CPU pour l'embarqué

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.

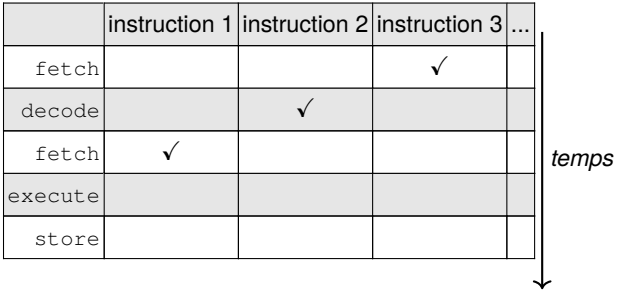
▷ **Temps réel** : contrôle de moteurs, robotique : latence basse et sûreté de fonctionnement élevée. Adaptés à des routeurs réseau, des lecteurs multimédias où les données doivent être disponible à un instant donné ;

▷ **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.

CPU pour l'embarqué

- «**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

Les périphériques de l'embarqué

- **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.

Les périphériques de l'embarqué

- **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».

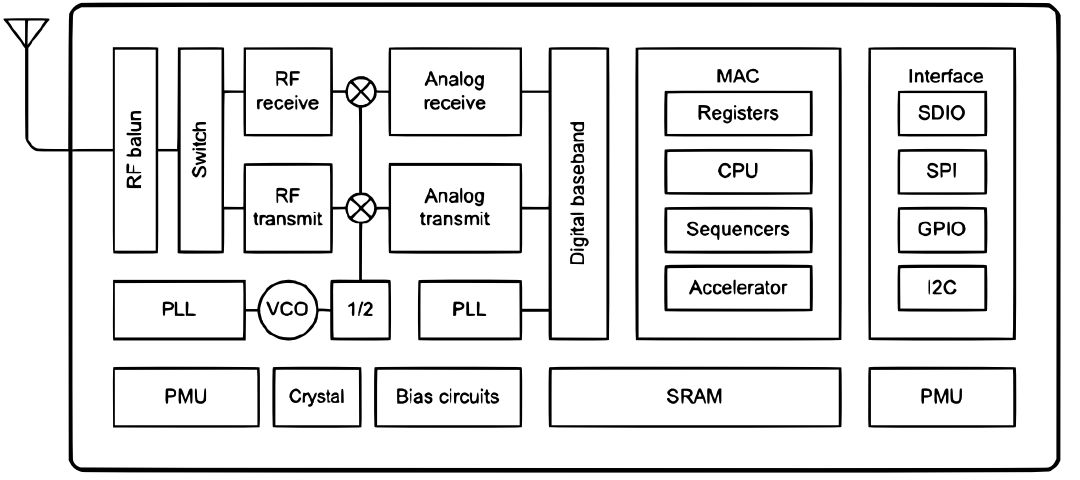
- **Timers** : compteur incrémentés ou décrémentés en fonction du temps gérés de manière indépendante du CPU
  - «*watchdog timer*» : un compteur qui doit être réinitialisé, «*kicked*», de manière logicielle avant qu'il n'atteigne zéro $\Longrightarrow$ s'il atteint zéro, le CPU subit un reset : l'idée 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 ATECC508 : 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.

ESP8226



- 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*» ;

M2M communication : «Machine To Machine»

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

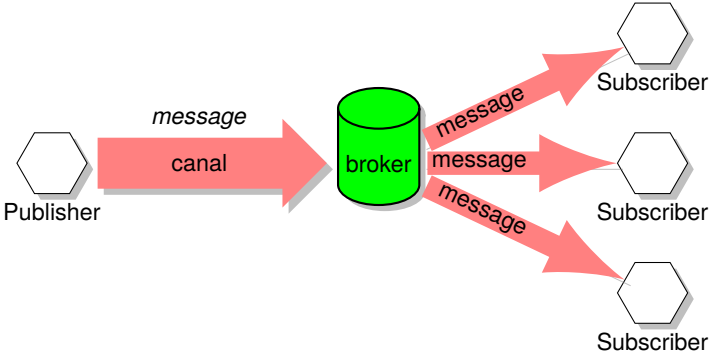
Protocol Name	Transport Protocol	Messaging Model	Security	Best-Use Cases	Architecture
AMPQ	TCP	Publish/Subscribe	High-Optional	Enterprise integration	P2P
CoAP	UDP	Request/Response	Medium-Optional	Utility field	Tree
DDS	UDP	Publish/Subscribe Request/Response	High-Optional	Military	Bus
MQTT	TCP	Publish/Subscribe 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

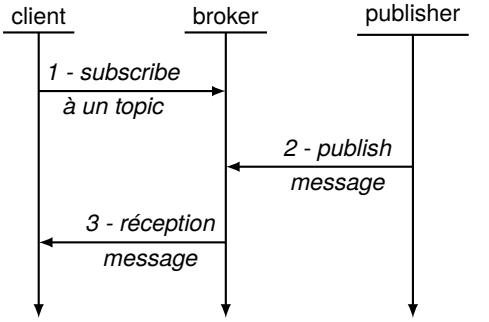
MQTT, «*Message Queue Telemetry Transport*»

- 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**

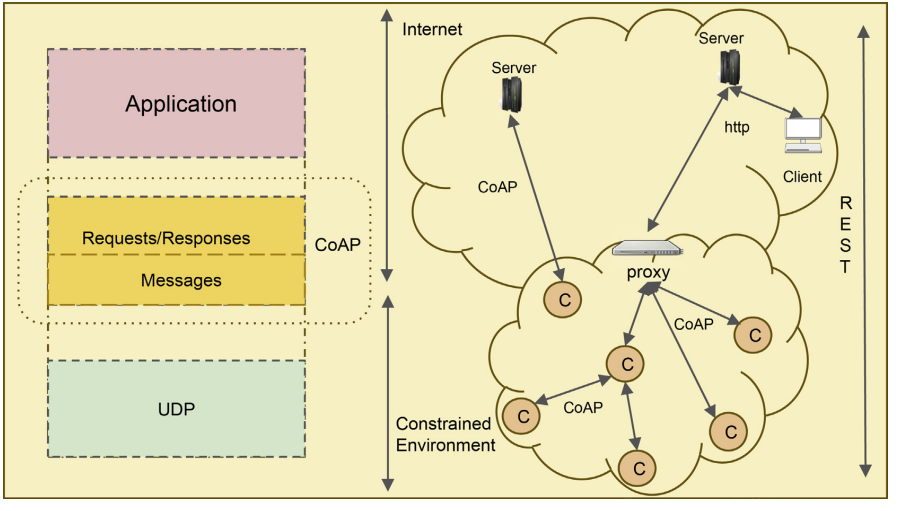
MQTT, «*Message Queue Telemetry Transport*»

- 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 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

CoAP : Constrained Application Protocol



- 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.

IoT : 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 :

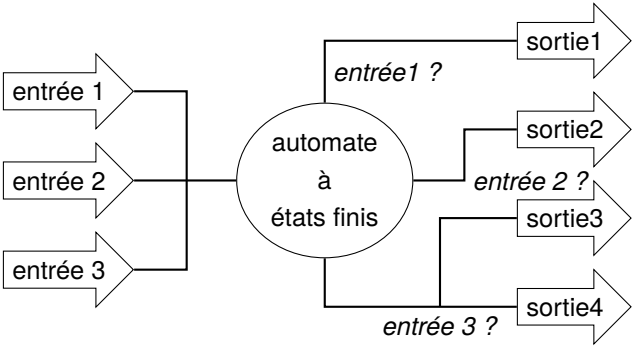
- ▷ *Si cette boucle est intégrée dans* `TravailB`*, alors le résultat peut être catastrophique pour les fonctions* `TravailA` *et* `TravailC`*: elles ne s'exécuteront que lorsque le changement d'état interviendra !*
- ▷ *Et si le changement d'état dépend d'une opération réalisée par* `TravailA` *ou* `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.*

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.*

IoT : la programmation

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

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 `driveCoroutineGood(flasher)`

```
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
7     bool, time = coroutine.resume(co)
8     tmr.delay(time * 1000)
9   end
10 end
```

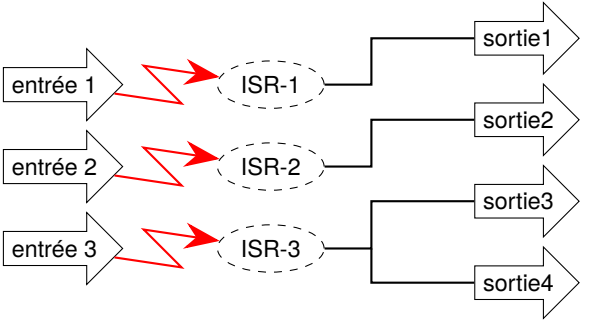
```
1 function driveCoroutineGood(proc)
2   co = coroutine.create(proc)
3   delay = 1
4   function resumeAfterDelay()
5     -- TODO: handle bool
6     bool, delay = coroutine.resume(co)
7     tmr.alarm(0, delay, 0, resumeAfterDelay)
8   end
9
10  resumeAfterDelay()
11 end
```

L'utilisation d'une alarme, `tmr.alarm` *permet de donner 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 :

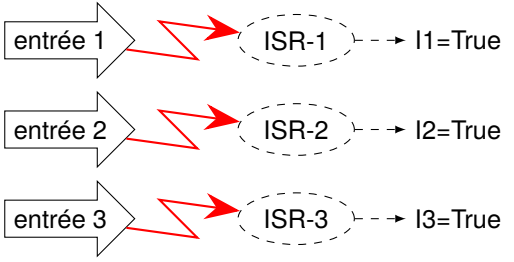
- ▷ une broche d'E/S est associée à un numéro : «*interrupt number*» ;
- ▷ un tableau, le «*interrupt vector*», est stocké à un emplacement précis de la mémoire :
  - ◇ chaque numéro est associé à l'adresse d'une fonction appelée lors du déclenchement de l'interruption associée ;
  - ◇ chacune de ces 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.

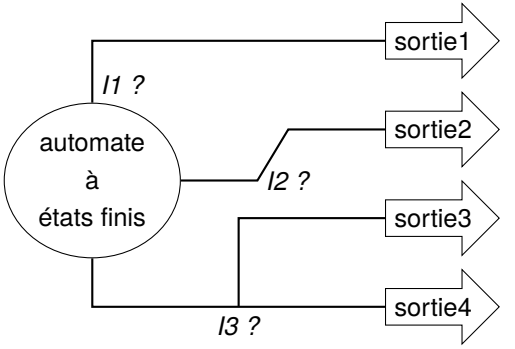


Ici, on a un traitement purement piloté par les interruptions :

- *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.*
- *souvent, 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****.*

Interruptions et automate à états finis





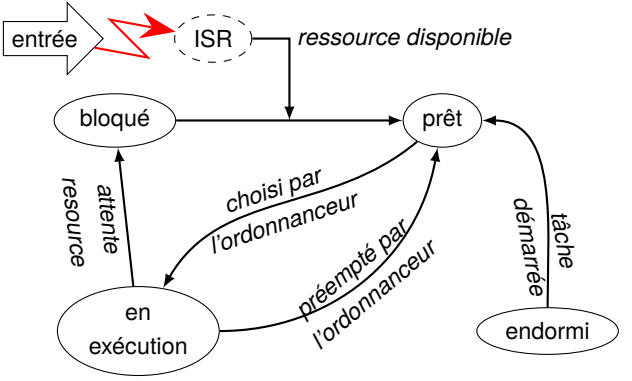
- *le traitement de l'interruption est rapide :*
  - ◇ *juste positionner un drapeau, «*flag*», à vrai ;*
  - ◇ *faible latence pour le traitement des autres interruptions.*
- *c'est l'automate qui gère les priorités s'il y en a besoin.*

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 ⟹ 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).

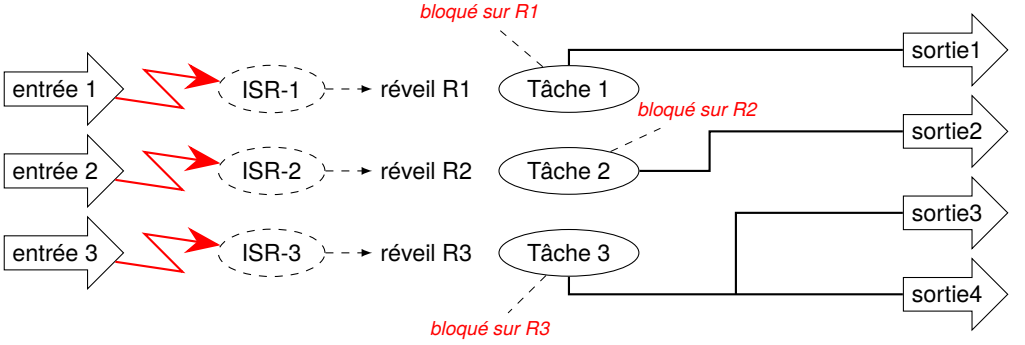
Noyau temps réel



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***».

Définitions et concepts

atomic	assure l'atomicité d'une variable susceptible d'être modifiée par différentes tâches
section critique	une section de code qui ne doit être exécutée que par une tâche à la fois
sémaphore	permet de gérer l'accès à n instances d'une même ressource : une tâche est bloquée si elle demande la sémaphore et que celle-ci tombe à zéro après decrémentation. Quand une tâche libère une sémaphore, celle-ci est incrémentée et réveille une tâche en attente de cette-ci.
loquet	implémenté par un mutex
mutex	similaire à une sémaphore mais avec un compteur limité à un
queue	mécanisme d'échange entre tâches : «*message passing*»
ré-entrant	une fonction qui peut être appelée de manière récursive : elle n'utilise pas de données statiques mais uniquement la pile
thread-safe	une fonction qui peut être utilisée par différentes threads en concurrence et dont le code est protégé par des loquets et des sections critiques