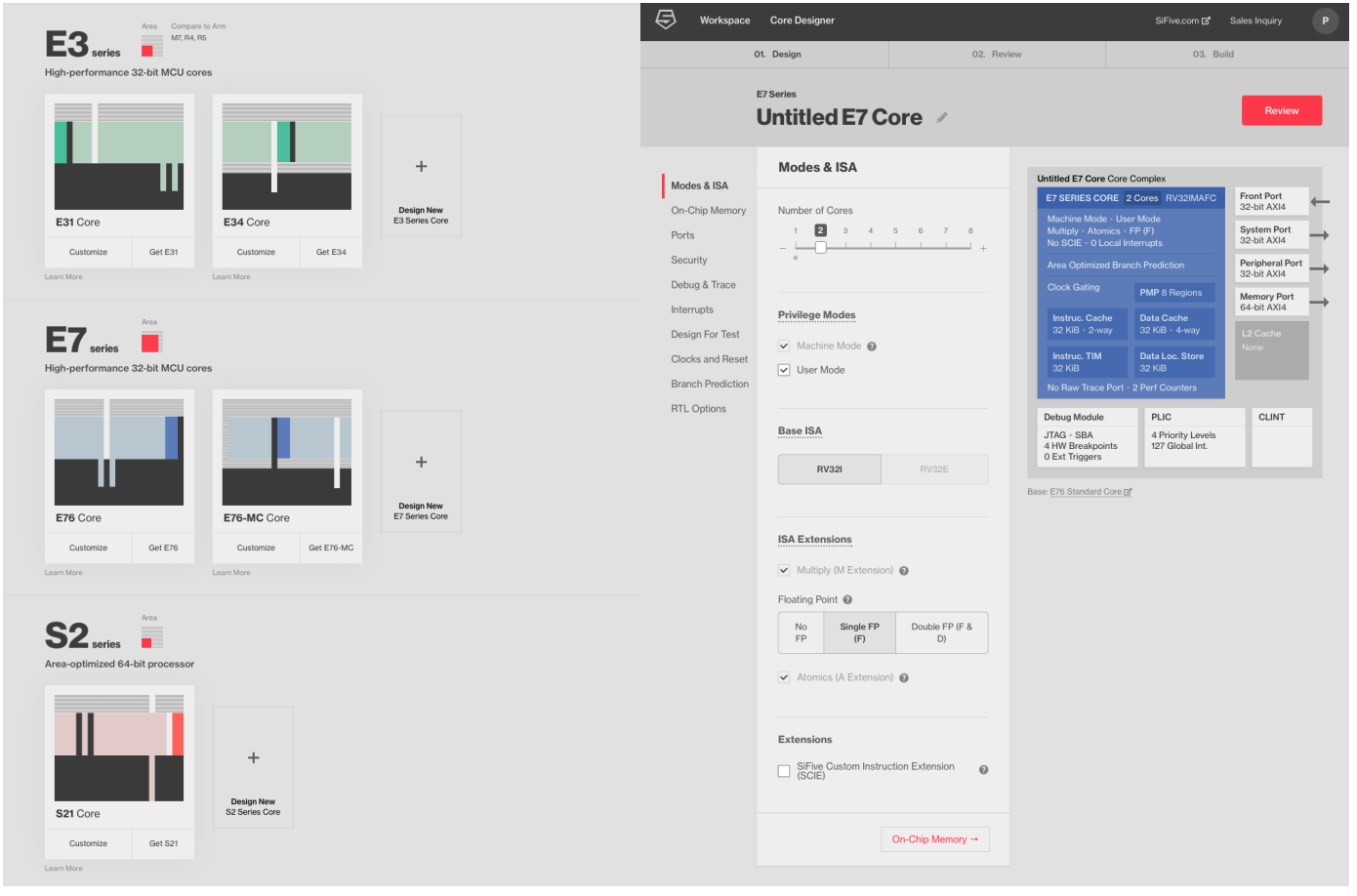
General data processing
I/O control tasks

Name	Description	Version	Status	Instruction count
	Base			
RVWMO	Weak Memory Ordering	2.0	Ratified	
RV32I	"Base Integer Instruction Set 32-bit"	2.1	Ratified	49
RV32E	"Base Integer Instruction Set (embedded) 32-bit 16 registers"	1.9	Open	49
RV64I	"Base Integer Instruction Set 64-bit"	2.1	Ratified	14
RV128I	"Base Integer Instruction Set 128-bit"	1.7	Open	14
	Extension			
M	Standard Extension for Integer Multiplication and Division	2.0	Ratified	8
A	Standard Extension for Atomic Instructions	2.1	Ratified	11
F	Standard Extension for Single-Precision Floating-Point	2.2	Ratified	25
D	Standard Extension for Double-Precision Floating-Point	2.2	Ratified	25
Zicsr	Control and Status Register (CSR)	2.0	Ratified	
Zifencei	Instruction-Fetch Fence	2.0	Ratified	
G	*"Shorthand for the IMAFDZicsr Zifencei base and extensions intended to represent a standard general-purpose ISA"*	N/A	N/A	
Q	Standard Extension for Quad-Precision Floating-Point	2.2	Ratified	27
L	Standard Extension for Decimal Floating-Point	0.0	Open	
C	Standard Extension for Compressed Instructions	2.0	Ratified	36
B	Standard Extension for Bit Manipulation	0.93	Open	42
J	Standard Extension for Dynamically Translated Languages	0.0	Open	
T	Standard Extension for Transactional Memory	0.0	Open	
P	Standard Extension for Packed-SIMD Instructions	0.2	Open	
V	Standard Extension for Vector Operations	0.10	Open	186
N	Standard Extension for User-Level Interrupts	1.1	Open	3
H	Standard Extension for Hypervisor	0.4	Open	2
Zam	Misaligned Atomics	0.1	Open	
Ztso	Total Store Ordering	0.1	Frozen	

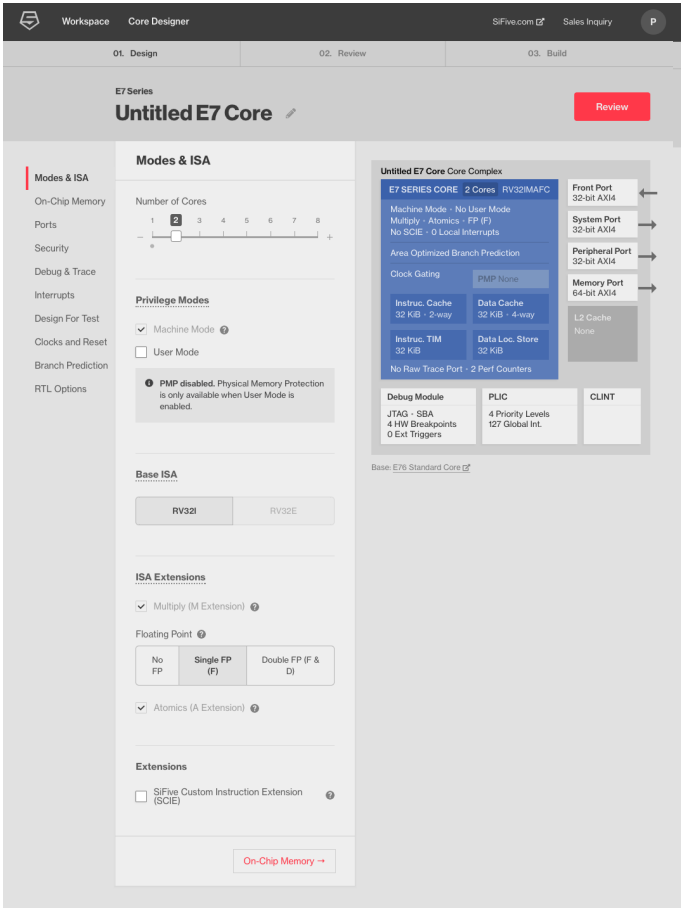
Floating point operation

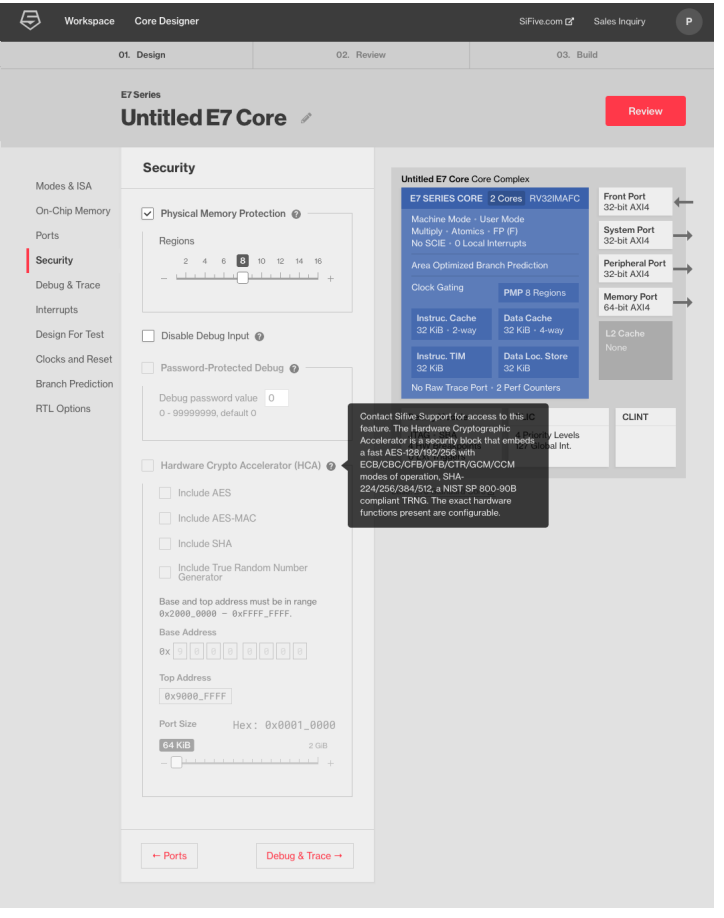
Le choix des extensions peut amener à des coûts supplémentaires...

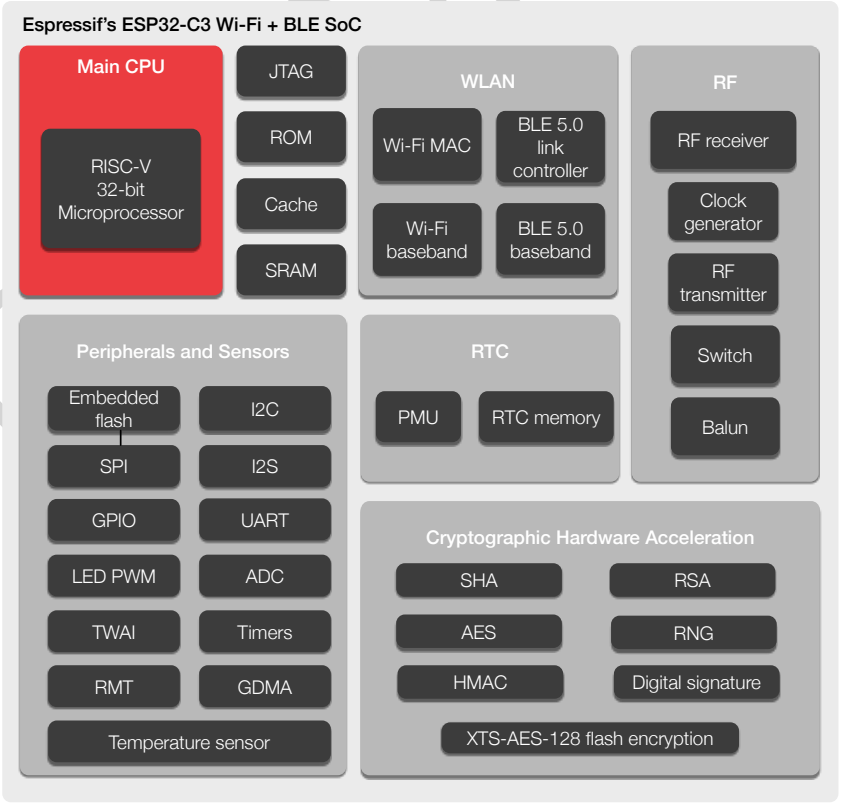
La configuration pour la production d'un processeur sur *scs.sifive.com*



La configuration pour la production d'un processeur sur *scs.sifive.com*





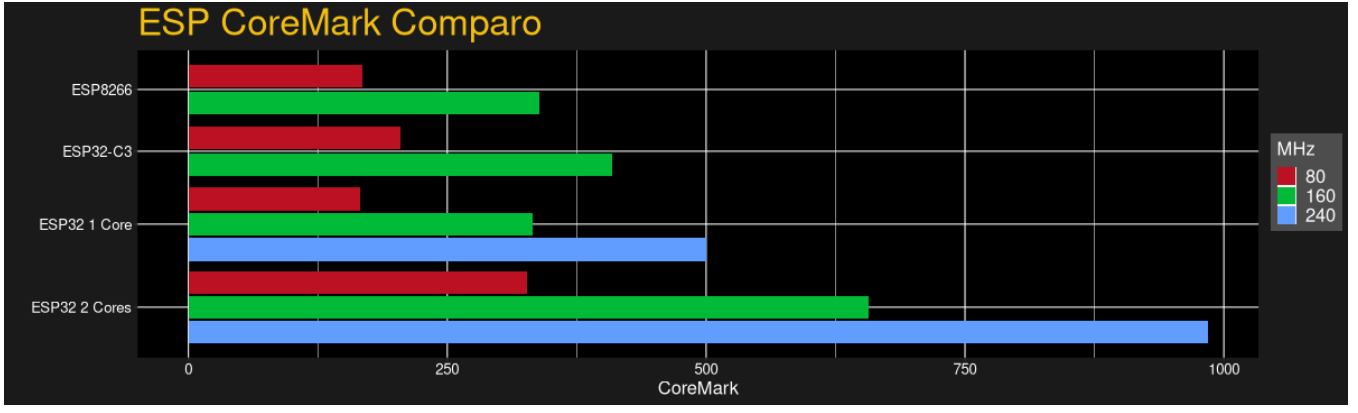


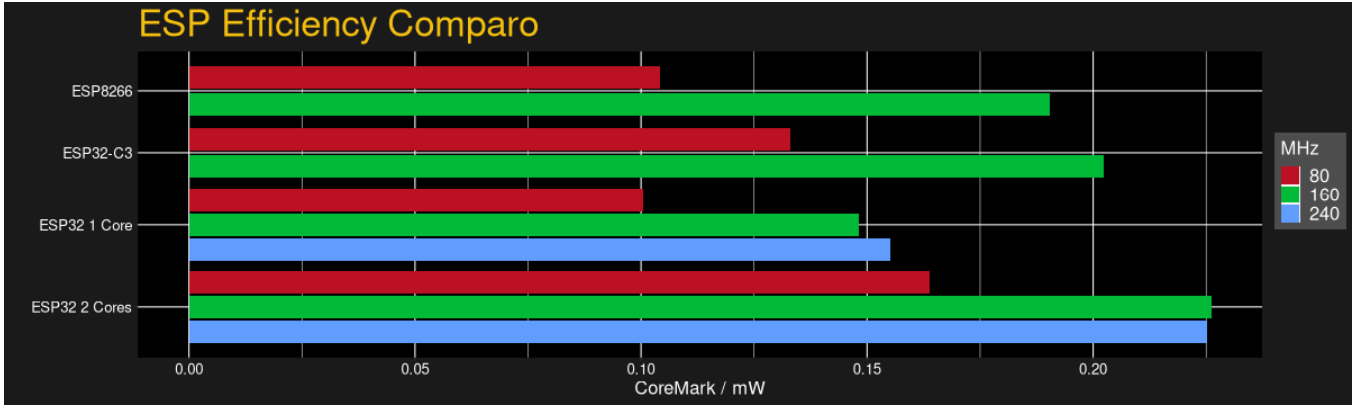
- ☐ A complete **Wi-Fi subsystem** that complies with IEEE 802.11b/g/n protocol and supports Station mode, SoftAP mode, SoftAP + Station mode, and promiscuous mode
- ☐ A **Bluetooth LE subsystem** that supports features of Bluetooth 5 and Bluetooth mesh
- ☐ State-of-the-art power and RF performance
- ☐ **32-bit RISC-V single-core processor** with a four-stage pipeline that operates at up to 160 MHz
- ☐ **400 KB of SRAM** (16 KB for cache) and **384 KB of ROM** on the chip, and SPI, Dual SPI, Quad SPI, and QPI interfaces that allow connection to **external flash**
- ☐ Reliable **security features** ensured by :
  - ◇ **Cryptographic hardware accelerators** that support AES-128/256, Hash, RSA, HMAC, digital signature and secure boot
  - ◇ **Random number generator**
  - ◇ **Permission control** on accessing internal memory, external memory, and peripherals
  - ◇ **External memory encryption** and decryption
- ☐ Rich set of peripheral **interfaces** and **GPIOs**, ideal for various scenarios and complex applications.

ESP32-C3 family has a low-power **32-bit RISC-V single-core microprocessor** with the following features:

- ◇ **RV32IMC** ISA (*voir table sur transparent précédent*) ;
- ◇ 32-bit **multiplier** and 32-bit **divider**
- ◇ up to **32 vectored interrupts** at seven priority levels
- ◇ up to **16 PMP regions** «*Physical Memory Protection*».

ESP32-C3 : comparaisons avec les autres ESPs





La programmation ?

Embarqué *vs* IoT : c'est la même chose, mais l'IoT utilise toujours une pile TCP/IP.

Aucun système d'exploitation : «*polling*» uniquement

Avant de démarrer le programme :

- ▷ construction du programme : compilation, édition de liens et localisation en mémoire ;
- ▷ utilisation d'un «chargeur» : copie du programme en mémoire, bloquer les interruptions, initialiser les zones mémoire pour les données, préparer la pile et les pointeurs de pile.

Conception basée sur une **boucle infinie** :

```
 1 int main()
 2 {
 3    for (;;)
 4    {
 5       TravailA();
 6       TravailB();
 7       TravailA();
 8       TravailC();
 9    }
10 }
```

- ▷ chaque fonction correspond à un «*processus*» ;
- ▷ ordonnancement«*round robin*», ou *tourniquet* ;
- ▷ ligne 3 : boucle infinie ;
- ▷ ligne 5&7 : la fonction «`TravailA`» obtient le CPU à des intervalles plus courts que les autres fonctions;
- ▷ chaque «*processus*» réalise la lecture des entrées quand elle a du temps : «*polling*».

Le «polling» : source potentielle de problèmes lors de l'intégration

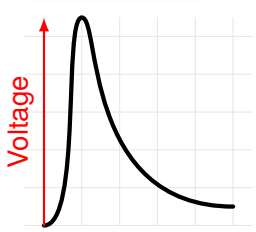
Boucle de «*polling*» utilisée pour surveiller une entrée qui ne s'arrête que lorsque l'entrée passe d'un état à un autre :

- ▷ *Si cette boucle est intégrée dans* `TravailB`*, alors le résultat peut être catastrophique pour les fonctions* `TravailA` *et* `TravailC`*: elles ne s'exécuteront que lorsque le changement d'état interviendra !*
- ▷ *Et si le changement d'état dépend d'une opération réalisée par* `TravailA` *ou* `TravailC` *alors on peut aboutir à un* ***deadlock*** *!*
- ▷ *Dans tous les cas, on fait de l'****attente active*** *ou* «busy waiting» *qui* ***gaspille de l'énergie****, car le CPU ne peut se mettre en veille et économiser son énergie.*

Exemple : traitement d'un capteur PIR, «*Passive InfraRed sensor*»

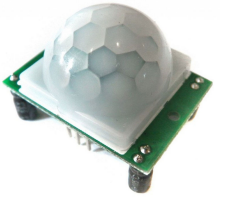
Capteur PIR : capteur électronique qui mesure la **quantité de lumière infrarouge** diffusée par les objets dans son champs de vision.

Quand une personne passe entre le capteur et un mur par exemple : la température à cet endroit dans le champs de vision du capteur passe de la température de la pièce à la température du corps, puis revient à la température de la pièce.



$\Longrightarrow$ application à la détection de mouvement : on détermine la différence entre la mesure «*habituelle*», température de la pièce, et un changement soudain, température du corps.

$\Longrightarrow$ le capteur convertit le changement dans la quantité de lumière infrarouge qu'il reçoit en un changement de son **voltage de sortie**.



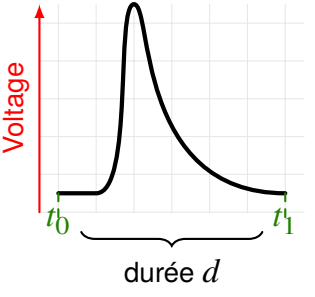
On peut ajouter un «*dôme*» sur le capteur afin d'augmenter son champs de vision.

Comment lire la valeur fournie par le capteur ?
$\Longrightarrow$en utilisant l'ADC, le convertisseur analogique/numérique du micro-contrôleur.
À quel rythme, «*fréquence*» faut-il *consulter* la mesure du capteur ?

Suffisamment souvent pour **détecter** le «*pic*» :

$$f = \frac{1}{d_m} \text{ avec } d_m < d/2 \text{ par exemple.}$$

La **lecture régulière** par le micro-contrôleur correspond à faire du «*polling*».
$\Longrightarrow$cette lecture doit être intégrée précisément dans une fonction de `travail` dont on garantit la **fréquence d'appel** en fonction de d_m.

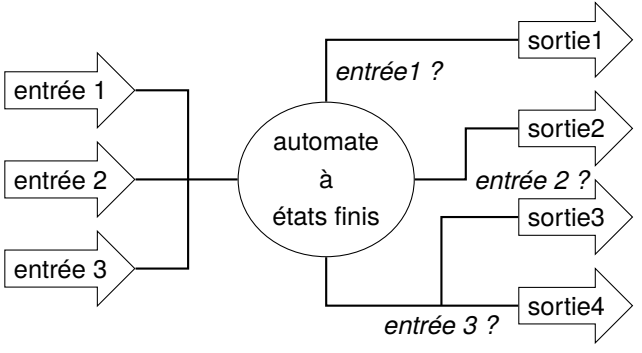


Machine à nombre d'états finis

```
1 int main()
2 {
3    for(;;)
4    {
5       lectureCapteurs();
6       extraireEvenements();
7       transitionEtatEvenement();
8    }
9 }
```

- ▷ ligne 5 : lecture des différentes entrées ;
- ▷ ligne 6 : analyse des entrées pour déterminer un événement : mesure supérieure à un seuil, bouton pressé...
- ▷ ligne 7 : traitement des événements : déclencher les actions à entreprendre ou changer d'état (évolution de l'analyse des entrées et déclencher de nouveaux événements).

Exemple d'utilisation d'un automate à états finis

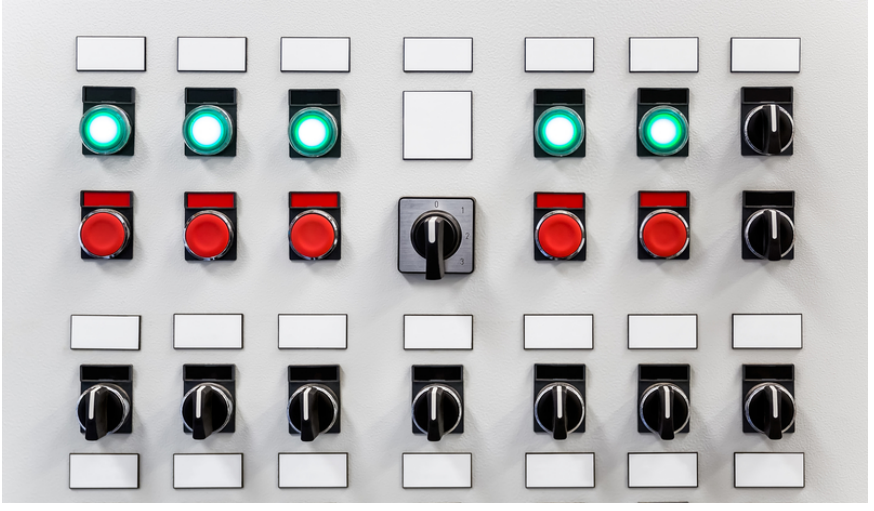


- □ événement sur «*entrée 1*» ⟹ «sortie 1» ;
- □ événement sur «*entrée 2*» ⟹ «sortie 2» ;
- □ événement sur «*entrée 3*» ⟹ «sortie 3» et «sortie 4» ;

Il est nécessaire de définir :

- ▷ *des **états** : ils permettent de mémoriser les événements déjas reçus dans le cas d'une combinaison de plusieurs événements à prendre en compte pour déclencher une opération ;*
- ▷ *des **transitions** : réception d'un événement ou gestion d'un compteur de temps (on compare une valeur mémorisée à une valeur courante et la différence indique l'intervalle de temps écoulé) ;*
- ▷ *des **actions** : le fait d'effectuer une transition peut déclencher une opération à réaliser sur une des sorties.*

Exemple : un panneau de contrôle



Il est constitué de :

- d'**indicateurs lumineux** : led contrôlées par le micro-contrôleur ;
- **boutons poussoirs** :
  - connectés à des entrées digitales du micro-contrôleur ;
  - ne conservent leur état que pendant la durée de la pression sur le bouton
- **sélecteurs rotatifs** :
  - connectés à des entrées digitales du micro-contrôleur ;
  - conservent leur état tant qu'il n'est pas changé par l'opérateur humain.

Comment combiner les différents boutons/indicateurs lumineux ?

- ▷ chaque état des boutons rotatifs constituent une partie d'**un état de l'automate fini** ;
- ▷ la **transition d'un état de l'automate fini** à un autre peut être «*visualisé*» par un indicateur lumineux ;
- ▷ la **détection de l'appui** d'un bouton poussoir nécessite de faire du **polling** pour détecter sa pression.

⟹**L'automate fini**, au contraire des fonctions de `travail` :

- ▷ représente le **comportement** du panneau de contrôle ;
- ▷ exprime les intéractions entre les capteurs, les décisions de l'opérateur humain

Par exemple: ◇ *un bouton rotatif peut servir à ignorer un capteur ou au contraire en activer l'utilisation ;*

◇ *un indicateur lumineux peut être lié à un capteur ;*

Les «co-routines»

- une «*co-routine*» peut être exécutée **plusieurs fois**, comme par exemple une fois par ressource identifiée ;
- une «*co-routine*» peut être **suspendue** pour laisser le CPU exécuter un autre traitement : elle conserve son état et peut reprendre son exécution là où elle s'était stoppée ;
- cette **suspension peut être invoquée** depuis la «*co-routine*» elle-même au profit d'une autre «*co-routine*» à l'aide de l'instruction «*yield*» et son exécution peut être reprise à l'aide de l'instruction «*resume*» ;
- il est possible de retourner des valeurs lors du «*yield*» et d'en recevoir lors du «*resume*».

Co-routine et Lua : la bonne façon d'en tirer partie

Version sans co-routines qui peut produire des crashes

```
1 while 1 do
2 gpio.write(3, gpio.HIGH)
3 tmr.delay(1000000) -- waits a second
4 gpio.write(3, gpio.LOW)
5 tmr.delay(1000000) -- and again
6 end
```

L'OS ne reçoit pas de temps pour lui avec `tmr.delay`

Version avec co-routines

```
1 flashDelay = 200 -- ms
2 function flasher()
3   while 1 do
4     gpio.write(3, gpio.HIGH)
5     coroutine.yield(flashDelay)
6     gpio.write(3, gpio.LOW)
7     coroutine.yield(flashDelay)
8   end
9 end
```

Le programme est appelé par `driveCoroutine(flasher)` en version «*good*» ou «*bad*» :

```
1  -- buggy one that will likely
2  -- crash the ESP8266
3  function driveCoroutineBad(proc)
4    co = coroutine.create(proc)
5    while 1 do
6      -- TODO: check bool here and end if appro
   priate
7      bool, time = coroutine.resume(co)
8      tmr.delay(time * 1000)
9    end
10 end
```

Ne donne pas de temps aux autres co-routines et à l'OS.

```
1  function driveCoroutineGood(proc)
2    co = coroutine.create(proc)
3    delay = 1
4    function resumeAfterDelay()
5      -- TODO: handle bool
6      bool, delay = coroutine.resume(co)
7      tmr.alarm(0, delay, 0, resumeAfterDelay)
8    end
9
10   resumeAfterDelay()
11 end
```

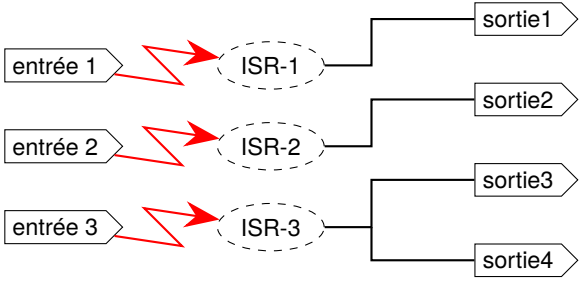
L'utilisation d'une alarme, `tmr.alarm` *donne du temps à l'OS.*

Les interruptions

Elles permettent d'éviter le «*polling*» : une **interruption** peut être générée lorsqu'une entrée change.

Une interruption correspond à un **changement d'exécution** du CPU :

- ▷ une broche d'E/S est associée à un numéro : «*interrupt number*» ;
- ▷ un tableau, le «*interrupt vector*», est stocké à un emplacement précis de la mémoire :
  - ◇ chaque **numéro** est associé à **l'adresse d'une fonction** appelée lors du déclenchement de l'interruption associée ;
  - ◇ chacune de ces **fonctions** est appelée une **ISR**, «*Interrupt Service Routine*» ;
  - ◇ lors d'une interruption, on appelle l'ISR :
    - ⋆ on sauvegarde les registres d'exécution de la fonction courante et on les empile sur la pile ;
    - ⋆ on peut activer ou non la prise en compte de nouvelles interruptions, «*nested*», éventuellement suivant des priorités.



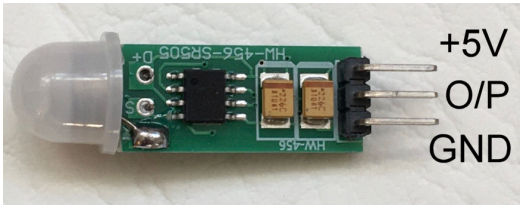
Ici, on a un traitement purement piloté par les interruptions :

- *__tout le travail__ lié à une entrée est effectué dans une ISR ;*
- *dans le cas où l'on __autorise__ le* «nested» *:*
  - ◇ *une interruption de __priorité supérieure__ doit connaitre l'état précis du système ;*
  - ◇ *il peut ne pas y avoir suffisamment de priorités pour gérer toutes les entrées.*
- *dans le cas où l'on __n'autorise pas__ le* «nested» *:*
  - ◇ *toutes les autres interruptions doivent attendre que la première soit terminée⟹ il peut y avoir des retards, «latency», sur les priorités les plus hautes.*

En général, plusieurs entrées déclenchent la même interruption et le premier travail de l'interruption est de trouver quelle est l'entrée qui a changé d'état : l'ordre de recherche défini une **sorte de priorité**.

Par exemple, lorsque l'on a besoin de traiter plusieur entrées sur une même interruption car il n'y a pas suffisamment d'interruptions disponibles pour traiter chaque entrée séparément.

le capteur PIR

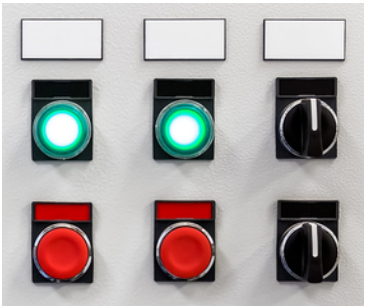


Le capteur PIR est associé à un circuit :

- ▷ réalise la comparaison d'un état à celui qui le suit pour **détecter un mouvement** ;
- ▷ **transmet un signal digital** sur une broche de sortie en cas de détection de mouvement.

La **sortie digitale** du capteur peut être connectée à une **entrée digitale** du micro-contrôleur.
Le micro-contrôleur peut associer une «*interruption*» lors de la modification de cette entrée digitale.
⟹ plus de «*polling*» nécessaire

Le panneau de contrôle



On peut :

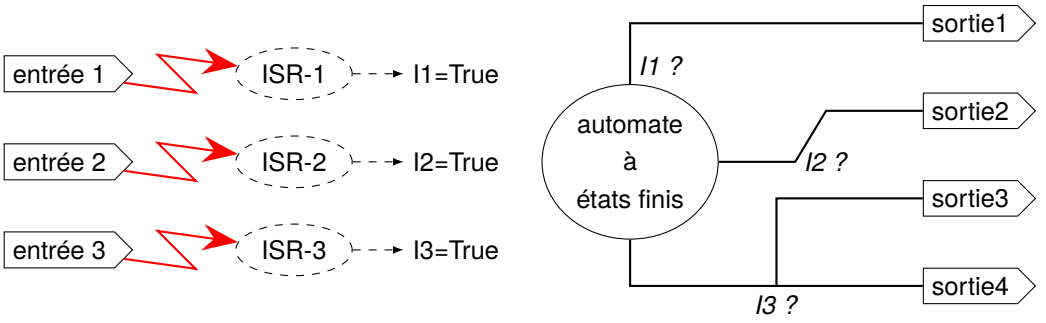
- ▷ associer une **entrée digitale** du micro-contrôleur à chaque bouton ;
- ▷ associer une **interruption** pour détecter la modification de chacun de ces boutons.

⟹ plus de «*polling*» nécessaire

En cas de **manque** d'entrée digitale/interruption, on peut **multiplexer** les différentes entrées digitales en une combinaison de bits `001100` où chaque bit est associé à un capteur/événement, puis au choix :

- ▷ les lire régulièrement ⟹ polling ;
- ▷ **déclencher une interruption** lors de leur modification, puis les lire pour déterminer les événements survenus.

Interruptions et automate à états finis

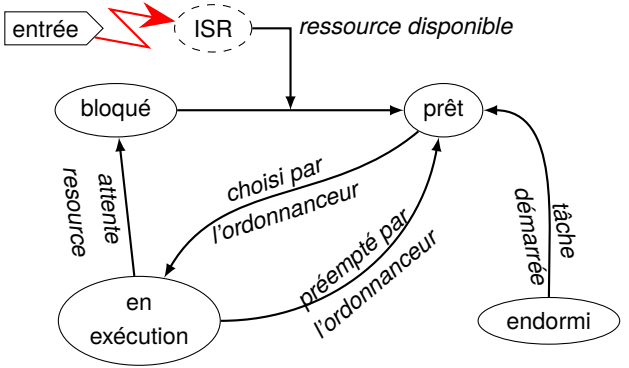


- ▷ *le traitement de l'interruption est rapide :*
  - ◇ *juste positionner un drapeau,* «flag»*, à vrai ;*
  - ◇ *faible latence pour le traitement des autres interruptions.*
- ▷ *c'est l'automate qui gère les priorités s'il y en a besoin.*

Attention

Il est possible de **choisir** comment est déclencher l'interruption :

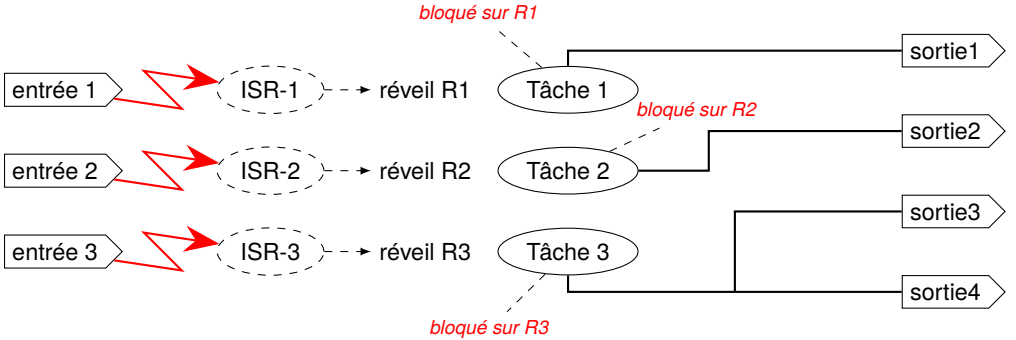
- ☐ «*level triggered*» : l'interruption se déclenche **tant que** le niveau (haut ou bas) sur la broche, ou «*pin*», est maintenu dans l'état associé à l'interruption $\Longrightarrow$ soit le matériel change le niveau au déclenchement de l'interruption, soit l'ISR doit changer le niveau, sinon l'interruption se **déclenchera de nouveau**.
- ☐ «*edge triggered*» : l'interruption ne se déclenche que lors de la **transition d'un niveau à l'autre**, c-à-d sur à la bordure montante ou descendante de l'impulsion.
  Si les interruptions n'étaient pas actives lors de ce **court instant**, alors **on n'obtiendra pas d'interruption** (à moins que le système la mémorise pour nous).



On dispose d'un SE, «Système d'Exploitation», qui gère les différentes tâches et alloue le CPU entre elles :

- «endormi» : la tâche est créée mais elle n'est pas encore démarrée : elle sera démarrée par le programme.
- «prêt» : la tâche peut être exécutée, mais elle *attend* le CPU ;
- «en exécution» : la tâche s'exécute, elle *possède* le CPU ;
- «bloqué» : la tâche est en attente d'un événement extérieur : une interruption liée à une entrée ou bien un événement réseau.

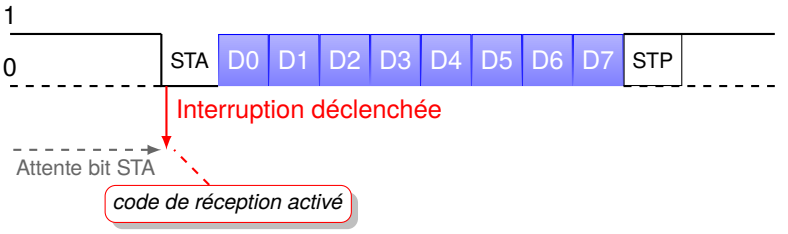
- ordonnanceur et préemption : si le noyau supporte la préemption, une tâche est **suspendue** après un «*timeslice*» : tous les registres sont sauvegardés et un **changement de contexte** est réalisé (dans le cas d'une interruption, on peut ne sauvegarder que les registres utilisés par l'ISR).

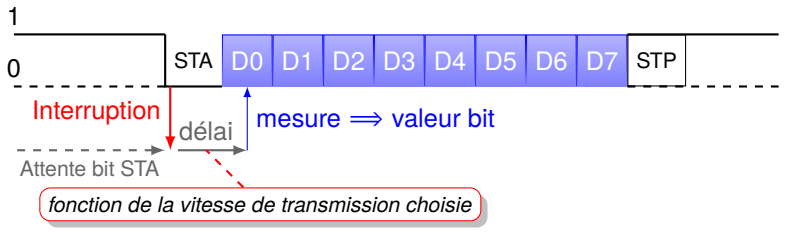


Chaque tâche est suspendue jusqu'à ce que le SE la réveille lorsque un événement se produit au travers d'une ISR.
Les tâches peuvent êtres synchronisées par des **sémaphores**, et communiquées entre elles par des «***message queues***».

Réception asynchrone : le port série

Le récepteur active une **interruption** qui s'active lors du passage du niveau haut au niveau bas : il exécute le **code de réception** lors du déclenchement de l'interruption :

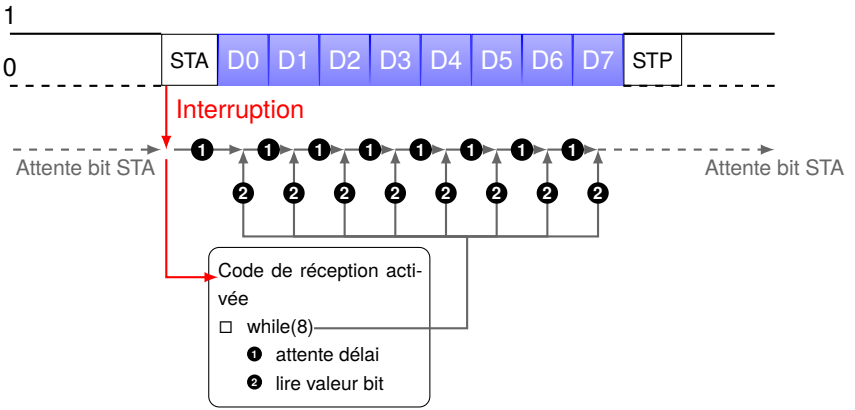




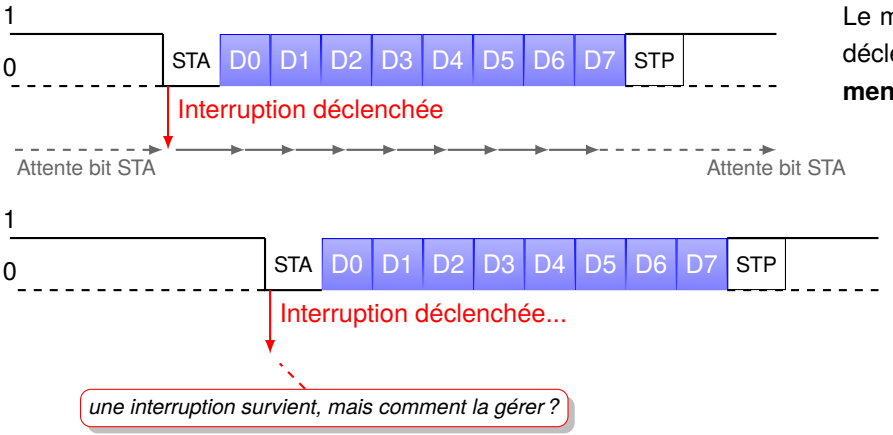
Le **code de réception** :

- ▷ attend pendant un certain délai ;
- ▷ mesure le niveau afin de déterminer la valeur du bit ;
- ▷ recommence jusqu'à la réception des 8 bits de données.

Le **code de réception** :



Gestion de deux canaux de réception asynchrone

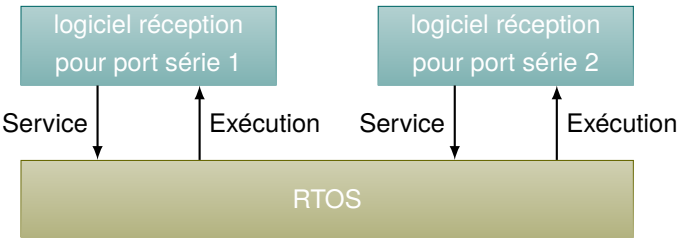


Le modèle de gestion basé sur les routines déclenchées par «*interruption*» est **difficilement extensible** !

La solution : utiliser un OS temps réel, «*RTOS*»

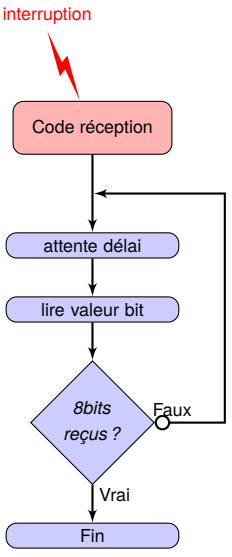
Avec RTOS, la configuration se simplifie :

- ▷ on utilise **deux occurences** d'une tâche de gestion de port série ;
- ▷ l'OS fournit un service et s'occupe de son exécution.



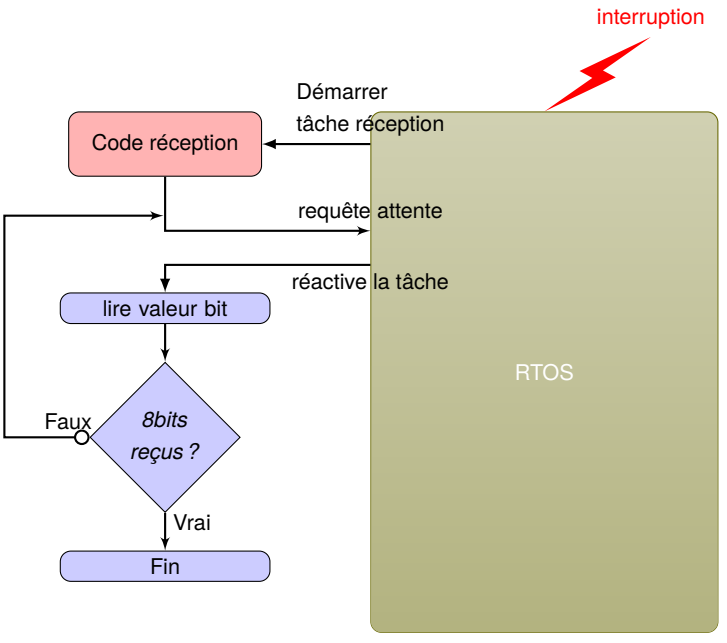
Le code de réception pour une communication asynchrone

Sans RTOS :



Le délai est interne au code de réception.

Avec RTOS :



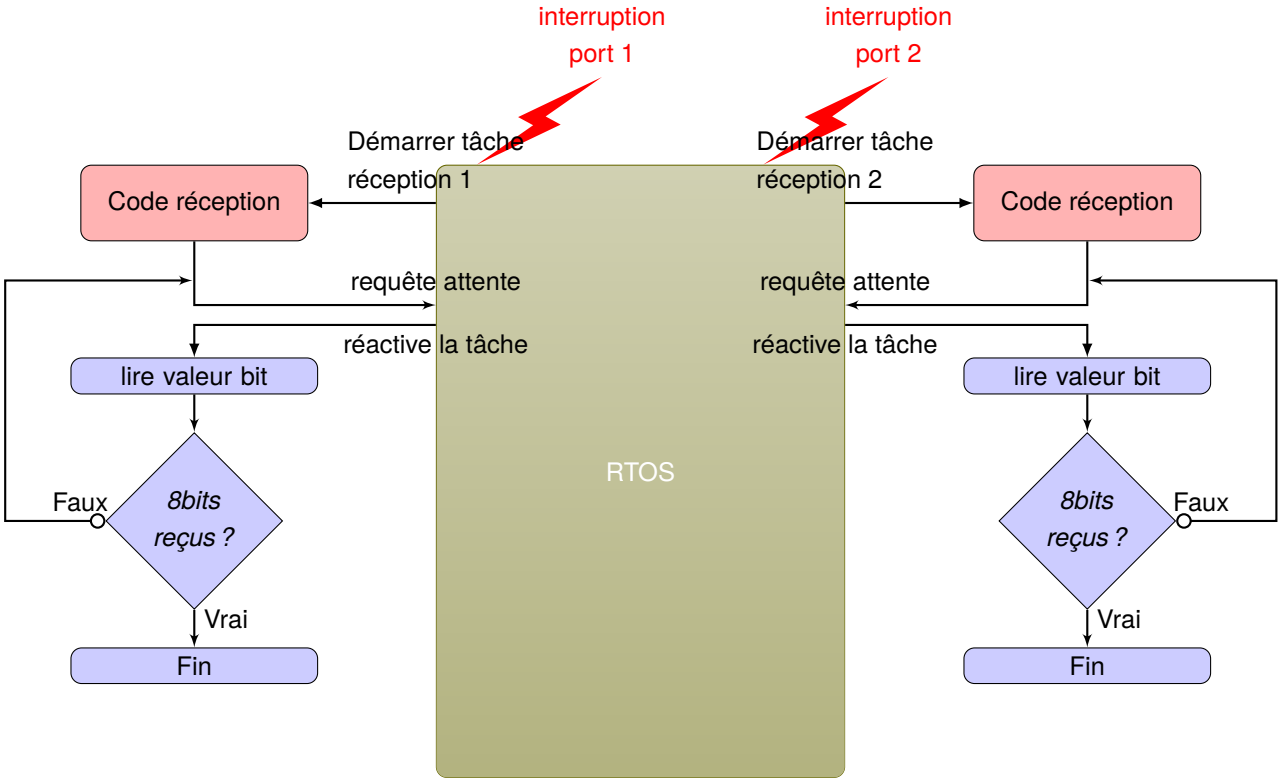
Le délai d'attente de la tâche de réception est du temps rendu à RTOS.

RTOS :

- gère l'interruption ;
- déclenche la tâche de réception ;
- récupère le délai d'attente de la tâche de récepytion$\Longrightarrow$Il peut l'utiliser pour faire autre chose !

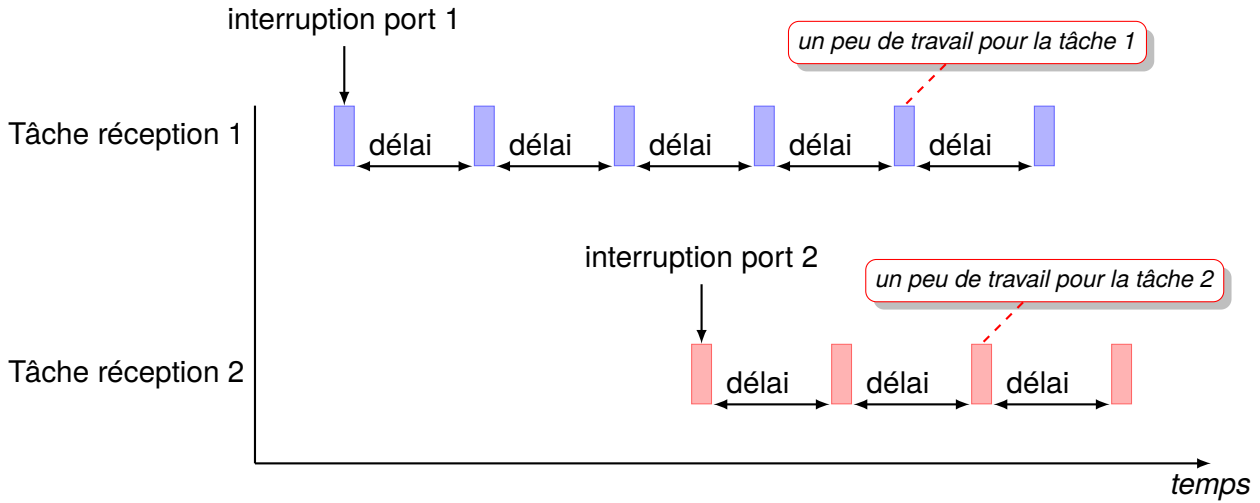
Gestion de deux canaux de réception asynchrone

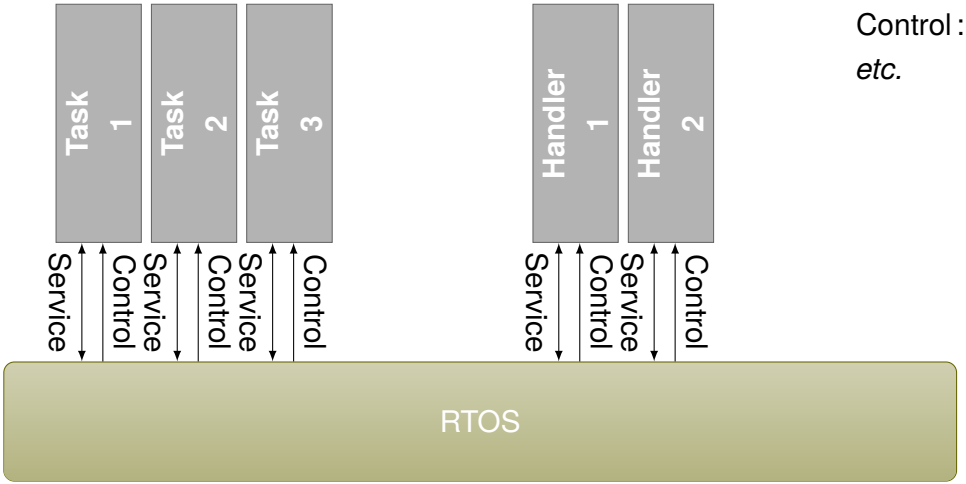
- On demande l'allocation de deux tâches de réception à RTOS ;
- RTOS partage le temps entre les deux tâches.



La responsabilité de RTOS

- exploite les ressources «*hardware*» de manière efficace :
  $\Longrightarrow$ fournit un mécanisme de bascule entre les différentes tâches ;
- fournit différents **services** aux tâches.

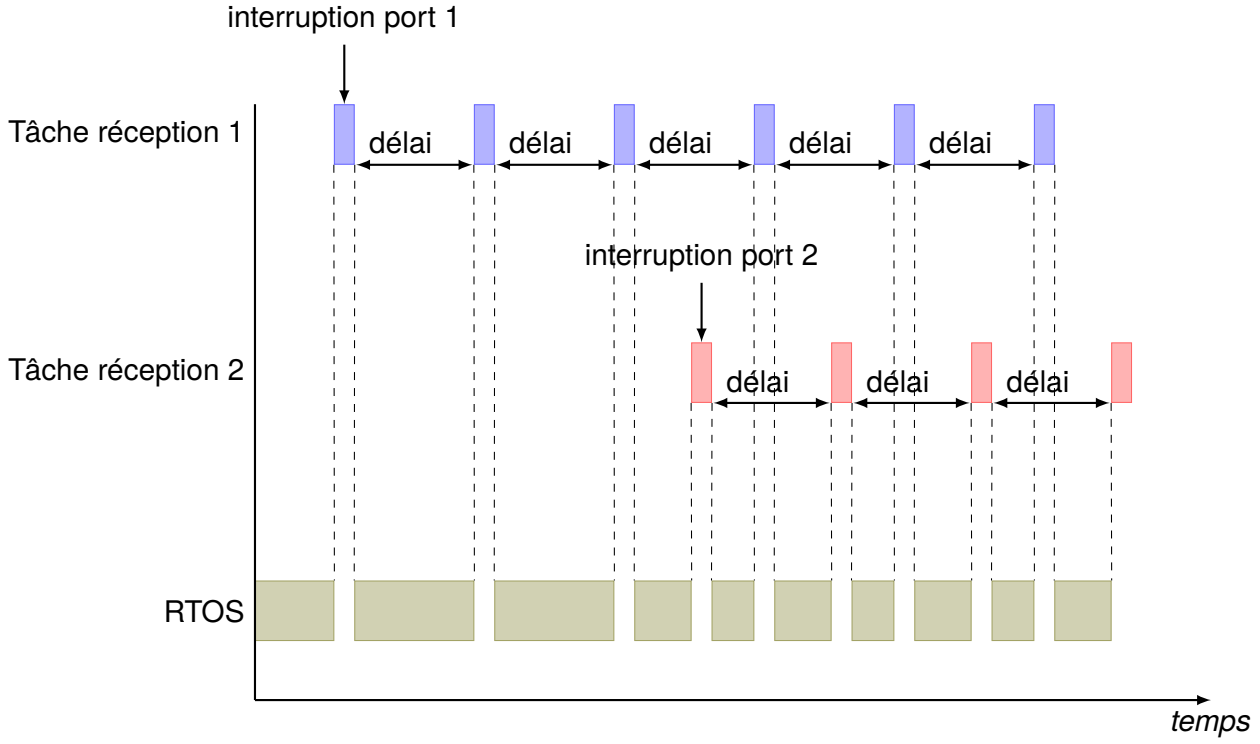




Control : activate, restart, terminate, preempt, *etc.*

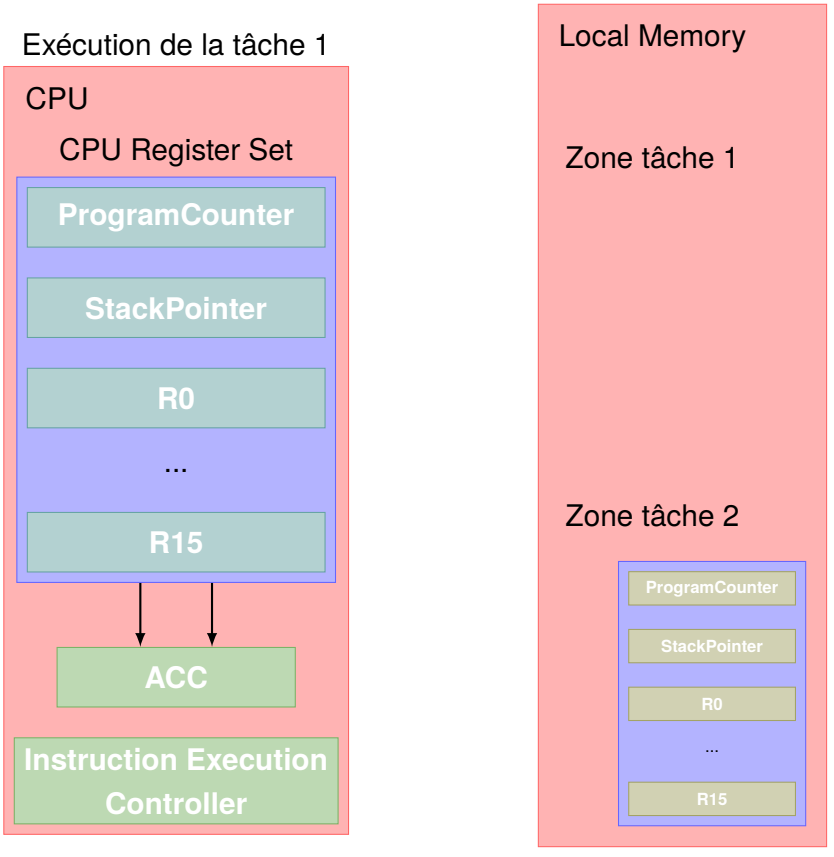
- **Task** : application gérée par RTOS :
  - RTOS en contrôle le lancement, la terminaison, *etc.* ;
  - une tâche peut également faire des appels systèmes vers RTOS, des APIs propres à RTOS ;
- **Handler** : logiciel exécuté lors d'une interruption de l'exécution du logiciel courant ;
  - examples : un «*Interrupt handler*» gestionaire d'interruption (ISR), un «*Cyclic handler*», un «*Exception handler*», *etc.*

Bascule exécution d'une tâche à une autre ou à RTOS



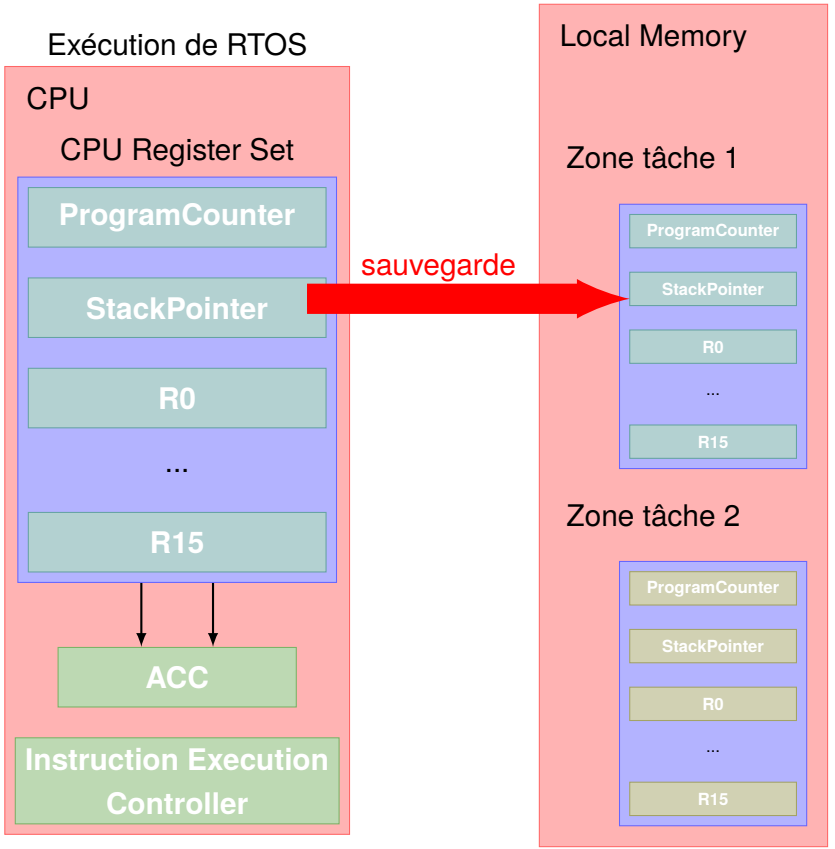
Lors d'un «*délai*», la tâche associée est «*suspendue*» et RTOS passe en mode «*Idle*» s'il n'a rien à faire ou peut réaliser du travail nécessaire au bon fonctionnement de l'OS.

Changement de tâche ou «*task switching*»



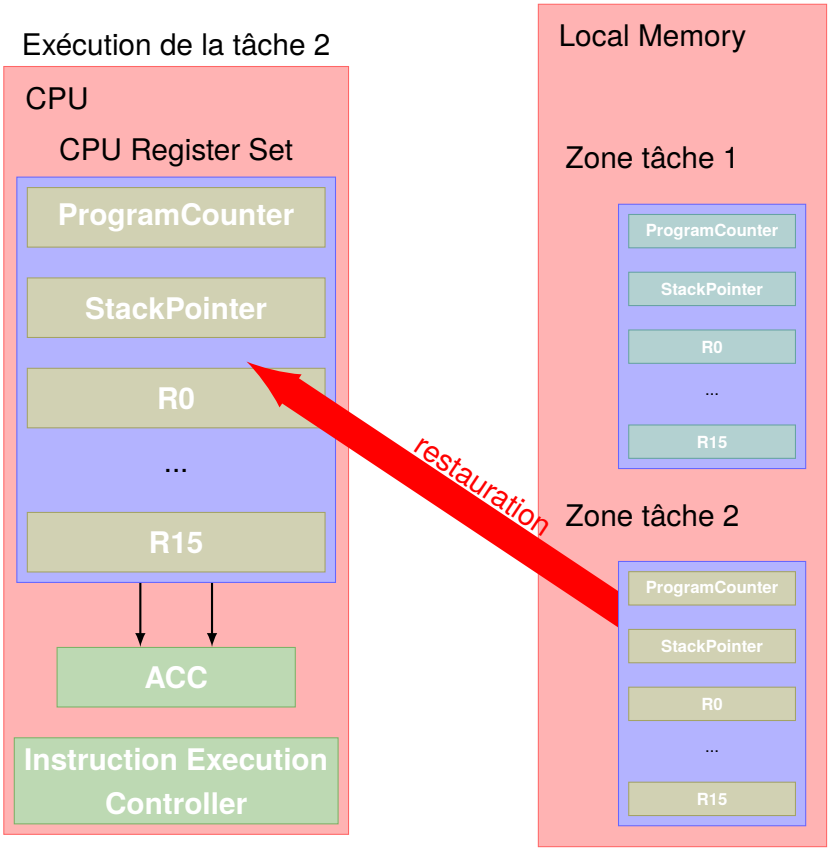
- La tâche 1 est en cours d'exécution sur le processeur ;
- on va passer à l'exécution de la tâche 2...

Changement de tâche ou «*task switching*»



- ▷ on va passer à l'exécution de la tâche 2...
- ▷ on va sauvegarder les valeurs des registres de la tâche 1 dans la zone mémoire réservée à cette tâche ;

Changement de tâche ou «*task switching*»



- ▷ on va passer à l'exécution de la tâche 2...
- ▷ on va restaurer les valeurs des registres de la tâche 2 dans les registres du processeur ;

⟹*On exécute maintenant la tâche 2*

Passer d'une tâche à une autre correspond à remplacer les données dans les registres du processeur.

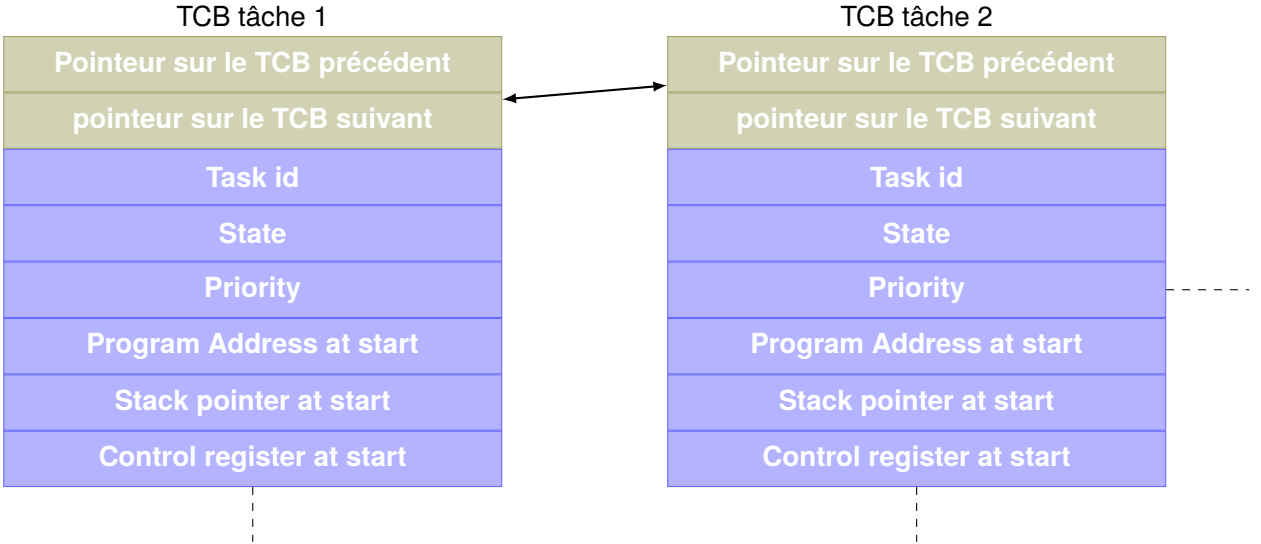
En particulier le registre «ProgramCounter» *pointe sur les instrutions de la tâche 2 à la place de celles de la tâche 1, une fois que RTOS a fini de réaliser l'échange.*

Changement de context ou «*context switching*»

- **Context** :
  - initialement, il correspondait au flux de traitement des instructions ;
  - maintenant, il correspond à l'ensemble des valeurs des registres du processeur durant l'exécution d'un processus ;

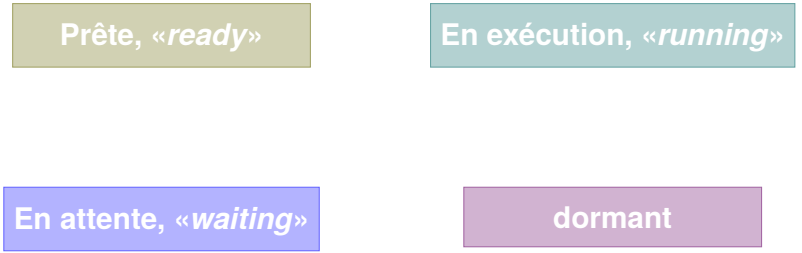
  Changement de context $\Longleftrightarrow$ changement des valeurs des registres du processeur ;
  Changement de tâche $\Longleftrightarrow$ changement de contexte ;
- **Dispatch** :
  - transférer les «*droits*» d'exécution d'une tâche à une autre ;
  - inclus un changement de contexte.

Notion de TCB, «*Task Control Block*» : utilisé par RTOS pour la gestion de chaque tâche



Les états d'une tâche

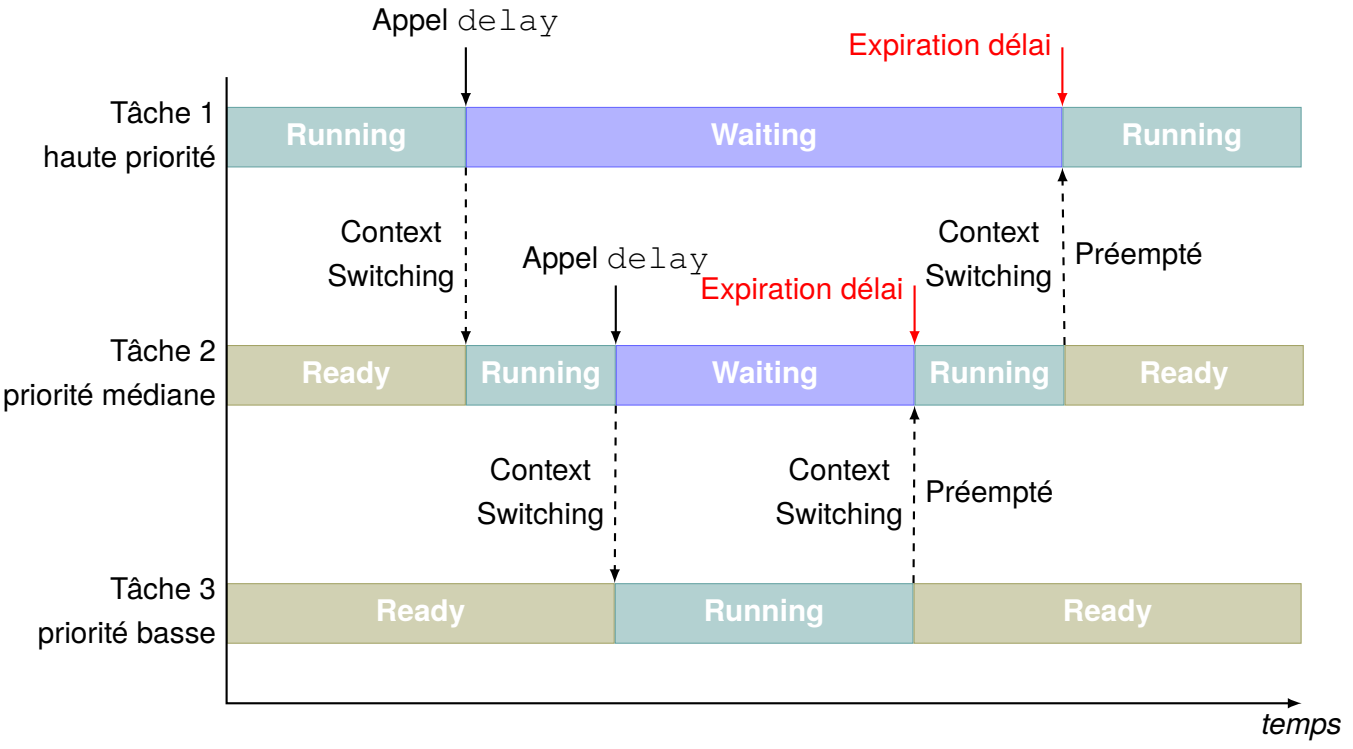
Chaque tâche appartient à un état parmis :



- «*running*» :
  - la tâche est en cours d'exécution, elle est assignée au CPU pendant cet état ;
  - une seule tâche à la fois est dans l'état «*running*» ;
  - c'est la tâche qui est de **priorité la plus élevée** qui est das cet état ;
- «*ready*»
  - une tâche est prête à être exécutée ;
  - une tâche est en attente d'exécution car il existe des tâches en état «*ready*» de **plus haute priorité** ;
  - la tâche était en cours d'exécution mais elle a éte **préemptée** ;
- «*waiting*»
  - une tâche est en attente d'un **événement** (Exemple : lors exécution d'une instruction de délai de l'API, la tâche passe en attente et l'événement correspond à la fin du délai demandé) ;
  - une tâche a abandonné son droit à être exécutée ;
- «*dormant*»
  - la tâche a été enregistrée auprès de RTOS, mais elle n'est pas encore activée ;

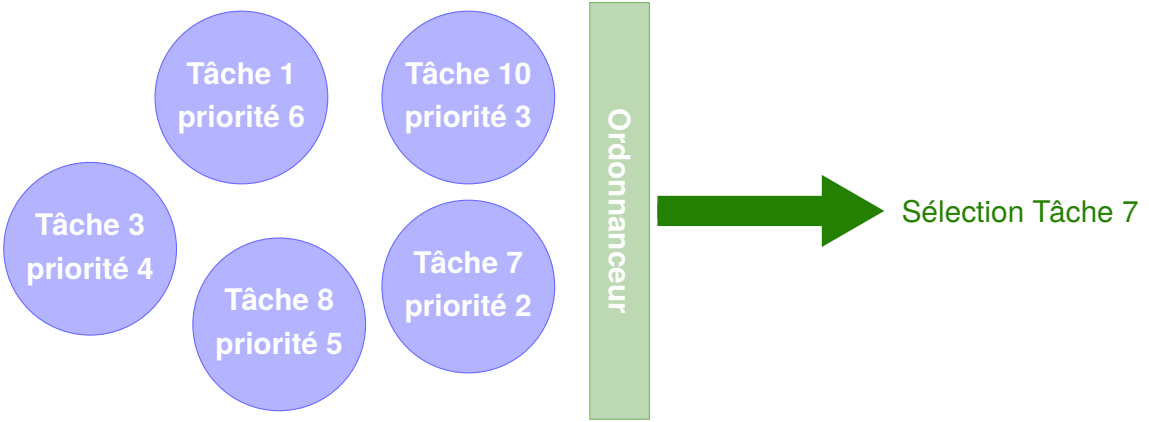
Changement de tâche et priorités

- chaque tâche a une priorité ;
- une tâche de plus haute priorité est à **exécuter avant** une tâche de plus basse priorité.



Ordonnancement, «*Scheduling*»

- chaque tâche possède une priorité fixée lors de l'initialisation ;
- RTOS sélectionne le tâche de priorité la plus élevée parmi celles en état «*ready*» ;

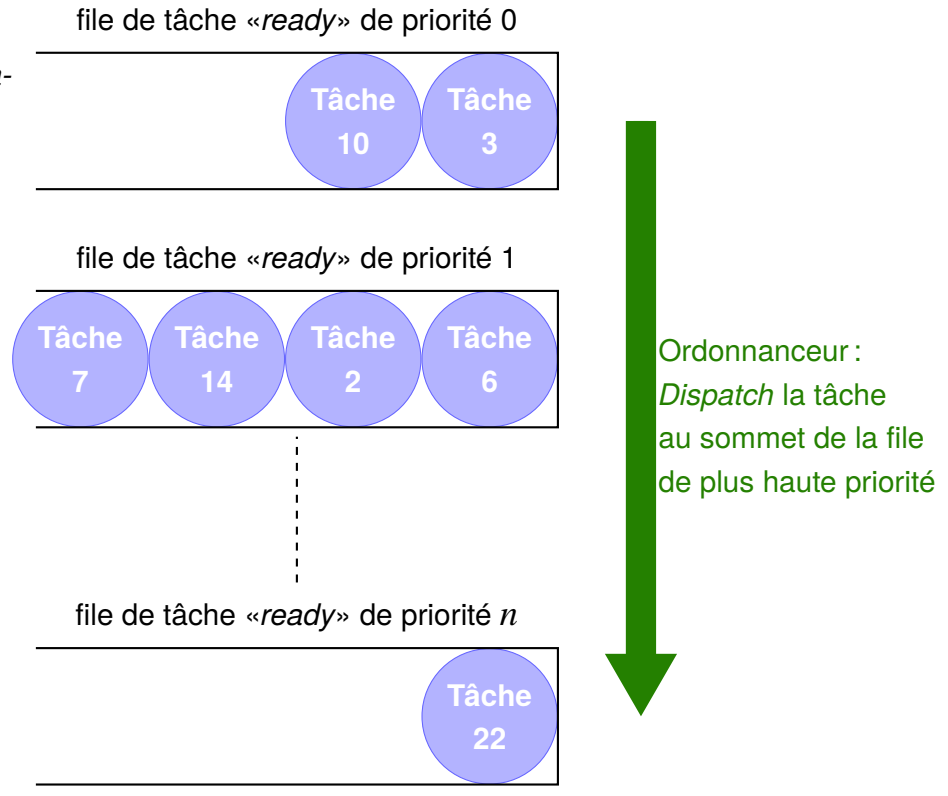


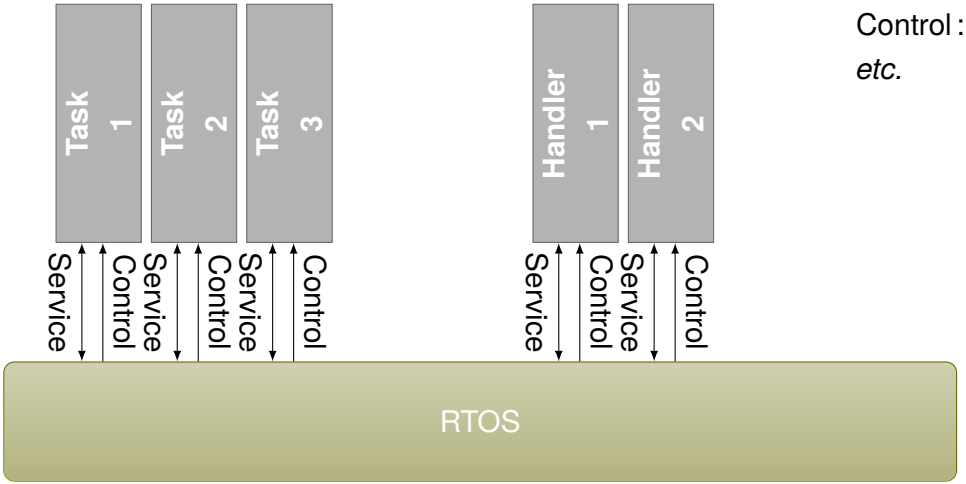
Plus la valeur de la priorité est basse, plus la priorité est élevée : la priorité 0 est la plus haute priorité.

Changement de tâche et priorités

Que se passe-t-il s'il y a plusieurs tâches avec la même priorité ?

- il y a une file pour chaque priorité ;
- lorsqu'une tâche passe à l'état «*ready*» : elle est **ajoutée à la fin** de la file associée à sa priorité ;
- l'algorithme est appelé «*Priority-based First-Come, First-Served*» ou «*FCFS*».





Control : activate, restart, terminate, preempt, *etc.*

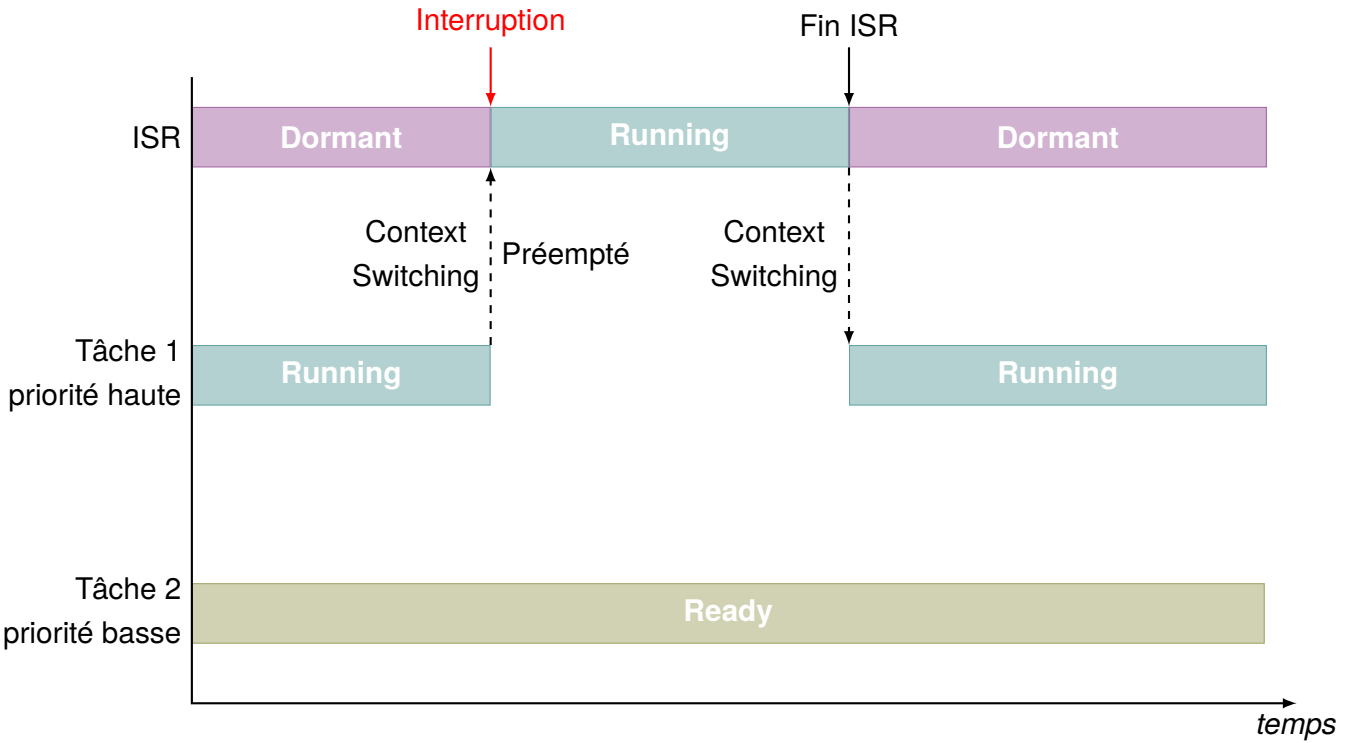
- **Task** : application gérée par RTOS :
  - RTOS en contrôle le lancement, la terminaison, *etc.* ;
  - une tâche peut également faire des appels systèmes vers RTOS, des APIs propres à RTOS ;
- **Handler** : logiciel exécuté lors d'une interruption de l'exécution du logiciel courant ;
  - examples : un «*Interrupt handler*» gestionaire d'interruption (ISR), un «*Cyclic handler*», un «*Exception handler*», *etc.*

C'est un code qui s'exécute après avoir **préempté** le code s'exécutant actuellement sur le processeur.

- «*Interrupt Handler*», ISR, «*Interrupt Service Routine*» :
  - travail activé par une interruption du à un signal extérieur ;
- «*Cyclic handler*» :
  - travail activé périodiquement ;
- «*Exception handler*» :
  - travail activé par une erreur, comme une erreur d'adressage, une erreur arithmétique, «*etc.*»
- La priorité d'exécution d'un «*handler*» est, en général, supérieur à la priorités des différentes tâches.
- Les interruptions sont, en général, désactivées pendant l'exécution d'un «*handler*».

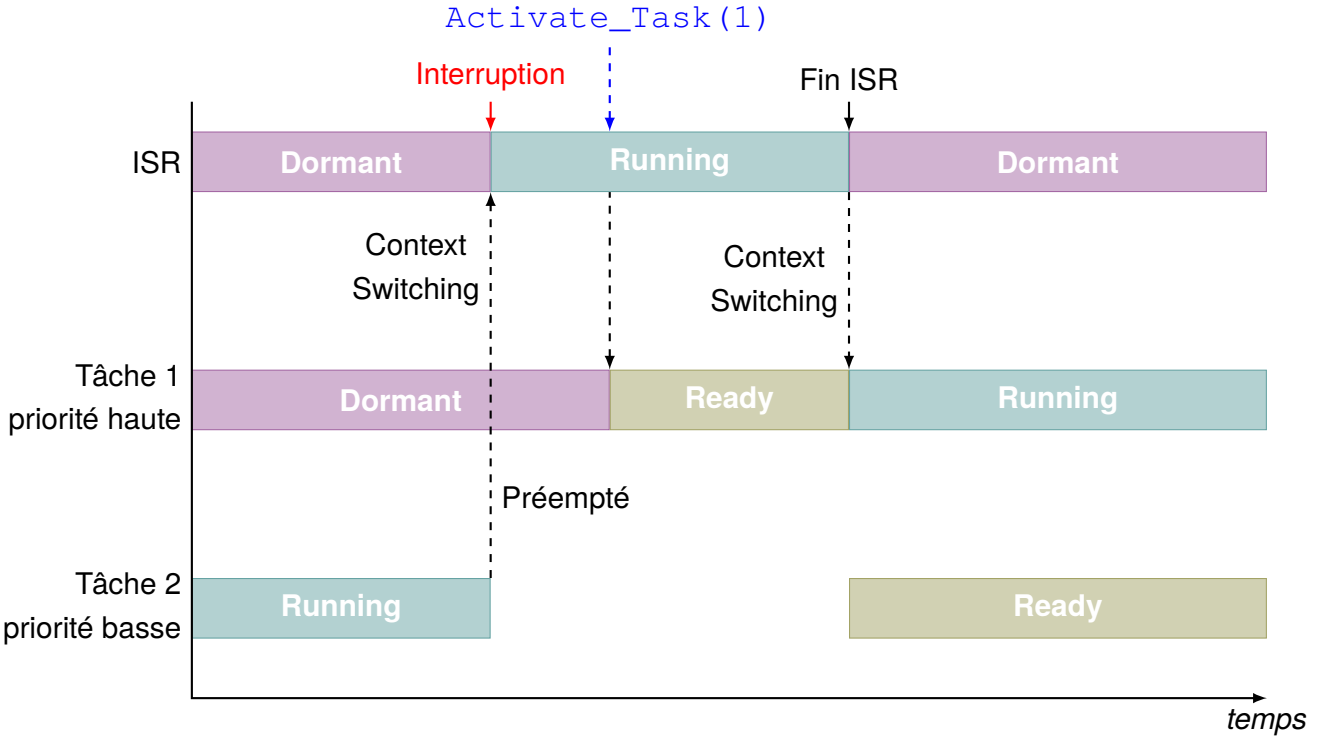
ISR, «*Interrupt Service Routine*»

- lorsqu'une interruption survient, RTOS démarre le gestionnaire, «*handler*», de l'interruption ;



Appel d'API dans une ISR, «*Interrupt Service Routine*»

- l'exécution peut ne pas «*retourner*» vers la tâche interrompue lorsque l'ISR termine ;

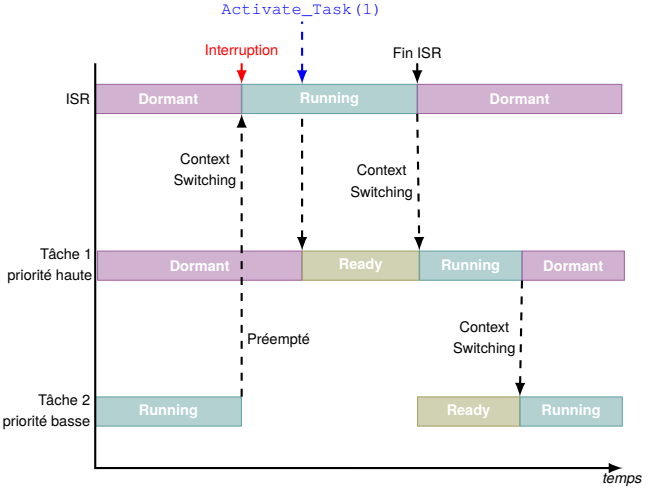


- ▷ L'ISR appelle l'API `Activate_Task(1)` qui passe la tâche 1 de l'état «*Dormant*» à «*Ready*» ;
- ▷ Lors de la fin de l'ISR, l'ordonnanceur choisit la tâche 1 qui est de priorité supérieure à tâche 2 qui passe en «*Ready*» $\Longrightarrow$ l'ISR «*communique*» vers la tâche 1.

Interruptions gérées par RTOS

- Interruptions non gérées par RTOS :
  - les APIs **ne peuvent pas** être appelées durant le traitement de l'interruption ;
  - faible surcoût, «*overhead*» ;
- Interruptions gérées par RTOS :
  - les APIS **peuvent** être appelées durant le traitement de l'interruption ;
    - ⋆ diminue la durée de l'ISR ;
    - ⋆ l'ISR peut changer quelle tâche peut être exécutée ;
    - ⋆ toutes les transitions d'une tâche à l'autre sont dues à des interruptions :
      - ▷ facile de configurer un traitement en cascade survenant à la suite d'une interruption ;
  - définition, en général, de la notion d'ISR ;
  - fort surcoût.

Cela n'est pas accessible à une interruption non gérée par RTOS :

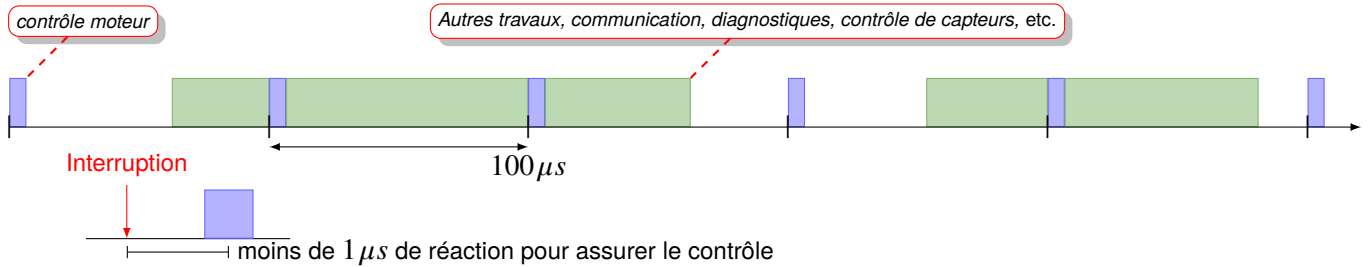


Attention

▷ Les interruptions non gérées par l'OS sont de priorités supérieures à celles gérées par RTOS.

▷ Le programme associé à leur gestion est indépendant de RTOS⟹pas d'appel d'APIs possible.

Exemple d'un contrôleur de direction dans une voiture

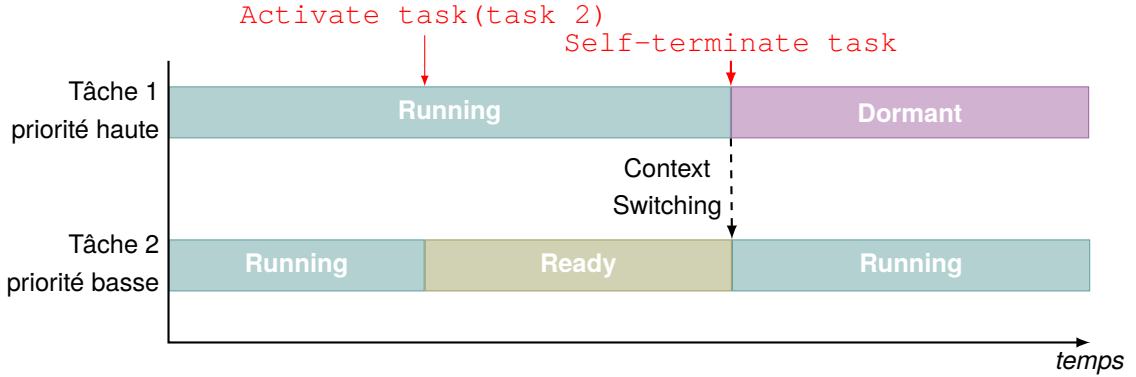


- ▷ on utilise un «*cyclic handler*» pour garantir que le contrôle s'effecture régulièrement : ici toutes les $100\,\mu s$;
- ▷ l'interruption doit être prise en compte dans un délai garanti : ici moins de $1\,\mu s$;
- ▷ entre le traitement de ces interruptions d'autres tâches peuvent être réalisées ;

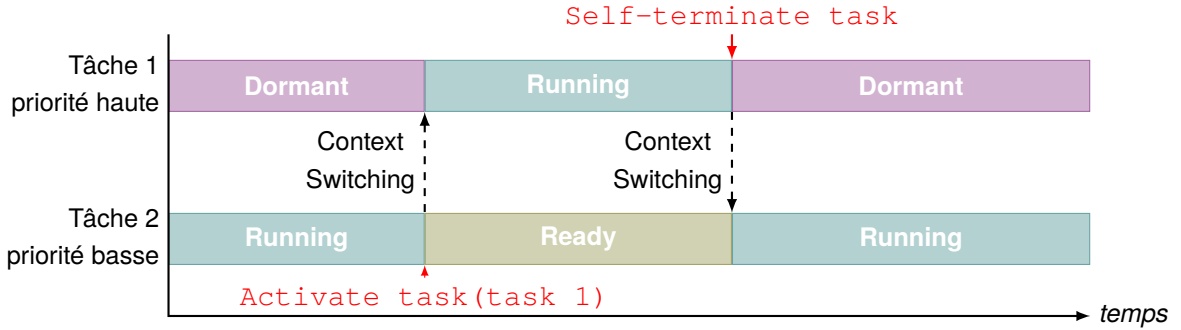
Le travail du «*cyclic handler*» est en relation directe avec des événements extérieurs/contrôleur (ici le moteur de la direction assistée du véhicule) qui sont «*temps réel*».

Un système RTOS permet de gérer ces opératios «*temps réel*», mais aussi de réaliser d'autres travaux moins prioritaires comme la gestion de l'affichage du tableau de bord, la lecture de capteurs *etc.*

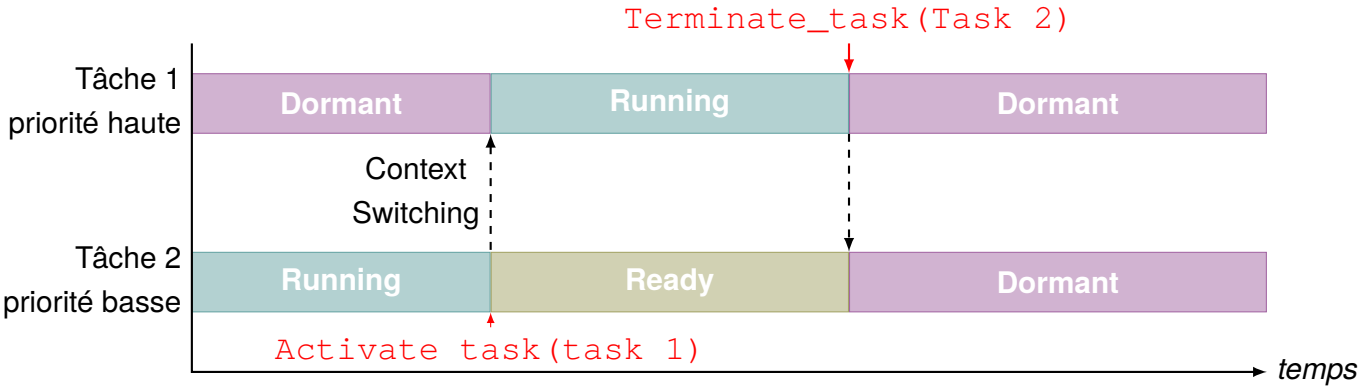
- `Activate_task(Task ID)`
  - RTOS change l'état de la tâche indiquée en argument de «*Dormant*» à «*Ready*» ;
- `Self-terminate_task`
  - RTOS change l'état de la tâche qui l'appelle de l'état «*Running*» à «*Dormant*» ;



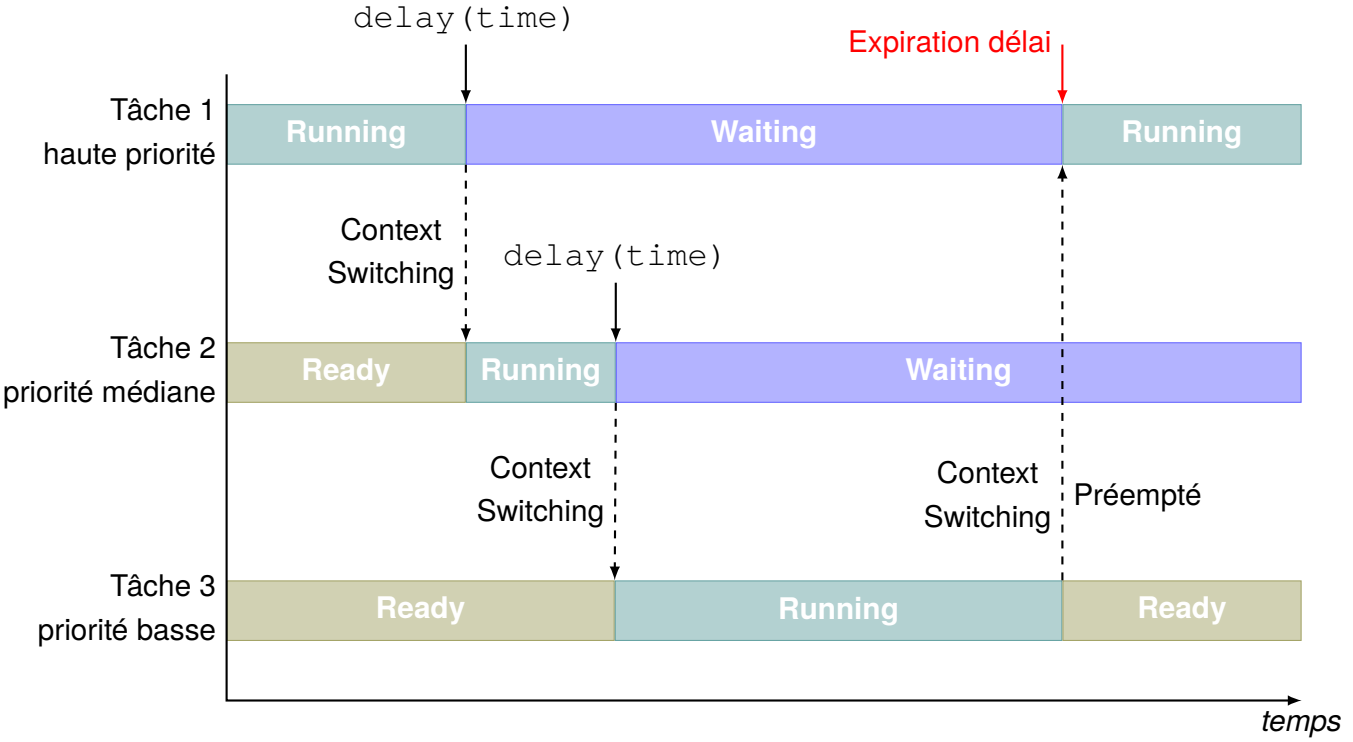
Exemple où une tâche de priorité inférieure active une tâche de priorité supérieure



- `Terminate_task(Task ID)`
  - RTOS change l'état de la tâche indiquée en argument vers l'état «*Dormant*» :
    - «*Ready*» ⟹ «*Dormant*» ;
    - «*Waiting*» ⟹ «*Dormant*».



- `Delay`
  - RTOS change létat de la tâche appellante à «*Waiting*» pendant la durée indiquée en argument ;
  - après le temps écoulé, RTOS change l'état de la tâche à «*Ready*».



Sémaphore

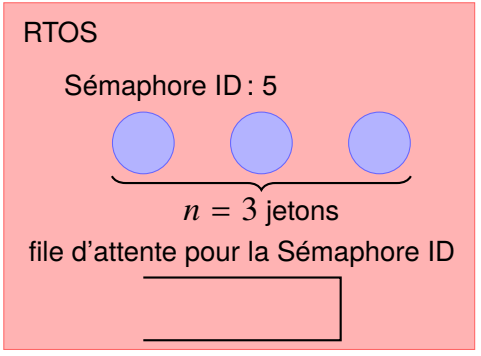
- ▷ fournit un contrôle :
  - ◇ acquérir une sémaphore pour autoriser :
    - ⋆ des opérations ;
    - ⋆ un accès ;
    - ⋆ *etc.*

    Le but dépend de l'application.
- ▷ l'application acquiert la sémaphore quand c'est nécessaire ;
- ▷ après l'avoir obtenue, l'application la libère juste après avoir terminé le travail qui exigeait une autorisation.

- □ `acquire_semaphore(Semaphore ID)` :
  - ◇ requête d'acquisition de la sémaphore indiquée ;
  - ◇ si la sémaphore est **disponible**, l'appel de la fonction RTOS retourne «*Succeed*» ce qui indique **l'obtention** de la sémaphore ;
  - ◇ si la sémaphore est **indisponible**, RTOS change l'état de la tâche appelante vers «*Waiting*».
- □ `Release_semaphore(Semaphore ID)` :
  - ◇ libère la sémpahore indiquée en argument ;
  - ◇ si une tâche est en attente de la sémaphore indiquée, alors RTOS donne la sémaphore à la tâche de plus haute priorité en état «*Waiting*».

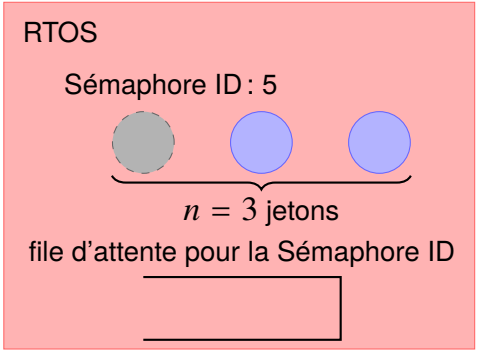
    RTOS change également l'état de cette tâche vers «*Ready*».

Sémaphore avec compteur



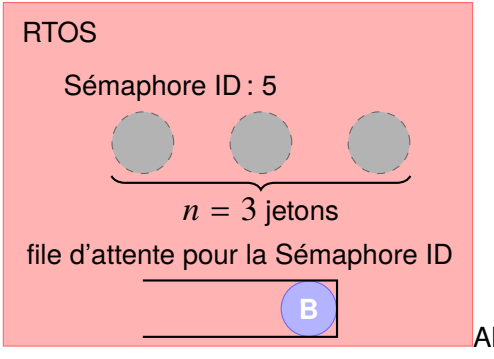
- le nombre de sémaphore est n ;
- les autres conditions sont les mêmes que pour les sémaphores binaires :
  - ◇ il existe plusieurs sémaphores pouvant être récupérées par plusieurs tâches ;
  - ◇ une sémaphore peut être libérer par une tâche que ne l'a pas acquise à l'origine ;
  - ◇ une ISR peut libérer une sémaphore ;
  - ◇ la valeur initiale de la sémaphore peut être zéro.

Si une tâche appelle `Acquire_semaphore(5)` :



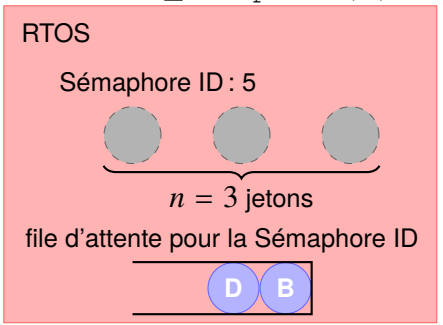
Un jeton/sémaphore est retiré/donné à la tâche.

Si une tâche B appelle `Acquire_semaphore(5)` et qu'il n'y a plus de sémaphore :



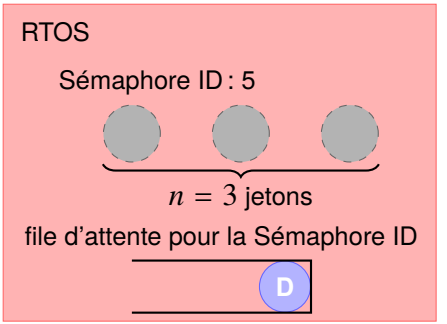
Alors la tâche B est mise dans la file d'attente $\Longrightarrow$ elle est en état «*Waiting*».

- Les deux tâches B et D sont dans la file d'attente ;
- ▷ une tâche va libérer la sémaphore avec `Release_sémaphore(5)` :

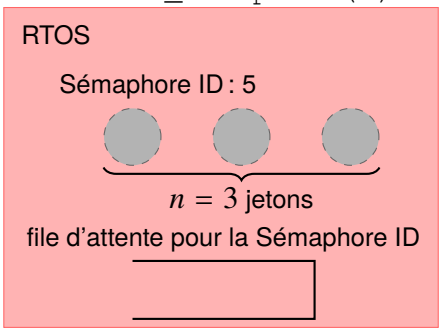


⟹RTOS va :

- ▷ affecter la sémaphore à la tâche B ;
- ▷ changer l'état de B à «*Ready*».



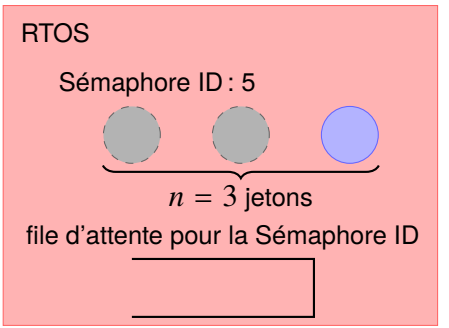
- ▷ une tâche va libérer la sémaphore avec `Release_sémaphore(5)` :



Il n'y a pas de tâches dans la file d'attente...

⟹RTOS va :

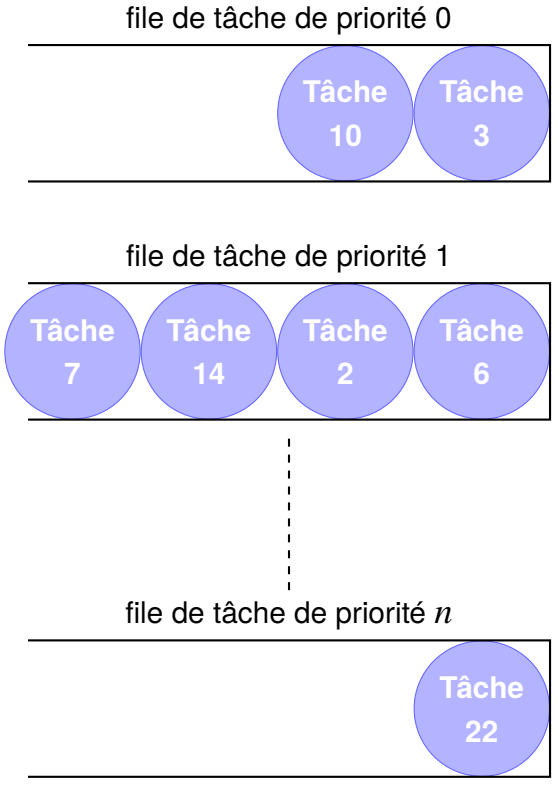
- ▷ augmenter le compteur de la sémaphore.



File d'attente de Sémaphore

Différentes tâches peuvent attendre la même sémaphore (même ID) :

File d'attente de la sémaphore ID=0



Chaque sémaphore a le même nombre de files que de priorités :

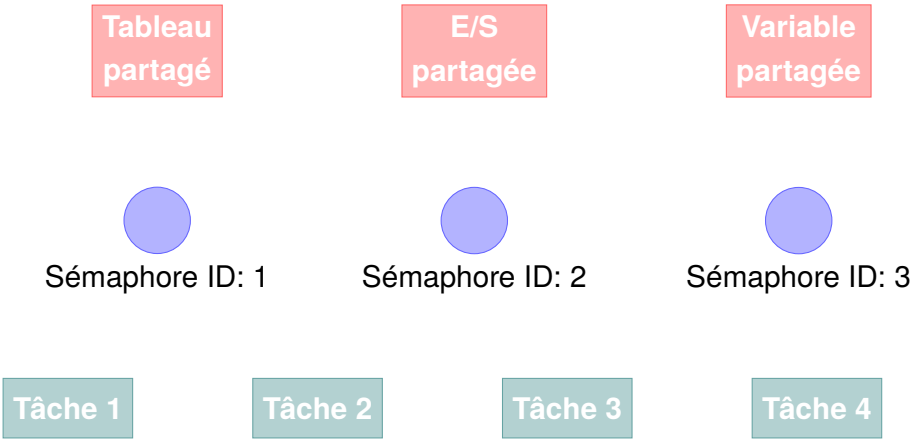
nombre total de files = nombre de Sémaphore ID∗ nombre de priorité

Si l'état d'une tâche passe à «*Waiting*» alors la tâche est ajouté à la file associée à la priorité de la tâche.

Ordonnanceur :
FCFS basé sur la priorité
la tâche au sommet de la file
de plus haute priorité est libérée
de l'état «*Waiting*»

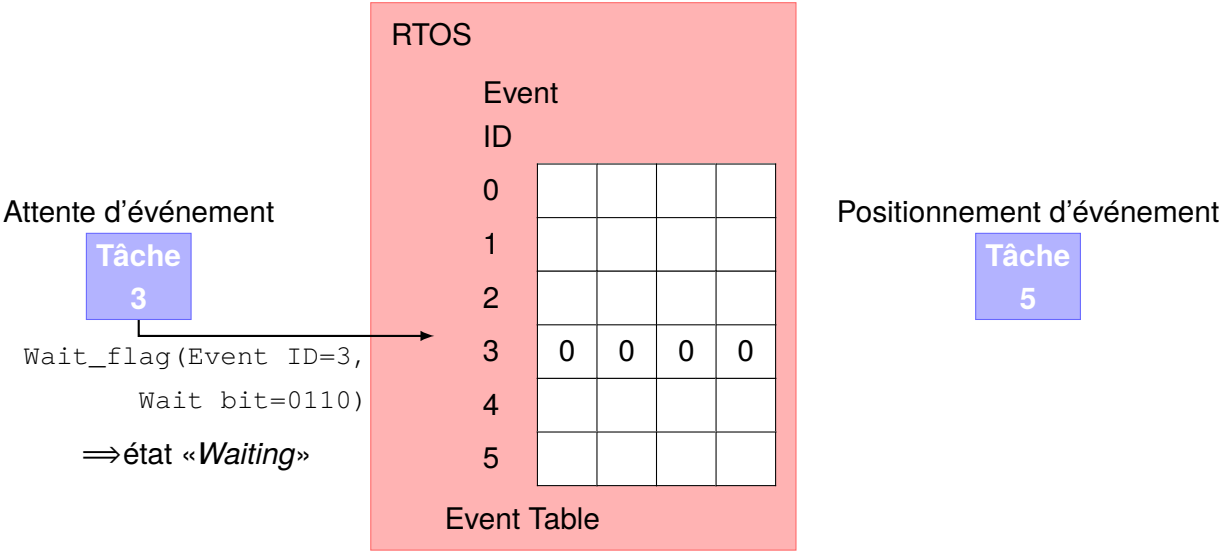
Utilisation des Sémaphores

Accès synchronisé à des ressources partagées



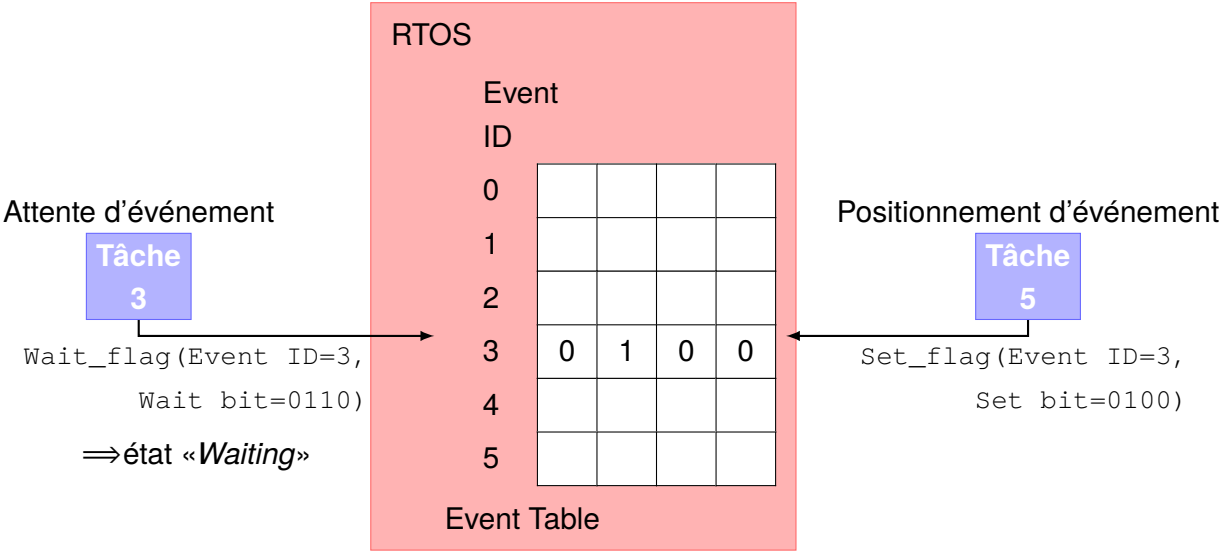
- «*raise a flag*» pour envoyer un signal à une autre tâche :
  - communication entre tâches ;
  - synchronisation entre tâches ;
- positionne ou lit un drapeau (bit) dans un registre particulier de la table «*Event*».
- `Wait_flag(Event ID, Flag pattern:`
  - si le «*flag pattern*» indiqué en argument **ne correspond pas** au registre de la table «*Event*» indiqué par le «*Event ID*» alors RTOS change l'état de la tâche appelante à «*Waiting*».
    La tâche restera dans l'état «*Waiting*» tant que le registre ne correspondra au «*flag pattern*» demandé.
  - si le «*flag pattern*» indiqué en argument **correspond** au registre de la table «*Event*» indiqué par le «*Event ID*» alors RTOS retourne «*Succeed*».
  - une ISR **ne peut pas appeler** cette API.
- `Set_flag(Event ID, Set bit`
  - RTOS réécrit le registre de la table «*Event*» indiqué par le «*Event ID*» avec le motif, «*pattern*», de bits donné en argument.
    Si une tâche en état «*Waiting*» et son motif d'attente «*match*», correspond à la valeur courante du registre alors RTOS change l'état de la tâche de «*Waiting*» à «*Ready*».
  - une ISR **peut appeler** cette API.

«*Event Flag*» ou drapeau d'événement



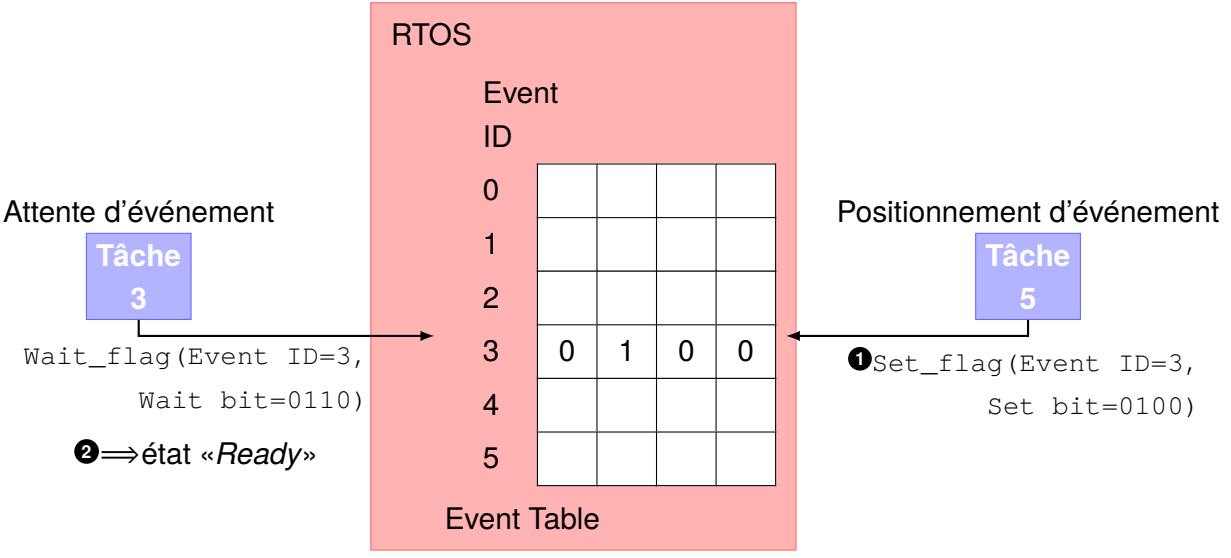
- la tâche 3 appelle `Wait_flag(Event ID=3, Wait bit=0110);`
- l'entrée 3 de la table d'événement ne correspond pas au motif demandé
  ⟹RTOS passe la tâche 3 en état «*Waiting*».

«*Event Flag*» ou drapeau d'événement



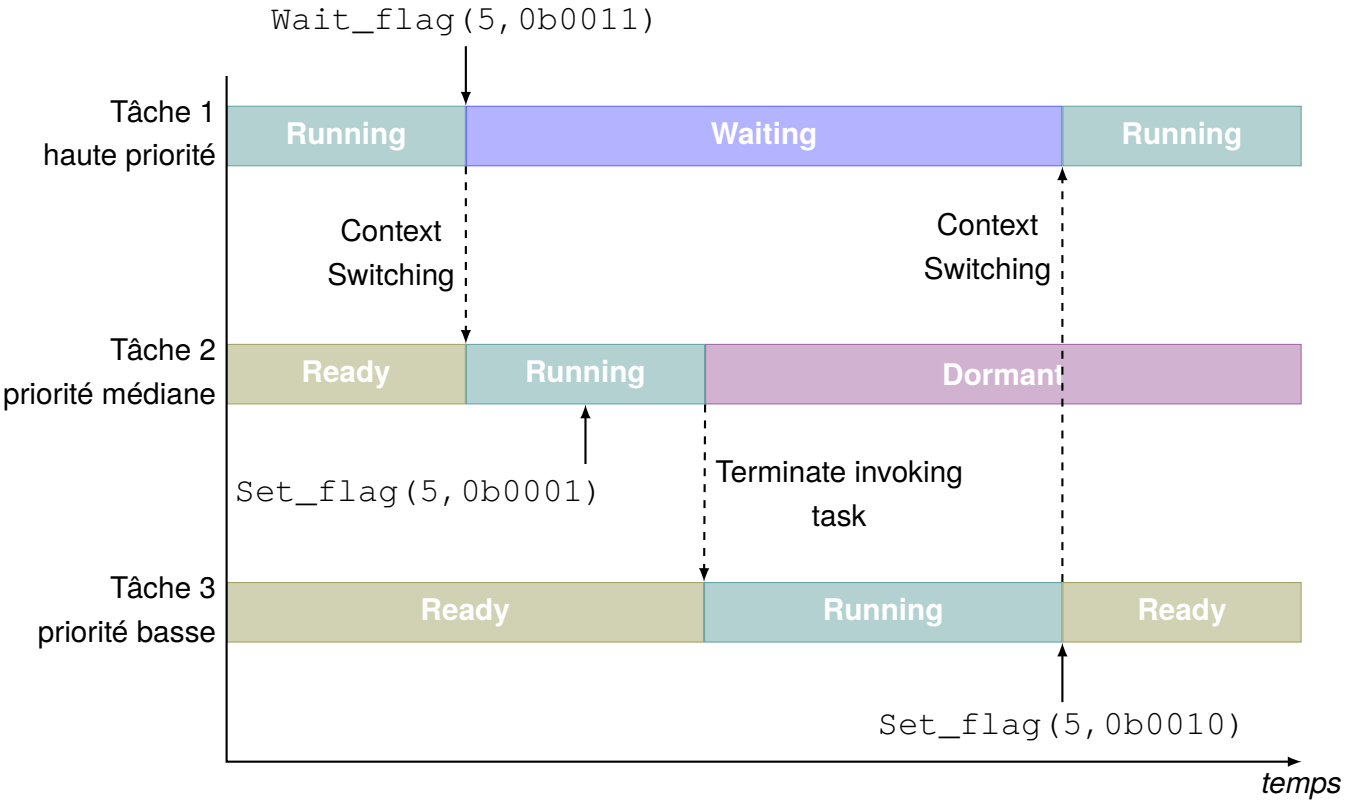
- la tâche 5 appelle `Set_flag(Event ID=3, Set bit=0100);`
- l'entrée 3 de la table d'événement devient `0100`;
- ce motif ne correspond pas au motif attendu par la tâche 3.

«*Event Flag*» ou drapeau d'événement



- la tâche 5 appelle `Set_flag(Event ID=3, Set bit=0010)`❶;
- l'entrée 3 de la table d'événement devient `0110`;
- ce motif correspond au motif attendu par la tâche 3
  ⟹RTOS passe la tâche 3 en «*Ready*»❷

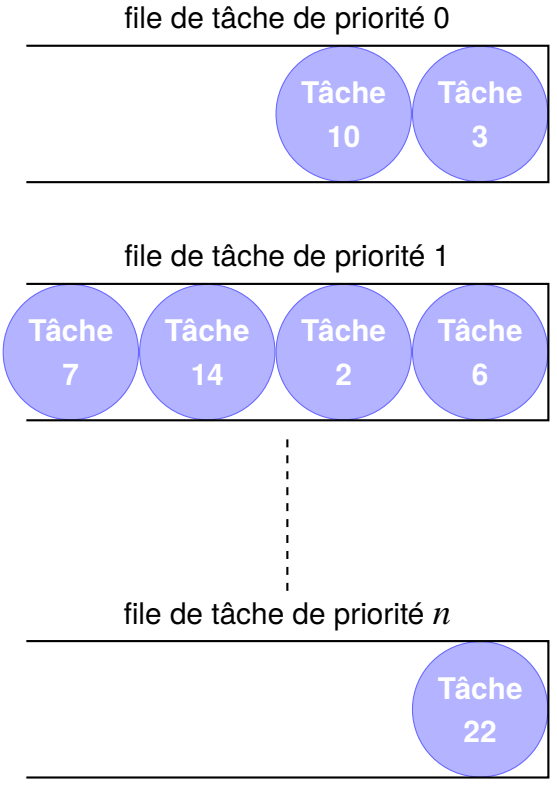
Les tâches 3 et 5 se sont synchronisées sur la construction d'un motif.


Wait_flag(5,0b0011)
Tâche 1
haute priorité
Running
Waiting
Running
Context
Switching
Context
Switching
Tâche 2
priorité médiane
Ready
Running
Dormant
Set_flag(5,0b0001)
Terminate invoking
task
Tâche 3
priorité basse
Ready
Running
Ready
Set_flag(5,0b0010)
temps

File d'attente d'Event

Différentes tâches peuvent attendre un «*falg pattern*» pour le même event ID :

File d'attente de l'Event ID=0



Chaque Event ID a le même nombre de files que de priorités :

nombre total de files = nombre de Event IDs $*$ nombre de priorité

Si l'état d'une tâche passe à «*Waiting*» alors la tâche est ajouté à la file associée à la priorité de la tâche.

Ordonnanceur :
FCFS basé sur la priorité
la tâche au sommet de la file
de plus haute priorité est libérée
de l'état «*Waiting*»

RTOS doit vérifier pour chaque tâche en attente suivant différents «*flag pattern*» $\Longrightarrow$ La file n'est pas une simple file FIFO.

- «*Mail Box*»
  - `Send_message(Mailbox ID, Message priority, Message)`
  - `Receive_message(Mailbox ID)` si la mailbox est vide ⟹passe la tâche en «*waiting*» d'un message
- «*Fixed length memory pool*»
  - `Acquire_fixed_memory_block(Memory Pool ID)` (*) si la mémoire est indisponible ⟹passe la tâche en «*waiting*» d'un message
  - `Release_fixed_memory_blco(top address of the block)`
- «*Variable length memory pool*»
  - `Acquire_variable_memory_block(Length of the block)` (*) identique au `fixed`
  - `Release_variable_memory_blco(top address of the block)`
- «*Sleep*»
  - `Sleep`
  - `Wakeup(Task ID)`
- «*Change priority*»
  - `Change_Priority(Task ID, Priority)`
- «*Rotate ready Queue*»
  - `Rotate_ready_queue(Priority)`
- «*Disable/Enable Dispatch*»
  - `Disable_dispatch`
  - `Enable_dispatch`
- «*Lock/Unlock CPU*»
  - `Lock_CPU`
  - `Unlock_CPU`

APIs pour la transition à l'état «*Waiting*»

APIs utilisées par RTOS pour passer la tâche appelante à l'état «*Waiting*»

- `Wait_flag(Event ID, Flag pattern)`
- `Acquire_semaphore(Semaphore ID)`
- `Receive_message(Mailbox ID)`
- `Send_data_queue(dataqueue ID)`
- `Receive_data_queue(dataqueue ID)`
- `Acquire_fixed_memory_block(Memory Pool ID)`
- `Sleep`

- Seule des tâches **peuvent** appeler ces APIs.
  ISRs et «*Cyclic Handlers*» **ne peuvent pas** les appeler :
  - les «*handlers*» doivent **terminer aussi vite que possible** car les interruptions sont désactivées durant leur exécution.
- Ces APIs ont :
  - une option Timeout ;
  - une option Polling ;

Les APIs utilisées par RTOS pour passer la tâche appelante dans l'état «*Waiting*» peuvent utiliser une option de «*Timeout*» :

- `Wait_flag(Event ID, Flag pattern,AND/OR, Timeout value)`
- `Acquire_semaphore(Semaphore ID, Timeout value)`
- `Receive_message(Mailbox ID, Timeout value)`
- `Acquire_fixed_memory_block(Memory Pool ID, Timeout value)`
- `Sleep(Timeout value)`

Lorsque le «*Timeout*» expire la tâche passe de l'état «*Waiting*» à l'état «*Ready*» **automatiquement**, y compris si elle est dans une file d'attente d'une sémaphore et quelque soit son rang dans cette file.

$\Longrightarrow$Si le logiciel rencontre un bug ou une exception, la tâche est capable d'être réactivée.

$\Longrightarrow$On évite qu'un blocage du système se produise.

- Les APIs utilisées par RTOS pour passer la tâche appelante dans l'état «*Waiting*» peuvent utiliser une option de «*Polling*» ;
- Les ISRs et «*Cyclic Handlers*» **peuvent** appeler ces APIs si l'option de polling est utilisée.
- Si l'option de «*polling*» est **utilisée**, RTOS **ne passe pas** la tâche appelante dans l'état «*Waiting*».
- Exemple : `Acquire_sempahore` :
  - Si la sémaphore est disponible, RTOS retourne un «*Succeed*» pour indiquer que la sémaphore a été obtenue.
  - S'il n'y a pas de sémaphore et que l'option de «*polling*» **n'est pas utilisée**, RTOS passe l'état de la tâche appelante à «*Waiting*» ;
  - S'il n'y a pas de sémaphore et que l'option de «*polling*» **est utilisée**, RTOS retourne un «*Failure of Acquiring semaphore*» et ne passe pas l'état de la tâche appelante à «*Waiting*» ;

Tick

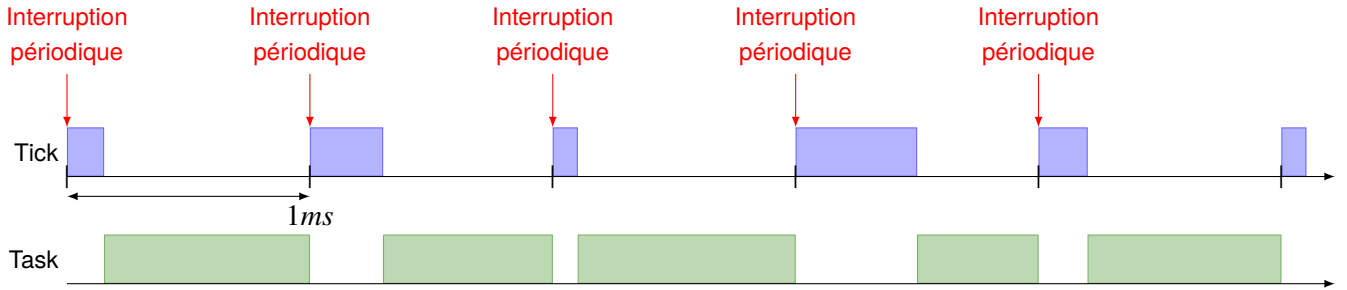
- Utiliser pour la gestion du temps dans un RTOS :
  - le «*Tick*» est activé par une **interruption périodique** ;
  - la période de ce «*Tick*» est appelé le «*Tick interval*», c-à-d l'unité de temps du RTOS :
    - plus le «*Tick interval*» est court ↘, plus la gestion du temps est précise ↗ ;
    - mais du au surcoût, «*overhead*», entrainé par le traitement du Tick par l'API...
    - la **période** est généralement de l'ordre de la milliseconde : $1ms$,
      et même pour des processeurs rapide supérieur à $100\mu s$ *;*
- Travaux intervenant à chaque Tick :
  - le RTOS vérifie si c'est le moment de déclencher un «*cyclic handler*» :
    - si oui, le RTOS active le «*cyclic handler*» ;
  - le RTOS décrémente le compteur «*timeout*» dans le TCB d'une tâche :
    - lorsque le compteur de la tâche atteint la valeur zéro, le RTOS change l'état de la tâche de «*Waiting*» à «*Ready*».
      Et le RTOS enlève le TCB de cette tâche de la file d'attente où elle était et l'ajoute à la file des tâches «*Ready*».

Attention

- Pendant la gestion du «*Tick*», les interruptions sont désactivées ;
- Le temps de gestion dépend de la taille des files à parcourir et à traiter (en particulier les «*Event flag*» sont plus longues à traiter) ;
- Cela peut impacter le traitement des interruptions matérielles extérieures.

Les problèmes liés au Tick

- Tick est une **fonction obligatoire** d'un RTOS ;
- Le traitement du Tick débute à chaque «*tick interval*» :
  - le Tick cause une surcharge périodique du CPU ;
- En général, les interruptions sont **désactivéees** durant les opérations sur les différentes files :
  - cause une **dégration du temps de réponse** pour le traitement de ces interruptions ;
- Le «*Tick interval*» est **l'unité de temps** à la base des RTOS :
  - plus le «*tick interval*» est court ↘, plus la gestion du temps est précise ↗ ;
  - plus le «*tick interval*» est court ↘, plus le surcoût est important ↗.



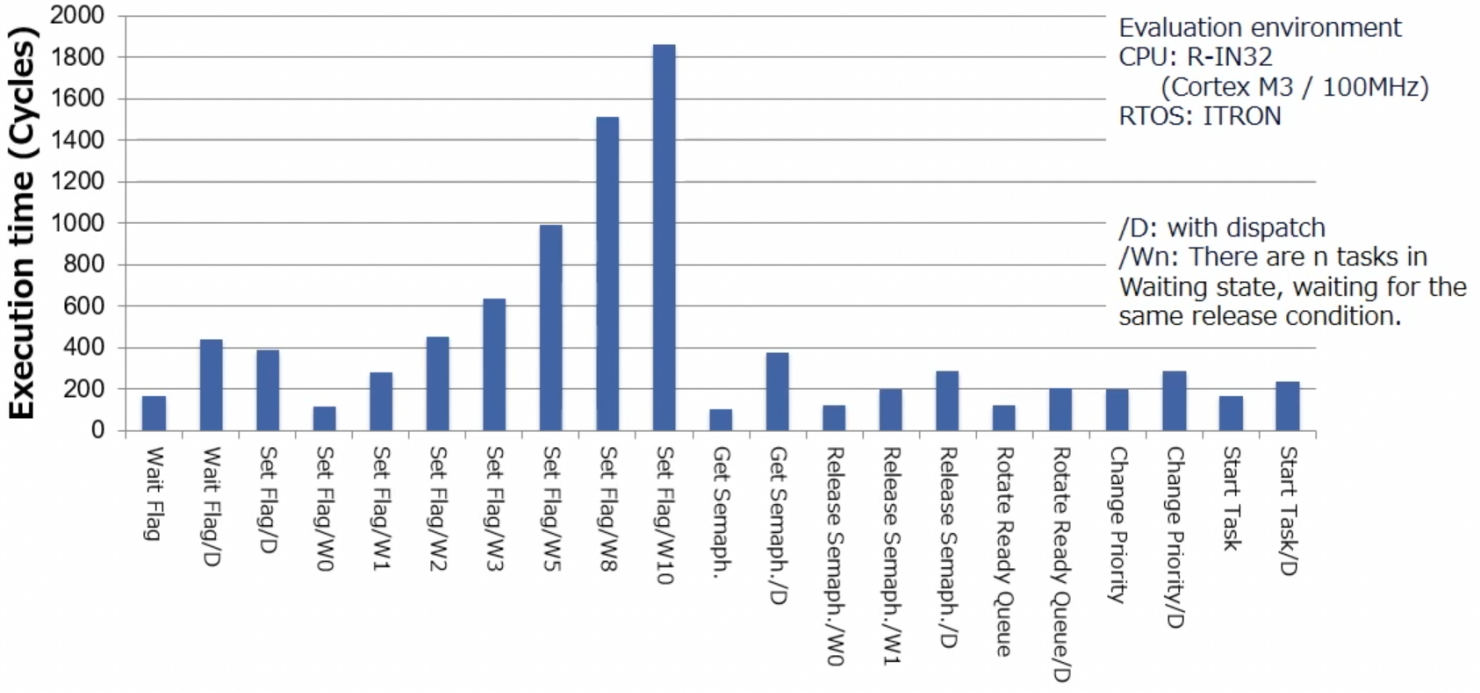
Du point de vue des tâches :

Plus le «*tick interval*» est court ↘, plus le traitement du «*Tick*» ↗ est vu comme une surchage du CPU.

⟹Diminuer de trop le «*tick interval*» diminue l'utilisation efficace du CPU.

L'efficacité de l'utilisation du CPU est un rapport entre le temps alloué à l'exécution de l'application par rapport au temps consommé par l'OS.

Temps d'exécution des différentes APIs

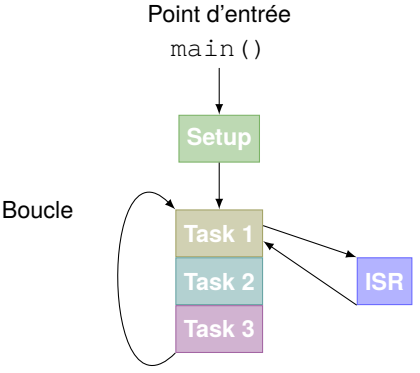


L'unité de temps est le nombre de **cycle processeur** : une instruction prend en général de 100 à cycles du CPU.

On remarque que les APIs utilisant les «Event flag» *augmentent considérablement le temps de traitement en fonction du nombre de tâches les utilisant.*

- Un nombre énorme de files sont nécessaires (dépend du nombre de priorités) ;
  - Cela dépend du système, mais peut atteindre plusieurs milliers.
  - consomme de la mémoire et augment le temps de traitement (parcours des files et traitement).
- Le temps de traitement des files prend du temps :
  - ce temps de traitement peut changer en fonction de l'état interne du système :
    - le traitement des files liées au «*Set_Flag Event*» est particulièrement long ;
  - Durant ce temps, la gestion des interruptions est désactivée
    $\Longrightarrow$allonge le temps de réponse
    $\Longrightarrow$diminue la réactivité du système.
- Le temps de gestion du Tick :
  - le «*tick interval*» est l'unité de temps du RTOS ;
- les problèmes liés au Tick :
  - les applications, tâches, sont périodiquement interrompues ce qui diminue l'efficacité de l'utilisation du CPU ;
  - la gestion du Tick désactive les interruptions$\Longrightarrow$la réactivité à ces interruptions diminue ↘.
  - lorsque le «*tick interval*» diminue ↘, l'éfficacité diminue ↘

La «*superloop*»



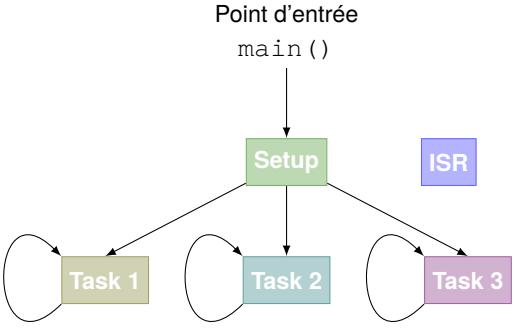
Avantages :

- ▷ facile à écrire, peu de surcoût en mémoire et cycle du CPU, facile à débuguer ;
- ▷ les tâches s'exécutent toujours dans le même ordre dans une boucle infinie ;
- ▷ une tâche peut lire un capteur, une autre faire des calculs sur ces mesures et la dernière faier un affichage sur un écran ;
- ▷ l'utilisation d'ISR permet d'obtenir des temps précis et prédictibles ;

Inconvénients :

- ▷ si la tâche 2 prend un temps plus long et que la tâche 3 met à jour l'écran alors un «*lag*» peut arriver ;
- ▷ si une tâche lit un capteur et prend du retard, alors elle peut rater des mesures.

Le RTOS



Avantages :

- ▷ les tâches peuvent être exécutées de manière concurrente ;
- ▷ avec plusieurs cœurs alors elle peuvent être exécutées en parallèle ;
- ▷ l'utilisation de priorité peut favoriser l'exécution d'une tâche par rapport à une autre ;

Inconvénients :

- ▷ plus dur à programmer correctement ;
- ▷ les ISR doivent avoir une priorité supérieure à toutes les tâches.

ATmega 328p
- 16 MHz
- 32 kB flash
- 2 kB RAM

STM32L476RG
- 80 MHz
- 1 MB flash
- 128 kB RAM

ESP-WROOM-32
- 240 MHz (dual core)
- 4 MB flash
- 520 kB RAM

Super Loop → RTOS

- ▷ La puissance du matériel permet de faire tourner un RTOS : plus de mémoire et de cycle de CPU à «*gaspiller*» sur un ordonnanceur ;
- ▷ la disponibilité de communication sans fil comme BLE et WiFi recquiert l'utilisation d'un RTOS ;
- ▷ l'utilisation de tâches concurrentes permet également de distribuer le travail dans une équipe de développeurs.

Les avantages liés à l'utilisation de RTOS

- le logiciel est **modulaire** : chaque partie de logiciel s'occupe d'une tâche bien identifiée et isolée ;
- ces parties deviennent des **composants réutilisables** ;
- le logiciel est **plus sûr**, «*reliable*» : plus facile d'isoler et de corriger les erreurs ;
- **le développement est plus efficace**.

Un RTOS particulier : FreeRTOS

`https://www.freertos.org`
Acheté et développé par Amazon.
«*Has a minimal ROM, RAM and processing overhead. Typically an RTOS kernel binary image will be in the region of 6K to 12K bytes*».

FreeRTOS

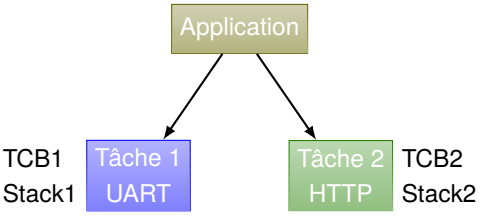
Developed in partnership with the world's leading chip companies over a 15-year period, and now downloaded every 175 seconds, FreeRTOS is a market-leading real-time operating system (RTOS) for microcontrollers and small microprocessors. Distributed freely under the MIT open source license, FreeRTOS includes a kernel and a growing set of libraries suitable for use across all industry sectors. FreeRTOS is built with an emphasis on reliability and ease of use.

Soit une **application** constituée de **deux tâches** :

1. gestion d'un port série sous interruption ;
2. gestion d'un serveur HTTP ;

Dans **FreeRTOS**, chaque tâche :

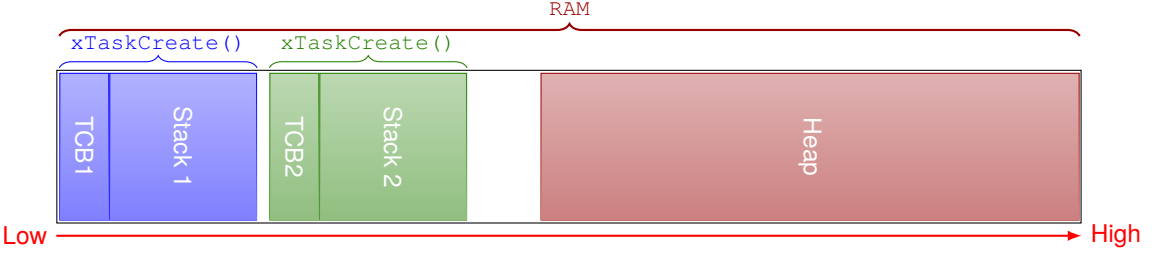
- ☐ est gérée par un TCB, «*Task Control Block*» ;
- ☐ dispose d'une pile, «*stack*», pour ses variables et ses appels de fonction.




```
1 void app_main()
2 {
3     static httpd_handle_t server = NULL;
4    init_uart();
5     // Creation d'une tache pour la gestion de l'UART par interruption
6     xTaskCreate(uart_event_task, "uart_event_task", 8192, NULL, 12, NULL);
7    // Creation d'une tache pour le serveur HTTP
8     xTaskCreate(https_get_task_alt, "https_get_task", 8192, NULL, 5, &tache_https);
9 }
```

Lors de la création de la tâche avec `xTaskCreate()`, la tâche reçoit une **zone fixe** de mémoire pour sa pile.

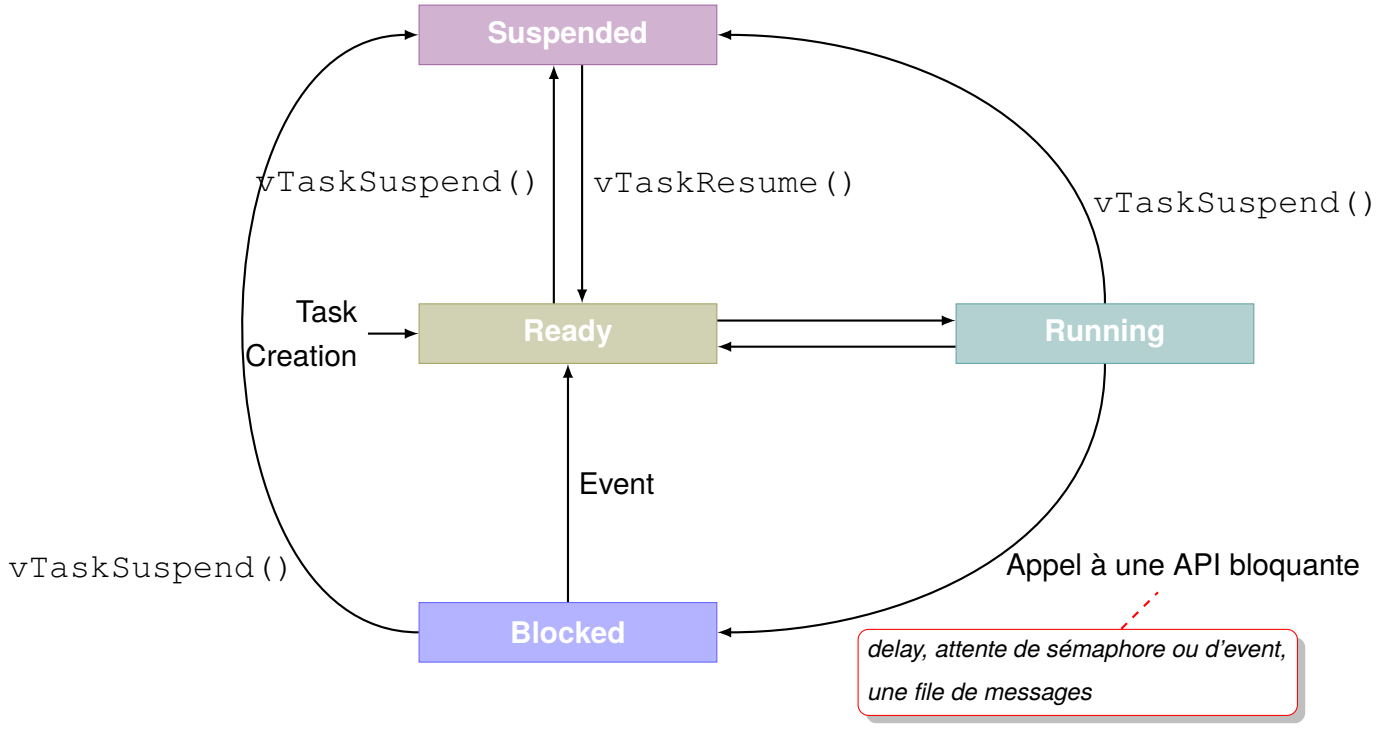
⟹ un **arrêt de l'application** survient si la tâche **dépasse** cette taille de pile allouée.



La **mémoire RAM du composant embarqué** est divisée en deux parties :

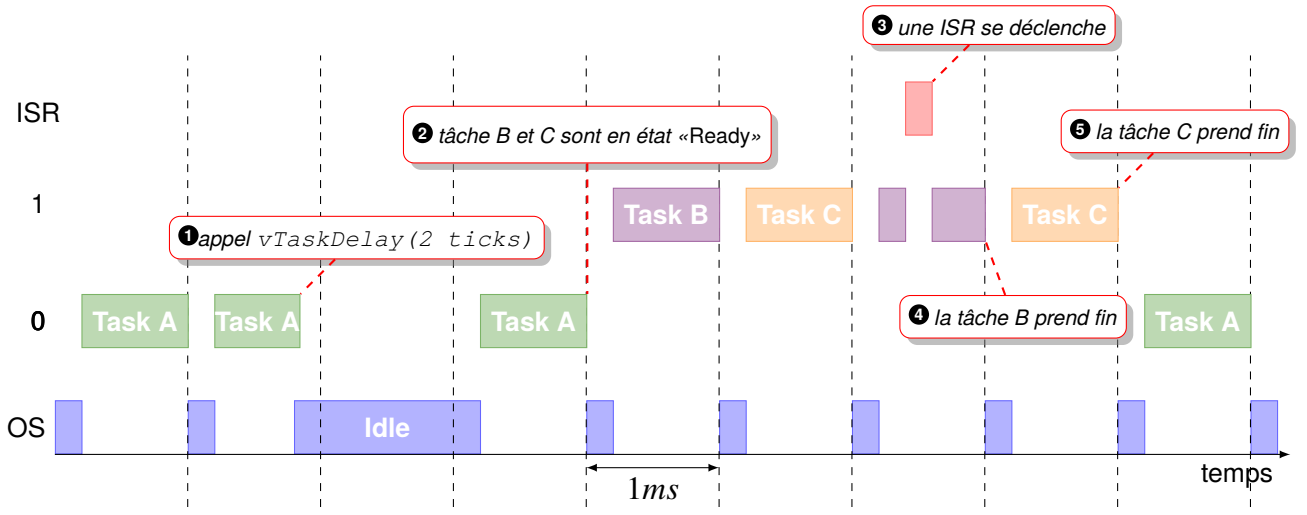
- ▷ une allouée aux différentes **tâches/piles** ;
- ▷ la seconde affectée au **tas**, «*heap*» de l'application complète.

Les différents états d'une tâche en FreeRTOS

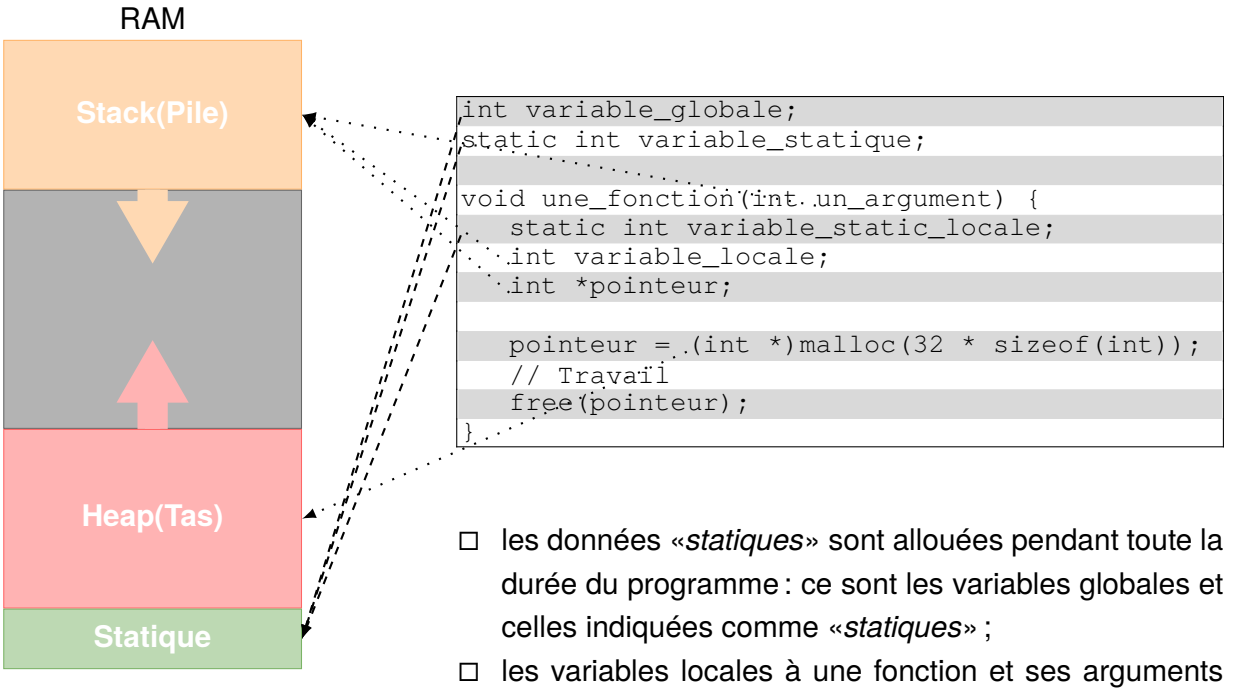


Par rapport au RTOS générique

- «*Dormant*» ⟹ «*Suspended*» ;
- «*Waiting*» ⟹ «*Blocked*» ;

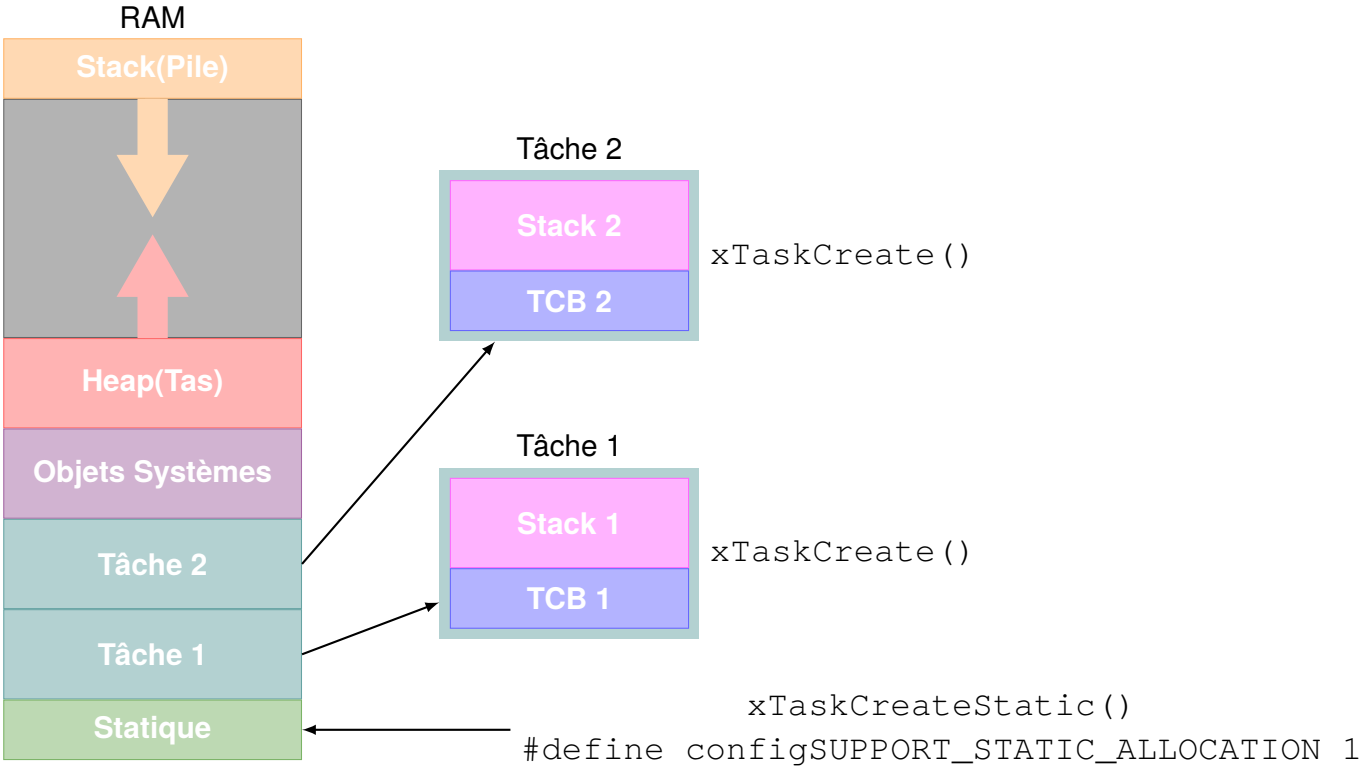


- le temps est découpé en «*slice*» de $1ms$: un «*tick*» ;
- à chaque «*slice*», l'OS tourne pour sélectionner la prochaine tâche à exécuter⟹ordonnanceur :
  - il choisit la tâche de plus haute priorité : «*Task A*» est de faible priorité mais elle est seule ;
  - en ❶, la tâche se met en attente et l'ordonnanceur n'a pas d'autre tâche à exécuter, «*Ready*»,⟹«*Idle*» ;
  - en ❷, la tâche A fait passer les tâches B et C en état «*Ready*»⟹elles ont une priorité identique et supérieure à celle de A⟹l'ordonnanceur alterne l'exécution de B et C ;
    ⟹ préemption.
  - en ❸, l'ISR se déclenche : elle a une priorité supérieure à toutes les tâches ;
  - en ❹ et ❺, l'ISR se déclenche : les tâches B et C terminent⟹ la tâche A peut de nouveau s'exécuter.



La pile, comme le tas, peuvent grossir

- les données «*statiques*» sont allouées pendant toute la durée du programme : ce sont les variables globales et celles indiquées comme «*statiques*» ;
- les variables locales à une fonction et ses arguments sont allouées dans la **pile** : elles sont valables jusqu'au retour de la fonction ;
- le **tas** est utilisé pour l'allocation dynamique de mémoire avec `malloc`. C'est au programmeur de libérer la mémoire avec un `free` sous peine de «*memory leak*».



Le choix de la taille allouée depuis le **tas** à une tâche comme pile locale à cette tâche est indiquée lors de sa création : il faut faire attention à en allouer suffisamment pour ne pas «*écraser*» des zones mémoires allouées à d'autres tâches.

Attention

- ▷ lors de l'allocation de mémoire depuis le tas à une tâche, l'OS choisit le bloc de mémoire contigüe le plus gros qui satisfait la demande ;
- ▷ si on répète souvent des opérations d'allocation et désallocation $\Longrightarrow$ **fragmentation** de la mémoire du tas et accélération de la **collision** du tas avec la pile.

Solution : différents algorithmes d'allocation mémoire dans le tas

- ☐ «*heap_1*» : le plus simple, ne permet pas à la mémoire d'être libérée ;
- ☐ «*heap_2*» : protège `malloc` et `free` contre l'exécution concurrente «*threadsafe*» ;
- ☐ «*heap_3*» : permet la désallocation mais ne fait pas l'aggrégation des blocs de mémoire contigüs ;
- ☐ «*heap_4*» : réalise l'aggrégation des blocs de mémoire contigüs et autorise le placement en donnant l'adresse absolue ;
- ☐ «*heap_5*» : similaire au «*heap_4*» mais avec la capacité de répartir le tas entre différents blocs mémoire non adjacents.

Remarques :

- ▷ «*heap_1*» est moins utile depuis que FreeRTOS supporte l'allocation statique de mémoire;
- ▷ «*heap_2*» est obsolète et c'est «*heap_4*» qui est préféré.

Exemple : utilisation d'une LED avec deux tâches de priorités différentes


```
// Utilisation d'un seul core pour la démo
#if CONFIG_FREERTOS_UNICORE
static const BaseType_t app_cpu = 0;
#else
static const BaseType_t app_cpu = 1;
#endif

// LED rates
static const int rate_1 = 500;  // ms
static const int rate_2 = 323;  // ms

// Pins
static const int led_pin = LED_BUILTIN;

// Our task: blink an LED at one rate
void toggleLED_1(void *parameter) {
  while(1) {
    digitalWrite(led_pin, HIGH);
    vTaskDelay(rate_1 / portTICK_PERIOD_MS);
    digitalWrite(led_pin, LOW);
    vTaskDelay(rate_1 / portTICK_PERIOD_MS);
  }
}

// Our task: blink an LED at another rate
void toggleLED_2(void *parameter) {
  while(1) {
    digitalWrite(led_pin, HIGH);
    vTaskDelay(rate_2 / portTICK_PERIOD_MS);
    digitalWrite(led_pin, LOW);
    vTaskDelay(rate_2 / portTICK_PERIOD_MS);
  }
}
```

Attente de la part la tâche : passage à l'OS et son ordonnanceur

Allumage de la LED pendant un certain délais

Exemple : utilisation d'une LED avec deux tâches de priorités différentes


```
void setup() {

  // Configure pin
  pinMode(led_pin, OUTPUT);

  // Task to run forever
  xTaskCreatePinnedToCore(  // Use xTaskCreate() in vanilla FreeRTOS
              toggleLED_1,  // Function to be called
              "Toggle 1",   // Name of task
              1024,         // Stack size (bytes in ESP32, words in FreeRTOS)
              NULL,         // Parameter to pass to function
              1,            // Task priority (0 to configMAX_PRIORITIES - 1)
              NULL,         // Task handle
              app_cpu);     // Run on one core for demo purposes (ESP32 only)

  // Task to run forever
  xTaskCreatePinnedToCore(  // Use xTaskCreate() in vanilla FreeRTOS
              toggleLED_2,  // Function to be called
              "Toggle 2",   // Name of task
              1024,         // Stack size (bytes in ESP32, words in FreeRTOS)
              NULL,         // Parameter to pass to function
              1,            // Task priority (0 to configMAX_PRIORITIES - 1)
              NULL,         // Task handle
              app_cpu);     // Run on one core for demo purposes (ESP32 only)

  // If this was vanilla FreeRTOS, you'd want to call vTaskStartScheduler() in
  // main after setting up your tasks.
}

void loop() {
  // Do nothing
  // setup() and loop() run in their own task with priority 1 in core 1
  // on ESP32
}
```

Exemple de lancement de tâches avec des priorités différentes


```
// Utilisation d'un seul core pour la démo
#if CONFIG_FREERTOS_UNICORE
static const BaseType_t app_cpu = 0;
#else
static const BaseType_t app_cpu = 1;
#endif

const char message[] = "Ceci est un message de l'ESP32 depuis la tache 1";

// Handles de tâches
static TaskHandle_t task1 = NULL;
static TaskHandle_t task2 = NULL;

// Les deux tâches

// tâche 1 : affiche sur le port série avec une priorité basse
void startTache1(void *parameter) {
  int longueur_msg = strlen(message);

  // Affichage sur la sortie série
  while(1) {
    Serial.println();
    for(int i = 0; i < longueur_msg; i++) {
       Serial.print(message[i]);
           vTaskDelay(100 / portTICK_PERIOD_MS);
    }
    Serial.println();
    vTaskDelay(1000 / portTICK_PERIOD_MS);
  }
}
```

Exemple de lancement de tâches avec des priorités différentes


```
// tâche 2 : affiche sur le port série avec une priorité haute
void startTache2(void *parameter) {
  // Affichage sur la sortie série
  while(1) {
    Serial.print('*');
    vTaskDelay(100 / portTICK_PERIOD_MS);
  }
}

// setup correspond à sa propre tâche de priorité 1 sur le core 1
void setup() {

  // Configure pin
  Serial.begin(300);

  // Attente pour éviter de rater la sortie sur le port série
  vTaskDelay(1000 / portTICK_PERIOD_MS);
  Serial.println();
  Serial.println("Demo taches et priorites FreeRTOS");
  Serial.print("Core :");
  Serial.print(xPortGetCoreID());
  Serial.print(" avec priorite :");
  Serial.println(uxTaskPriorityGet(NULL));

  // lancement des taches
  xTaskCreatePinnedToCore(  // Use xTaskCreate() in vanilla FreeRTOS
              startTache1,  // Function to be called
              "Tache 1",    // Name of task
              1024,         // Stack size (bytes in ESP32, words in FreeRTOS)
              NULL,         // Parameter to pass to function
              1,            // Task priority (0 to configMAX_PRIORITIES - 1)
              &task1,       // Task handle
              app_cpu);     // Run on one core for demo purposes (ESP32 only)
```

Obtenir le Core sur lequel tourne la tâche

Obtenir la priorité de la tâche

Exemple de lancement de tâches avec des priorités différentes


```
xTaskCreatePinnedToCore(  // Use xTaskCreate() in vanilla FreeRTOS
            startTache2,  // Function to be called
            "Tache 2",    // Name of task
            1024,         // Stack size (bytes in ESP32, words in FreeRTOS)
            NULL,         // Parameter to pass to function
            2,            // Task priority (0 to configMAX_PRIORITIES - 1)
            &task2,       // Task handle
            app_cpu);     // Run on one core for demo purposes (ESP32 only)

// If this was vanilla FreeRTOS, you'd want to call vTaskStartScheduler() in
// main after setting up your tasks.
}

void loop() {
// Suspension de la tache de plus haute priorité de temps en temps
for(int i=0; i < 6; i++) {
  vTaskSuspend(task2);
  vTaskDelay(2000 / portTICK_PERIOD_MS);
  vTaskResume(task2);
  vTaskDelay(2000 / portTICK_PERIOD_MS);
}
// Détruire la tache de plus basse priorité
if (task1 != NULL) {
  vTaskDelete(task1);
  task1 = NULL;
}
}
```

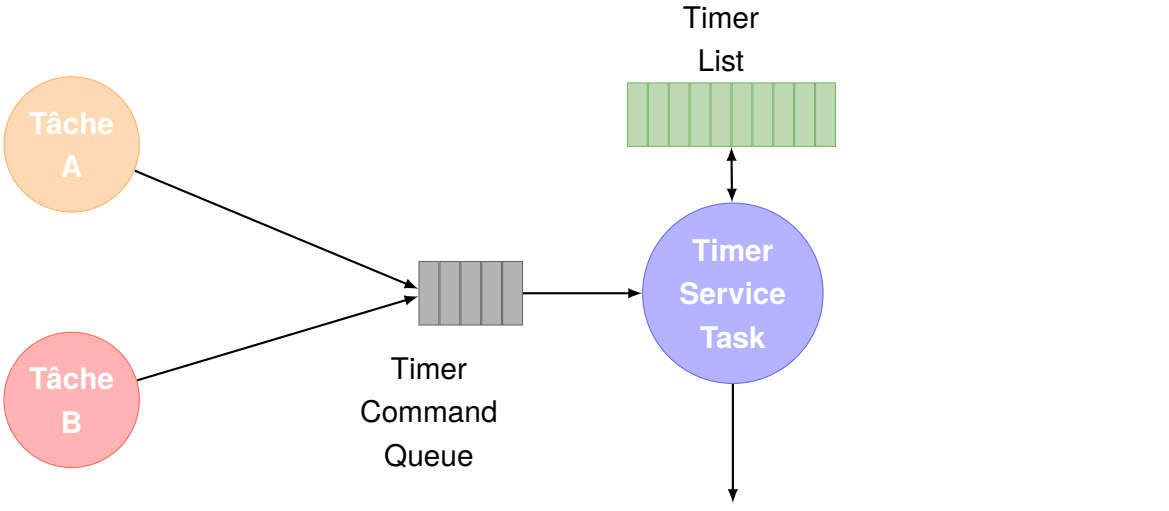
Ici, on se sert de la fonction «`loop`» qui correspond à une tâche de priorité 1

Pour l'utilisation des Sémaphores

```
#include <semphr.h>

static SemaphoreHandle_t mutex;

mutex = xSemaphoreCreateMutex();

// Prendre la Sémaphore en mode bloquant
// xSemaphoreTake(mutex, portMAX_DELAY);

// Prendre la Sémaphore en mode Polling
if (xSemaphoreTake(mutex,0) == pdTrue) {
   // Section Critique
   // Libérer la Sémaphore
   xSemaphoreGive(mutex);
}
```

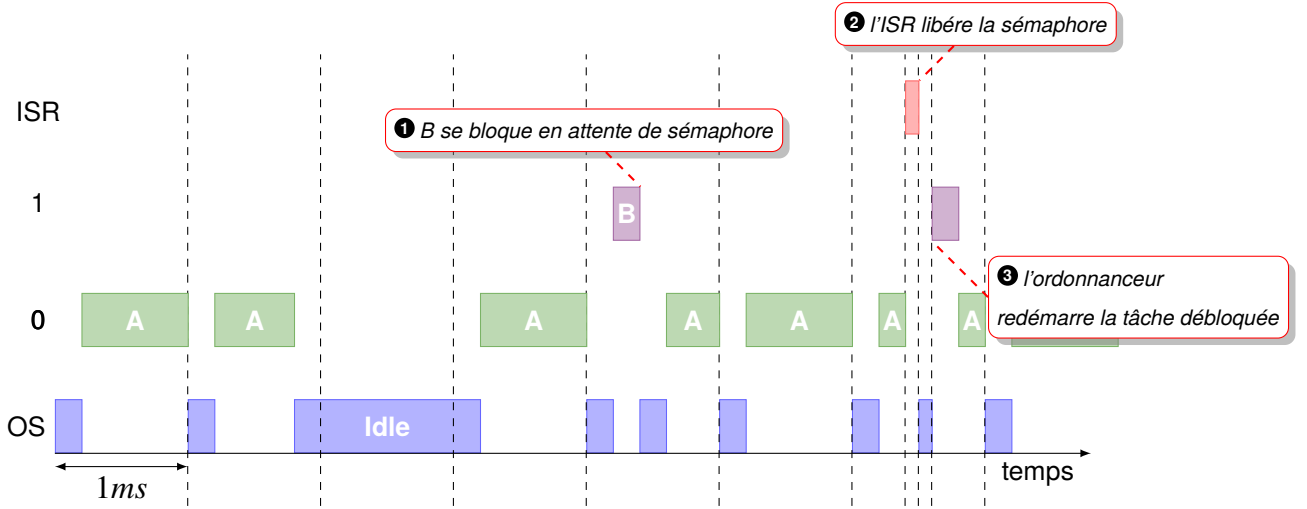
Pour l'utilisation des files de messages

```
static QueueHandle_t file_messages;
    // Lecture du message en mode Polling
     if (xQueueReceive(file_messages, (void *)&element, 0) == pdTRUE) {
      Serial.println(element);
    }

 // Pour l'envoi de message dans la file
 if (xQueueSend(file_messages, (void *)&n, 10) != pdTrue){
    Serial.println("File pleine");
  }
```




```
static TimerHandle_t mon_timer = NULL;

void mon_timer_callback(TimerHandle_t t) {
  Serial.println("Le timer a expire");
}

  mon_timer = xTimerCreate(
                "Mon timer",                  // Nom du timer
            2000 / portTICK_PERIOD_MS,   // période du timer en ticks
            pdFALSE,                     // Auto-reload
            (void *) 0,                  // Timer ID
            mon_timer_callback);         // callback function
 xTimerStart(mon_timer, portMAX_DELAY);
```

Un appel à `xTimerStart` permet de redémarrer un timer qui n'a pas encore expiré.



- En ❶ la tâche B se bloque en attente de la sémaphore⟹l'oronnanceur choisit la seule tâche dans un état «*Ready*» : la tâche A ;
- la tâche A poursuit son exécution jusqu'à ce que l'ISR se déclenche en ❷⟹l'ISR change la valeur d'une variable globale par exemple, puis elle libére la sémaphore et rend la main à l'ordonnanceur ;
- l'ordonnanceur déclenche B qui a été débloquée et qui est de priorité supérieure à celle de A ;
- une fois son travail terminé, B peut se rebloquer en attente de la sémaphore et A peut reprendre son exécution.

*On appelle cela une **délégation d'interruption** :*

- *la prise en compte de l'interruption est très court ;*
- *le travail lié à l'interruption est déléguée à une tâche qui elle-même peut être réalisé en concurrence avec d'autres tâches.*

Pour l'installation d'une ISR sur un «*timer*» matériel :

```
static const uint16_t timer_divider = 80;
static const uint64_t timer_max_count = 1000000;

static const int adc_pin = A0;

static hw_timer_t *timer = NULL;
static volatile uint16_t val;
static SemaphoreHandle_t bin_sem = NULL;

void IRAM_ATTR onTimer() {
  BaseType_t task_woken = pdFALSE;
  val = analogRead(adc_pin);

  xSemaphoreGiveFromISR(bin_sem, &task_woken);

  if (task_woken) { // Si une tâche a été ré
veillée
    portYIELD_FROM_ISR();
  }
}
```

Installation du «*timer*»

```
// Création/démarrage du timer (num, divider, coun
tUp)
timer = timerBegin(0, timer_divider, true);

//Attache ISR et timer (timer, function, edge)
timerAttachInterrupt(timer, &onTimer, true);

// Valeur du compteur l'ISR doit être déclenchée
timerAlarmWrite(timer, timer_max_count, true);

// Autoriser l'ISR à se déclencher
timerAlarmEnable(timer);
```

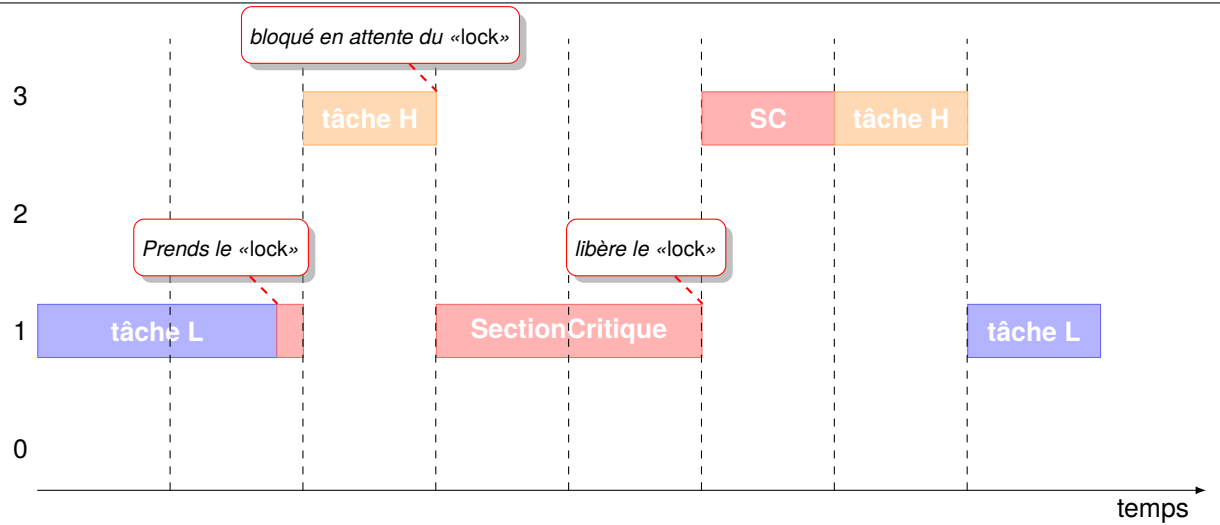
Tâche à laquelle on délègue la gestion de l'interruption :

```
void afficherValeurs(void *parameters) {
   while(1) {
     xSemaphoreTake(bin_sem, portMAX_DELAY);
     Serial.println(val);
     }
}
```

Déclenchement de la tâche en priorité 2 :

```
xTaskCreatePinnedToCore(
            afficherValeurs,
            "afficher Messages",
            1024,
            NULL,
            2,
            NULL,
            app_cpu);
```

Inversion de priorités



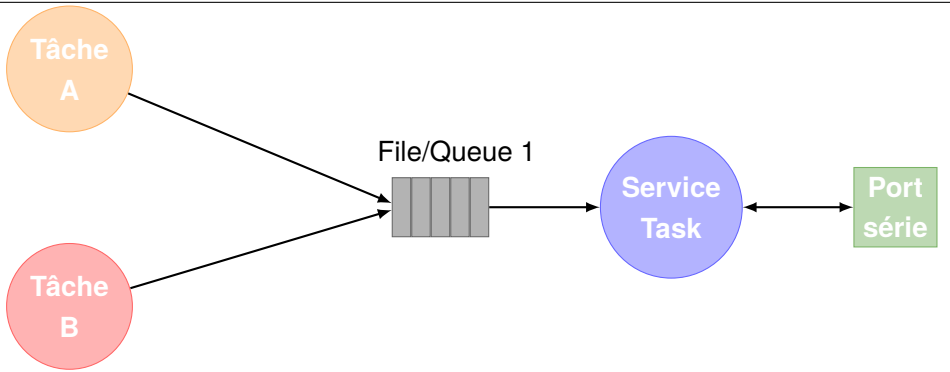
- le «*lock*» : peut être une sémaphore ou un mutex ;
- la tâche L de basse priorité prends le «*lock*» ;
- elle est interrompue par la tâche H de plus haute priorité ;
- à un moment donné la tâche H essaye d'entrer dans la même «*section critique*», mais elle ne peut obtenir le «*lock*» qui est détenu par la tâche L⟹elle est suspendue ;
- l'ordonnanceur retourne à l'exécution de la tâche L qui peut finir sa section critique et libérer le «*lock*» ;

⟹**Inversion de priorité** : la tâche L s'exécute alors que la tâche H **de plus haute priorité** attend !

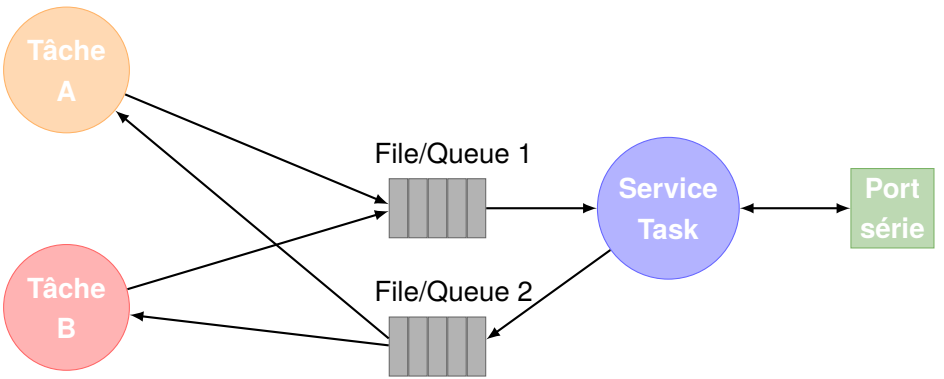
Le temps d'attente de la tâche H est contraint par le temps d'exécution de la tâche L dans la «*section critique*».

Solution ? ne pas utiliser de section critique, disposer d'un multi-cœur ou limiter la durée de la «*SC*».

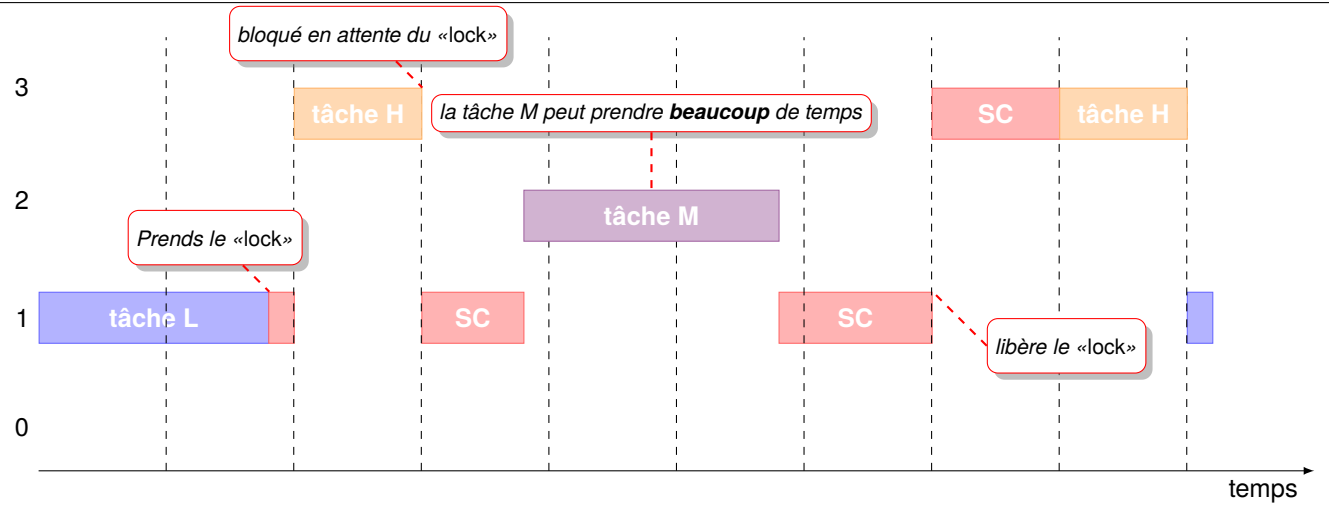
Solution : utiliser une «*tâche de service*»



- le «*port série*» est une ressource critique⟹il est affecté à une seule tâche : la «*Service Task*» ;
- les tâches désirant utiliser le port série passe par cette tâche : elles envoient un message à cette tâche au travers d'une file ou «*queue*» et seule la «*service task*» peut envoyer sur le port série ;



Si on veut lire depuis le «*port série*» : on ajoute une seconde file.



- la tâche L de basse priorité prends le «*lock*» ;
- elle est interrompue par la tâche H de plus haute priorité ;
- à un moment donné la tâche H essaye d'entrer dans la même «*section critique*», mais elle ne peut obtenir le «*lock*» qui est détenu par la tâche L⟹elle est suspendue ;
- l'ordonnanceur retourne à l'exécution de la tâche L, mais la tâche M s'active et s'exécute car elle a une priorité plus grande que L...

  ⟹**Inversion de priorité** : les tâches L et M s'exécutent alors que la tâche H **de plus haute priorité** attend !

  Attention :
  - ◇ M prend **beaucoup de temps** et bloque la tâche H en bloquant la sortie de la tâche L de la «*SC*» ;
  - ◇ M prend **tellement de temps**⟹le «*Watchdog*» **réinitialise** le «*système complet*» !

Le temps d'attente de la tâche H dépend du temps d'exécution de la tâche M, ce qui peut conduire à un «*reset*».
Solution ? Augmenter la priorité de L **temporairement** pendant qu'elle a le «*lock*» pour éviter l'interruption par M.

Que choisir ?

Bare-metal programming

- Little or no software overhead
- Low power requirement
- High control of hardware
- Single-purpose or simple applications, hardware-dependent
- Strict timing (e.g. motor control)

Real-time Operating System (RTOS)

- Scheduler overhead
- More powerful microcontroller required
- High control of hardware
- Multithreading, some common libraries
- Multiple tasks: networking, user interface, etc.

Embedded General Purpose Operating System (GPOS)

- Large overhead (scheduler, memory management, background tasks, etc.)
- Microprocessor usually required (and often external RAM+NVM)
- Low direct control of hardware (files or abstraction layers)
- Multiple threads and processes, many common libraries (portable application code)
- Multiple complex tasks: networking, filesystem, graphical interface, etc.

Les communications IoT ou M2M

IIoT, Industrial Internet of Things

M2M : communications entre machines, c-à-d communications temps réel de données sans intervention humaine :

- ▷ télémétrie ;
- ▷ information temps réel en cas d'échec ;
- ▷ contrôle à distance de l'état d'une machine ;
- ▷ acquisition temps réel de données.

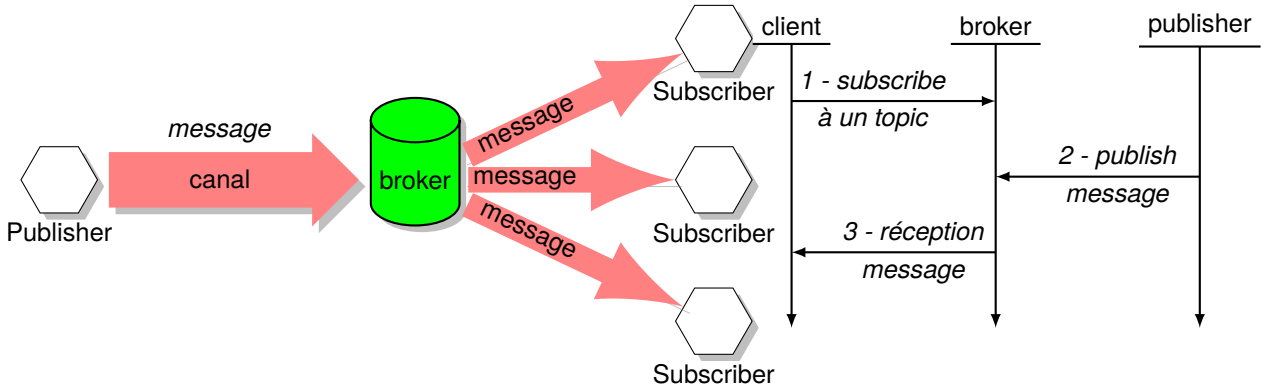
Différents protocoles

Protocol Name	Transport Protocol	Messaging Model	Security	Best-Use Cases	Architecture
AMPQ	TCP	Publish/Subscribe	High-Optional	Enterprise integration	P2P
CoAP	UDP	Request/Response	Medium-Optional	Utility field	Tree
DDS	UDP	Publish/Subscribe Request/Response	High-Optional	Military	Bus
MQTT	TCP	Publish/Subscribe Request/Response	Medium-Optional	IoT messaging	Tree
UPnP	-	Publish/Subscribe Request/Response	None	Consumer	P2P
XMPP	TCP	Publish/Subscribe Request/Response	High-Compulsory	Remote management	Client/Server
ZeroMQ	UDP	Publish/Subscribe Request/Response	High-Optional	CERN	P2p

- Ports réseau :
  - **1883**: This is the default MQTT port. 1883 is defined at IANA as **MQTT over TCP**.
  - **8883**: This is the default MQTT port for **MQTT over TLS**. It's registered at IANA for **Secure MQTT**.

```
xterm
sudo nmap -sS -sV -v -p 1883,8883 --script mqtt-subscribe p-fb.net
```

- Publish/Subscribe modèle :

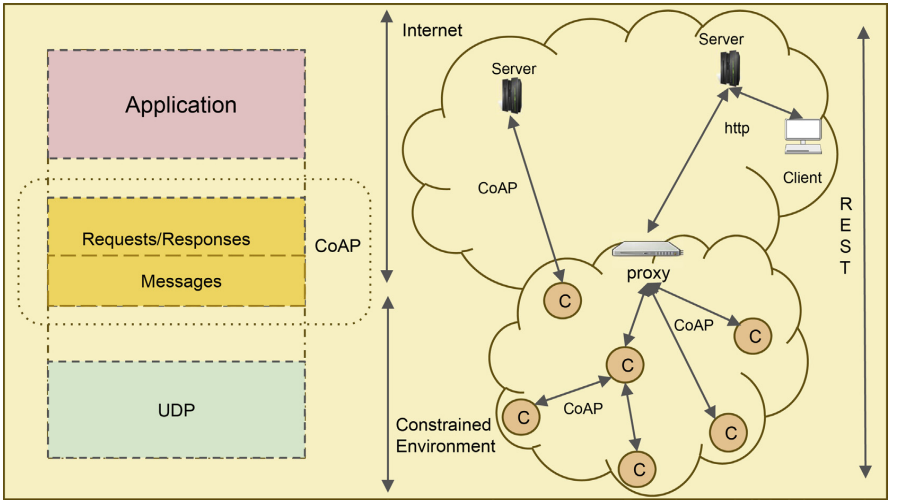


There are a number of **threats** that solution providers should consider.

For example:

- Devices could be **compromised**
- Data at rest in Clients and Servers might be **accessible**
- Protocol behaviors could have **side effects** (e.g. "timing attacks")
- **Denial of Service** (DoS) attacks
- Communications could be **intercepted**, altered, re-routed or disclosed
- Injection of **spoofed Control Packets**

- le message peut être au format **JSON** ;
- un message est identifié par des **topics** qui sont organisés en arborescence où chaque niveau est séparé par un «`/`» :
  - l'opérateur # permet de sélectionner l'ensemble des sous-niveaux : «`capteurs/temperature/maison/#`» permet d'obtenir :
    - `capteurs/temperature/maison/couloir`
    - `capteurs/temperature/maison/chambre`
    - `capteurs/temperature/maison/chambre/fenêtre`
    - utiliser juste «`#`» renvoie la totalité des topics ;
    - `capteurs/temperature/maison\#` est invalide ;
    - `capteurs/\#/maison/` est invalide ;
  - l'opérateur + permet de «*matcher*» un seul niveau : «`capteurs/temperature/maison/+`» permet d'obtenir :
    - `capteurs/temperature/maison/couloir`
    - `capteurs/temperature/maison/chambre`
    - «`+`» est valide ;
    - «`+/maison/#`» est valide ;
    - «`capteurs+`» est invalide ;
    - «`capteurs/+/maison`» est valide ;
  - «$SYS/» permet d'obtenir des informations sur le serveur MQTT.
- **sécurité** : le message est en clair, mais la communication peut avoir lieu en TLS/SSL ;
- **QoS**, «*Quality of Service*» :
  - QoS 0, «At Most Once» : un message est délivré au plus une fois ou pas délivré ;
  - QoS 1, «At least Once» : un message est délivré au moins une fois et si le récepteur n'acquitte pas la réception le message est transmis de nouveau ;
  - QoS 2, «Exactly only Once» : un message est délivré une seule fois ;
- MQTT solutions are often deployed in **hostile communication environments**.
  In such cases, implementations will often need to provide mechanisms for:
  - **Authentication** of users and devices
  - **Authorization** of access to Server resources
  - **Integrity** of MQTT Control Packets and application data contained therein
  - **Privacy** of MQTT Control Packets and application data contained therein



- asynchrone et basé sur UDP, port 5683, RFC 7252, `http://coap.technology` ;

```
xterm
nmap -p U:5683 -sU --script coap-resources p-fb.net
```

- peut fonctionner pour des environnements < 10ko ;
- Quatre types de message :
  - Acknowledgement
  - Reset
  - Confirmable
  - Non-Confirmable : envoyer des requêtes qui n'ont pas besoin de «*reliability*»
- les requêtes sont proches du modèle REST : GET, POST, PUT et DELETE
- le contenu du message peut être au format JSON ;
- le chiffrement peut être basé sur DTLS.

CRUD

HTTP	Usage	SQL
POST	«*C*»reates information	INSERT
GET	«*R*»etrieves information	SELECT
PUT	«*U*»pdates information	UPDATE
DELETE	«*D*»eletes information	DELETE

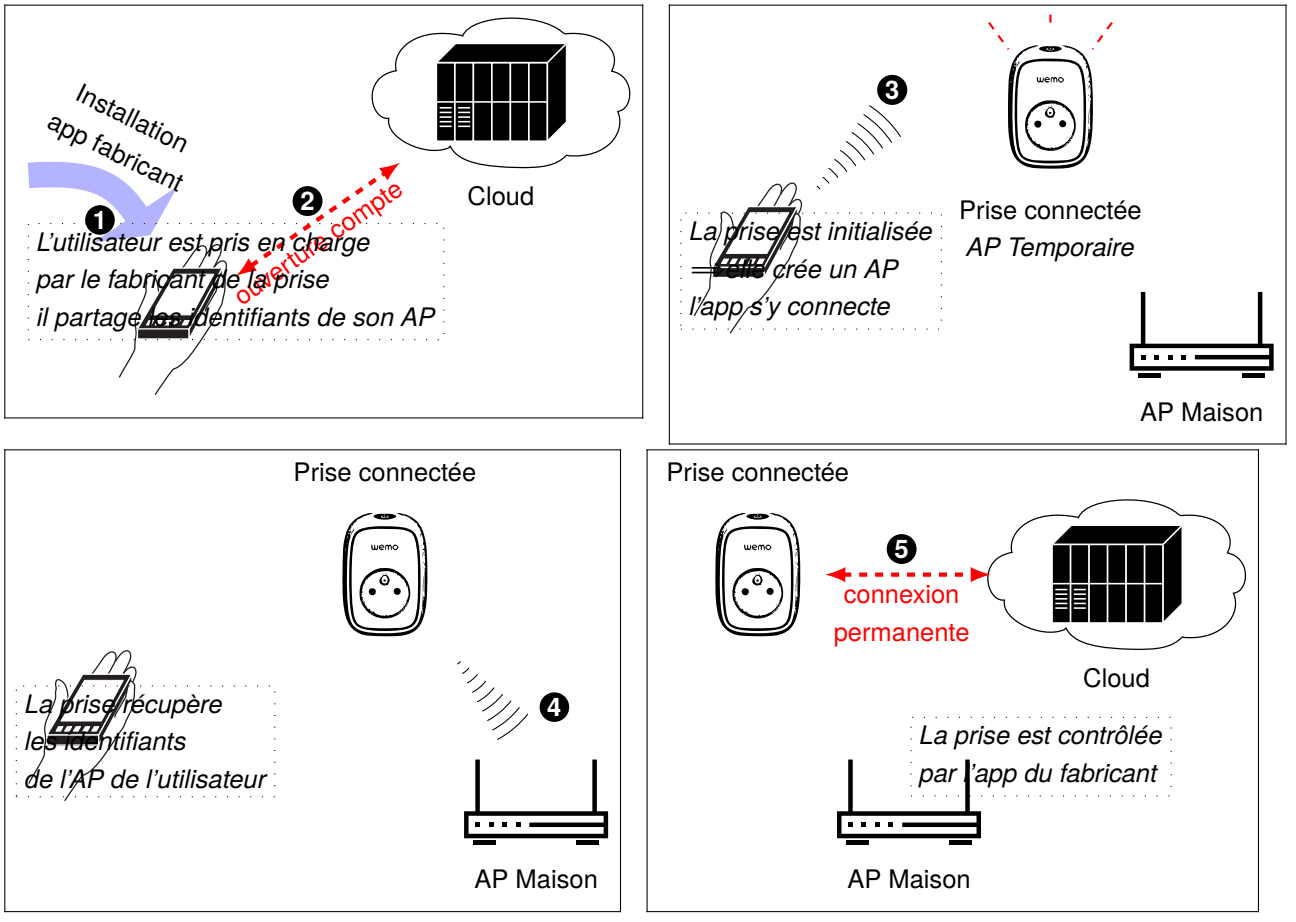
Différence entre PUT et POST ? GET récupère une ressource donnée par son URL et PUT permet de la mettre à jour. Comment créer un objet quand une URL pour y accéder n'existe pas encore ? On utilise POST : POST permet de créer un objet en demandant au containeur parent de créer un élément enfant et l'on reçoit en retour l'URL exacte de cet élément.

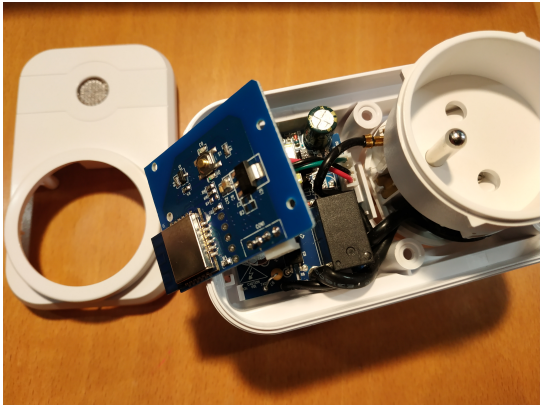
Architecture REST, «*Representational State Transfer*»

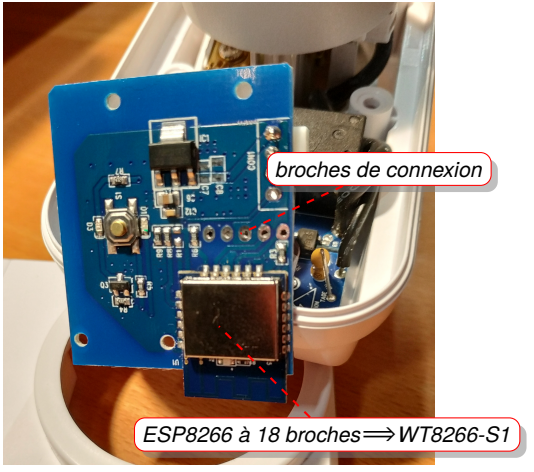
Règle	**Explication**
Accessible	tout est vu comme une ressource qui peut être accessible au travers d'une URI/URL
Sans état	le client et le serveur ne peuvent être désynchronisés. *Contre exemple : lors d'une demande de téléchargement dans un échange FTP, le client et le serveur doivent s'entendre sur le répertoire courant où l'opération aura lieu ⟹ un premier échange ne peut être fait par un serveur FTP puis le second par un autre serveur FTP ⟹ non* «scalable»
Sûr	l'information peut être retrouvée sans causer d'effet de bord *Récupérer une page plusieurs fois ne doit pas avoir d'effet sur le serveur*
Idempotent	la même action peut être réalisée plusieurs fois sans effet de bord *Si une requête correspond à «incrémenter» une valeur de 5 à 6, alors elle doit demander 6 ⟹ elle ne doit pas incrémenter une valeur courante !*
Uniforme	utiliser une interface simple et connue de tous ⟹ HTTP et le CGI, «*Common Gateway Interface*».

Exemple d'IoT : Alexa et une prise connectée
Le matériel, firmware et protocoles

Scénario de l'intégration d'une prise connectée à la maison

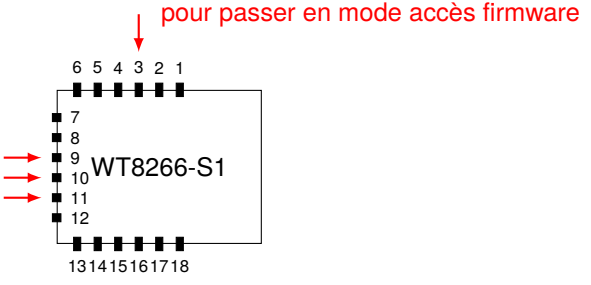






Identifier les broches du WT8266-S1 à l'aide d'un multimètre et de la doc constructeur

Pin	Info
3	IO0
9	URXD
10	GND
11	UTXD



Pour récupérer le firmware présent dans la mémoire flash de 16Mb ou 2MB avec un adaptateur USB/série :

xterm

```
$ esptool.py --port /dev/ttyUSB0 read_flash 0x00000 0x200000 tuya.bin
```

Récupération du firmware

Une fois le firmware récupéré, on peut regarder ce qu'il contient :

xterm

```
$ strings tuya.bin | less
http://a.gw.tuyaeu.com/gw.json
http://a.gw.tuyaus.com/gw.json
...
mq.gw.airtakeapp.com
mq.gw1.airtakeapp.com
...
mqtt_client.c
...
ESP.ty_ws_mod.dev_ap_ssid_key={"ap_ssid":"SmartLife","ap_pwd":null}
ESP.ty_ws_mod.gw_active_key={"token":null,"key":null,"lo
cal_key":null,"http_url":null,"mq_url":null,"mq_url_bak":null,"timeZone":null,"re
gion":null,"reg_key":null,"wxappid":null,"uid_acl":null}
ESP.ty_ws_mod.dev_if_rec_key={"id":"04200063b4e62d00468a","sw_ver":"1.0.0","sche
ma_id":null,"etag":null,"product_key":"n8iVBAPLFKAAAszH","ability":0,"bind":false,"sync":false}
ESP.ty_ws_mod.wf_nw_rec_key={"ssid":null,"passwd":null,"wk_mode":0,"mode":0,"type":0,"source":0,
"path":0,"time":0,"random":0}
ESP.ty_ws_mod.gw_sw_ver_key={"sw_ver":"1.0.0","bs_ver":"5.06","pt_ver":"2.1"}
:ESP.device_mod.dp_data_key={"relay_switch":false,"switch":true,"work_mode":1,"bright":180,"tem
per":255,"colour_data":"1900000000ff19","scene_data":"00ff0000000000","rouguang_scene_da
ta":"ffff500100ff00","binfeng_scene_data":"ffff8003ff000000ff000000ff0000000000000000","xuan
cai_scene_data":"ffff5001ff0000","banlan_scene_da
ta":"ffff0505ff000000ff00ffff00ff00ff0000ff000000"}
ESP.device_mod.fsw_cnt_key={"fsw_cnt_key":0}
ESP.device_mod.appt_posix_key={"appt_posix":0}
ESP.device_mod.power_stat_key={"power":0}
{"mac":"68a","prod_idx":"04200063","auz_key":"TJJ7AGs644QGICVFovUcpeVeGz0JTbR1","prod_test":false}
```

l'URL de la requête

un FQDN

une bibliothèque utilisée

le SSID et le mot de passe non encore renseigné

une clé produit

une clé d'API

xterm

```
$ dig +short mq.gw.airtakeapp.com
120.55.106.107
$ curl http://a.gw.tuyaus.com/gw.json
{"t":1537555117,"e":false,"success":false,"errorCode":"API_EMPTY","errorMsg":"API"}
```

Utilisation du protocole uPnP

- adresse multicast : `239.255.255.250` ;
- port : `1900`

Alexa détecte mes appareils

xterm

```
(b'M-SEARCH * HTTP/1.1\r\nHOST: 239.255.255.250:1900\r\nMAN: "ssdp:discover"\r\nMX: 1\r\nST:
urn:dial-multiscreen-org:service:dial:1\r\n\r\n', ('192.168.0.108', 50892))
(b'M-SEARCH * HTTP/1.1\r\nHOST: 239.255.255.250:1900\r\nMAN: "ssdp:discover"\r\nMX: 15\r\nST:
urn:Belkin:device:**\r\n\r\n', ('192.168.0.118', 50000))
(b'M-SEARCH * HTTP/1.1\r\nHOST: 239.255.255.250:1900\r\nMAN: "ssdp:discover"\r\nMX: 15\r\nST:
urn:schemas-upnp-org:device:basic:1\r\n\r\n', ('192.168.0.118', 50000))
```

La recherche qui nous intéresse est `urn:schemas-upnp-org:device:basic:1`.

La réponse de l'objet connecté

En envoi par uPnP :

```
upnp_response = (b'HTTP/1.1 200 OK\r\n'
                 b'CACHE-CONTROL: max-age=3600\r\n'
                 b'LOCATION: http://%s:%s/setup.xml\r\n'
                 b'ST: urn:Belkin:device:**\r\n'
                 b'USN: uuid:%s::urn:Belkin:device:**\r\n\r\n'
                )
```

On remarque que uPnP utilise du HTTP encapsulé dans un datagramme UDP.

- ▷ l'uuid est un identifiant unique sur 14 octets⟹il identifie l'objet ;
- ▷ le `LOCATION:` indique une URL permettant de se connecter à l'objet.

Exemple de réponse pour définir les objets «*lampe*» et «*couloir*»

xterm

```
Responding to search for lampe
HTTP/1.1 200 OK
CACHE-CONTROL: max-age=86400
DATE: Sat, 29 Sep 2018 13:34:44 GMT
EXT:
LOCATION: http://192.168.0.106:51176/setup.xml
OPT: "http://schemas.upnp.org/upnp/1/0/"; ns=01
01-NLS: d67be184-acb7-4c11-8d5f-24e2faf47a1e
SERVER: Unspecified, UPnP/1.0, Unspecified
ST: urn:Belkin:device:**
USN: uuid:Socket-1_0-20f6c616d70656::urn:Belkin:device:**
X-User-Agent: redsonic


Responding to search for couloir
HTTP/1.1 200 OK
CACHE-CONTROL: max-age=86400
DATE: Sat, 29 Sep 2018 13:34:44 GMT
EXT:
LOCATION: http://192.168.0.106:51177/setup.xml
OPT: "http://schemas.upnp.org/upnp/1/0/"; ns=01
01-NLS: 0391c0d9-bd4b-4dfe-bb84-02293968f6e9
SERVER: Unspecified, UPnP/1.0, Unspecified
ST: urn:Belkin:device:**
USN: uuid:Socket-1_0-2fd636f756c6f6::urn:Belkin:device:**
X-User-Agent: redsonic
```

L'objet connecté doit hébergé un serveur HTTP et servir deux URIs :

▷ `/setup.xml` :

```
b"""<?xml version="1.0"?>
<root>
  <device>
    <deviceType>urn:MakerMusings:device:controllee:1</deviceType>
    <friendlyName>%s</friendlyName>
    <manufacturer>Belkin International Inc.</manufacturer>
    <modelName>Emulated Socket</modelName>
    <modelNumber>3.1415</modelNumber>
    <UDN>uuid:%s</UDN>
  </device>
</root>
"""
```

*Où «*friendlyName*» est le nom utilisé pour contrôler l'objet à la voix*.

▷ `/upnp/control/basicevent1` :

```
b'<?xml version="1.0" encoding="utf-8"?>
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/" s:encoding
Style="http://schemas.xmlsoap.org/soap/encoding/">
<s:Body><u:SetBinaryState xmlns:u="urn:Belkin:service:basicevent:1">
<BinaryState>1</BinaryState></u:SetBinaryState>
</s:Body></s:Envelope>'
```

- requête POST dont le contenu est au format SOAP ;
- le `<BinaryState>1</BinaryState>` indique l'allumage de l'objet connecté.