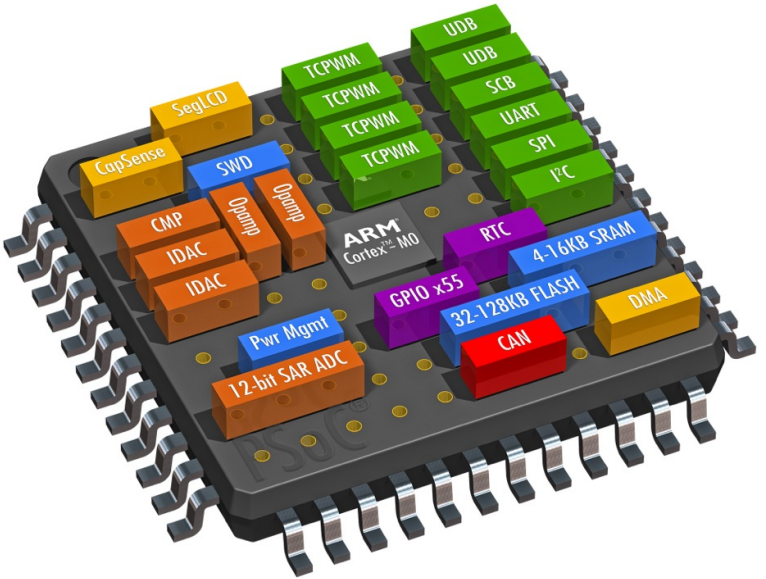
Faculté des Sciences & Techniques

Université de Limoges

Master 1ère année

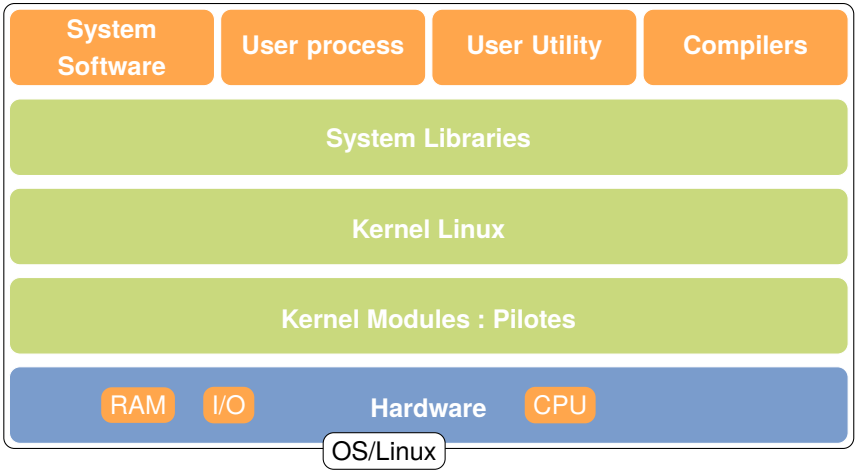


Systèmes Embarqués — P-F. Bonnefoi

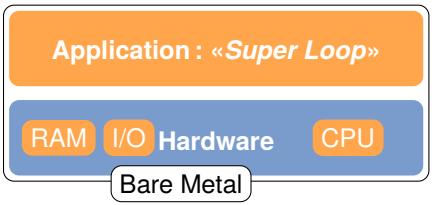
Version du 11 décembre 2023

OS *vs* «*Bare Metal*» *vs* RTOS

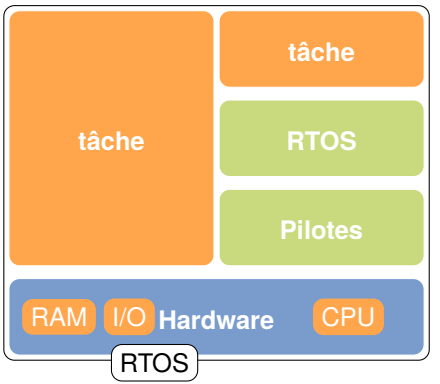
Un système basé Linux

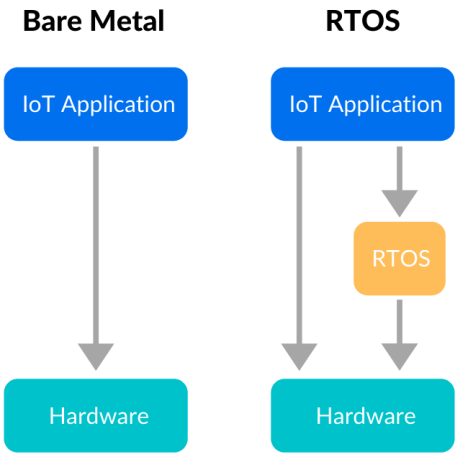


Une seule application qui fait tout



Différentes tâches qui peuvent ou non profiter du RTOS

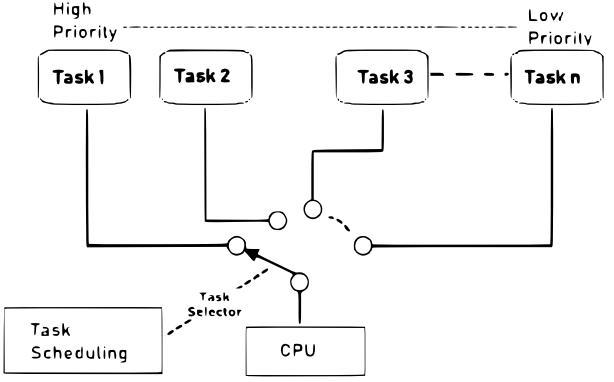




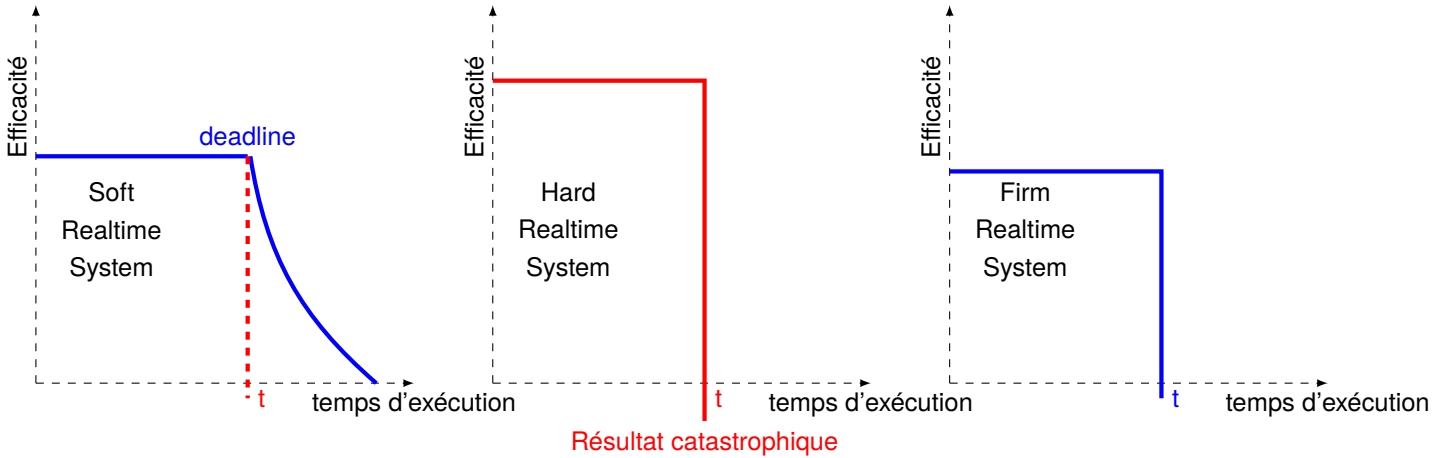
- **Bare Metal** :
  - appelé «*Super Loop*» : on a une **seule boucle** qui contient l'ensemble du travail **excepté le traitement des interruptions** ;
  - le **code de l'application**, le «*firmware*», est écrit en utilisant des périphériques contrôlés par des **registres mappés en mémoire** :
    - une **adresse accessible** par le processeur **ne va pas dans la mémoire** mais sert de **port de communication** avec le périphérique ;
    - il **écrit** à cette adresse : il transmets au périphérique (ordres/données) ;
    - il **lit** à cette adresse : il reçoit depuis le périphérique (état/données).

  $\Longrightarrow$ le logiciel peut **s'exécuter directement**, «*bare metal*», sur le micontrôleur et **piloter directement** les périphériques sans couches d'abstraction comme les pilotes, «*device drivers*» ou un OS, «*Operating System*».

- **RTOS** :
  - chaque tâche est **ordonnancée**, «*scheduled*», suivant une **période spécifique** ;
  - des tâches **sporadiques** ou **non périodiques** peuvent être **facilement ordonnancées** ;
  - l'ordonnancement se fait suivant les **priorités des tâches** ;
  - le **noyau de l'OS** et les **pilotes de périphériques** fournissent une **interface** entre le code de l'application et le hardware du micro contrôleur.



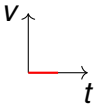
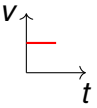
- **Hard Real Time** :
  - la «*deadline*» d'une tâche est géré de manière très stricte :
    - une tâche doit démarrer à un temps spécifié et ordonnancée de manière à finir après une durée spécifiée.
    - cela concerne des systèmes pour des soins médicaux critiques, de l'avionique, *etc*
- **Firm Real Time** :
  - ce type de RTOS doit également **suivre des deadlines**, mais le fait de **râter une deadline** ne doit pas avoir un gros impact mais seulement **un effet indésirable** comme la réduction de la qualité du produit ;
  - cela concerne les applications multimédia.
- **Soft Real Time** :
  - ce type de RTOS accepte des délais de la part de l'OS : si une deadline a été spécifiée pour une tâche spécifique, un délai minimal est acceptable.
  - cela concerne les systèmes de transaction en ligne.



C’est quoi un *bit* au fait ?

Qu'est-ce qu'un bit, «*binary digit*» ?

Un **bit** représente un système à **deux états** possibles :

- «*allumé*», ou «*éteint*», ;
- «*allumé*», (I) ou «*éteint*», (O), d'où le symbole présent sur les interrupteurs poussoir : ou
- «*Vrai*» ou «*Faux*» ;
- un **voltage** «*bas*»  ou un voltage «*haut*»  (où «*v*» est le voltage et «*t*» le temps) ;
- une **magnétisation** de sens nord-sud N|S ou de sens sud-nord S|N sur un support magnétique ;
- la **valeur** «*1*» ou la valeur «*0*».

Qu'est-ce qu'un bit de mémoire dans un ordinateur ?

«*Un bit est juste un emplacement de stockage d'électricité :*

- ▷ *s'il n'y a pas de charge électrique alors le bit est 0 ;*
- ▷ *s'il y a une charge électrique alors le bit est 1*

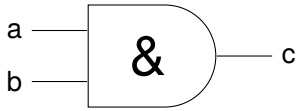
La seule chose que l'ordinateur peut mémoriser est si le bit est à 1 ou 0.»

Chaque bit de mémoire correspond à une case dans laquelle on peut stocker un bit de données, soit la valeur 1, soit la valeur 0.

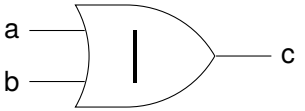
Et un processeur ça marche comment ?

C’est fait comment physiquement ?

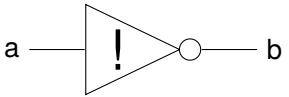
Le «*et*» : $c = a \,\&\, b$ ou $c = a \wedge b$



Le «*ou*» :$c = a|b$ ou $c = a \vee b$

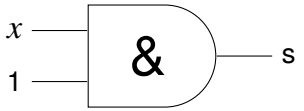


Le «*non*» : $b = !a$ ou $b = \neg a$

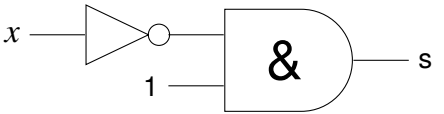


Créer des opérations

L'opération $x = 1$



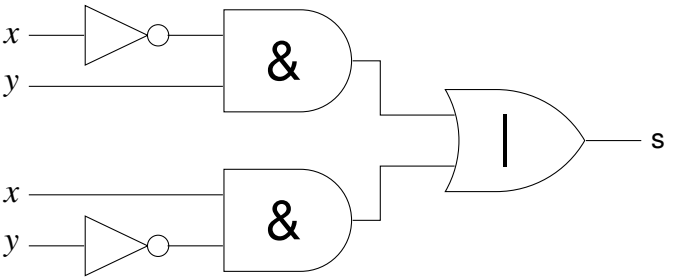
L'opération $x = 0$



$\Longrightarrow$ **L'opérateur «*xor*»**

Table de vérité du *xor*

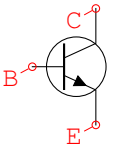
a	b	$\oplus$
0	0	0
0	1	1
1	0	1
1	1	0



On constate que le *xor* est vrai si $a \neq b$

Le transistor

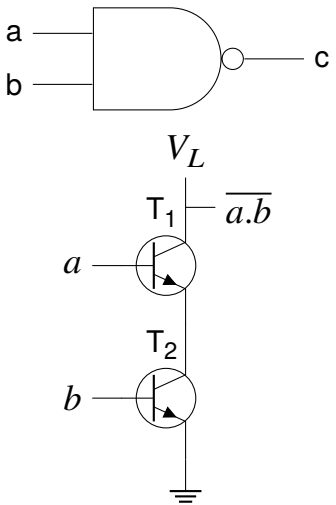
Le **transistor** agit comme un **interrupteur** : le courant peut circuler du «*collecteur*» vers «*l'émetteur*» uniquement si une tension est présente sur la «*base*» :



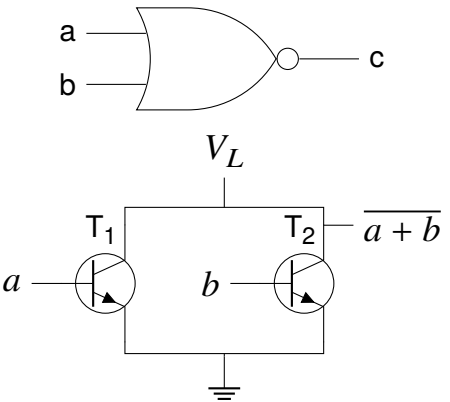
Simuler les portes logiques ?

Il est «*plus simple*» de construire des portes logiques **négatives** :

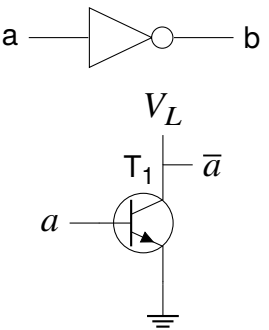
Le «*non-et*» ou NAND :



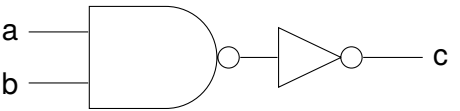
Le «*non-ou*» ou NOR :



Le «*non*» :



Et si on voulait une porte logique «*positive*», comme un «*OU*» ?

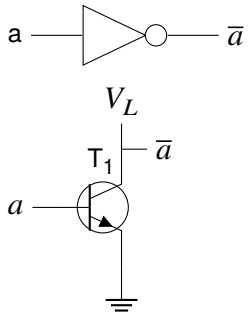


Et en vrai ?

Passer d'un **valeur logique** à un autre revient à **changer le voltage** :

- ▷ $0v$ pour le «*0*» logique $\Longrightarrow$ ce qui est relié directement au «*ground*» ;
- ▷ V_L pour le «*1*» logique $\Longrightarrow$ ce qui est relié directement à V_L ;

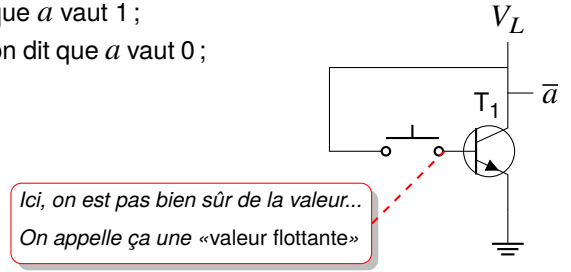
Exemple sur le «*non*» :



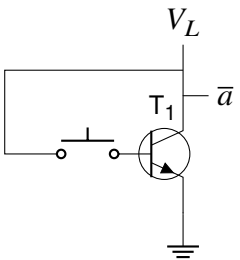
Si l'entrée a est connectée à

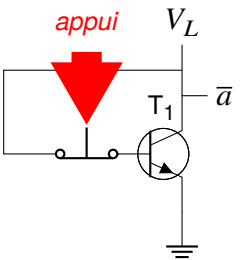
- ▷ V_L alors on dit que a vaut 1 ;
- ▷ «*ground*» alors on dit que a vaut 0 ;

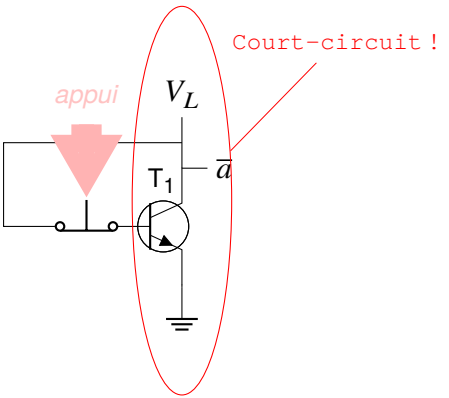
Ce qui donne :



Et électriquement, ça marche ?







Le **courant électrique** se comporte comme un liquide dont le **flot** circule du «*plus*» vers le «*moins*» :

- le **voltage**, exprimé en volts, qui exprime la «*pression*» du flot ;
- la **résistance**, exprimée en ohms, qui mesure la résistance opposée à ce flot ;
  *On notera également qu'une **chute de voltage** se produit à la sortie d'une résistance comme pour un liquide où une haute pression en entrée d'un obstacle donne une plus faible pression en sortie*
- l'**intensité**, exprimé en ampères, qui indique la quantité de liquide qui circule.
  En réalité, le nombre de charges électriques circulant dans le flot (électrons).

En général, c'est l'intensité du courant, son ampérage, qui entraîne des problèmes dans un circuit.

Loi d'Ohm $U = R * I$, ou «*volts et résistance crée l'ampérage*»

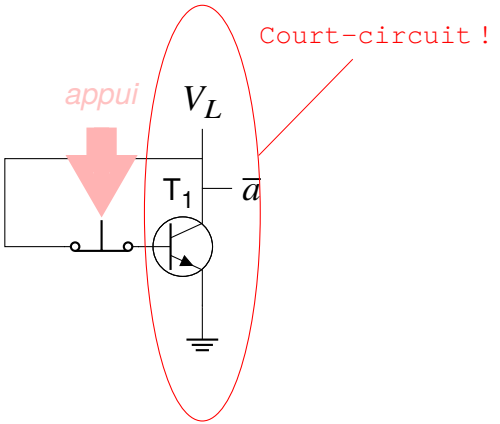
$\boxed{\frac{V}{\Omega} = A}$ ce qui se traduit pour un voltage constant par :
$\Longrightarrow$ l'ampérage ↗ quand la résistance ↘
$\Longrightarrow$ l'ampérage ↘ quand la résistance ↗

Ce qui permet de distinger **3 situations de panne** dans un circuit :

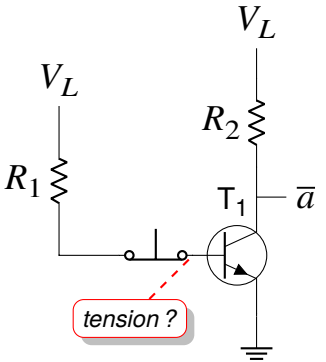
- ▷ le **circuit ouvert** où il n'y a pas de circulation $\Longrightarrow$ la **résistance** est infinie et le flot est nul ;
- ▷ le **court-circuit** où le flot va directement vers le «*ground*» ce qui entraîne trop de flot $\Longrightarrow$ la **résistance** est très proche de zéro et l'ampérage tend vers l'infini $\Longrightarrow$ les composants brûlent !
  Ils libèrent la fumée magique qui les faisait fonctionner...
- ▷ **pas assez de flot de courant** pour que le circuit fonctionne correctement $\Longrightarrow$ la **résistance** est trop élevée.

On remarque que chaque panne est liée à un changement de résistance...

Et finalement ?



Il faut ajouter des résistances pour limiter l'intensité du courant, c-à-d son ampérage.



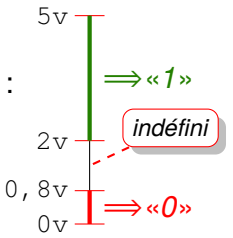
On évite deux court-circuits possibles avec R_1 et R_2.

Par contre, on ne connait pas la tension à l'entrée de la «base» du transistor...

Comment distinguer un «*0*» et un «*1*» ?

Pour des circuits électroniques «*standards*» (TTL) :

- ▷ de $5v$ à $2v \Longrightarrow$ «*1*» ;
- ▷ de $0,8v$ à $0v \Longrightarrow$ «*0*» ;
- ▷ de $0,9v$ à $1,9v \Longrightarrow$ «*indéfini*» ou «*flottant*» ;



Autre usage de ces résistances

Elles garantissent une tension :

- ▷ **Pull up** resistor : garantie une tension proche de V_L, c-à-d un «*1*» logique, *ici, R_1 et R_2* ;
- ▷ **Pull down** resistor : garantie une tension proche de 0, c-à-d un «*0*» logique, *ici, il n'y en a pas !*

Et finalement ?

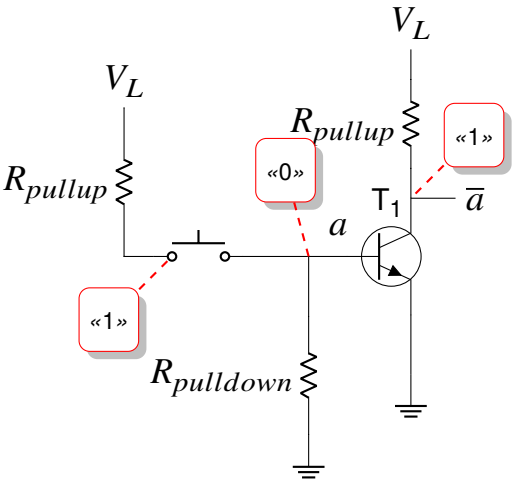
On rajoute des résistances de «*pull up*» pour :

- ▷ **forcer une tension** interprétable comme un «*1*» logique ;
- ▷ **éviter un court circuit** en cas d'utilisation d'interrupteur pour ouvrir/fermet le circuit ;

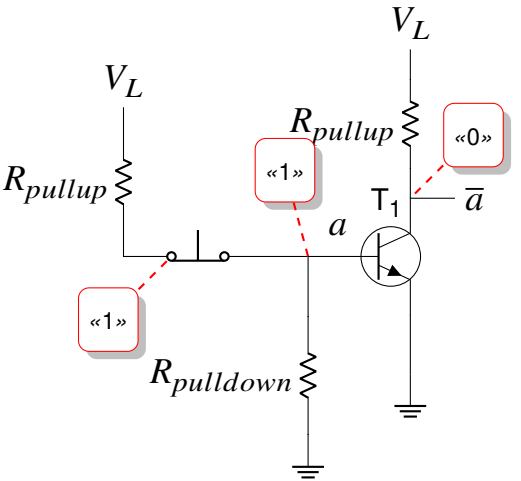
On rajoute des résistances de «*pull down*» pour :

- ▷ **forcer une tension** interprétable comme un «*0*» logique ;
- ▷ **éviter un court circuit** en cas d'utilisation d'interrupteur pour ouvrir/fermet le circuit ;

D'où le circuit final :

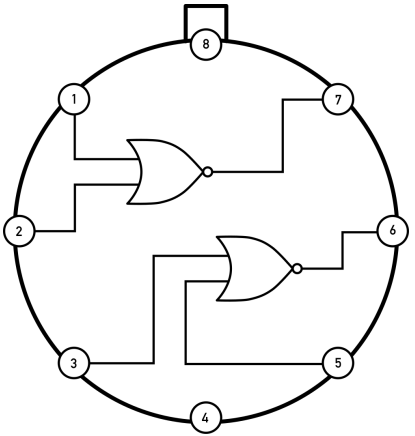


Si $a = 0$ alors $\overline{a} = 1$



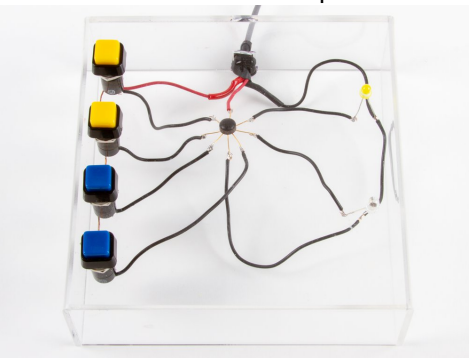
Si $a = 1$ alors $\overline{a} = 0$

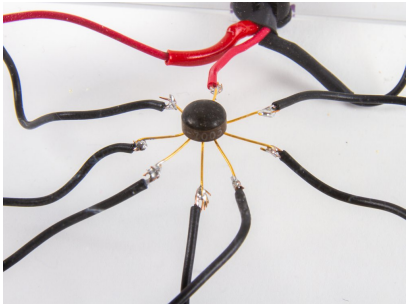
Et en réalité ? Exemple du composant µL914 de la société Fairchild



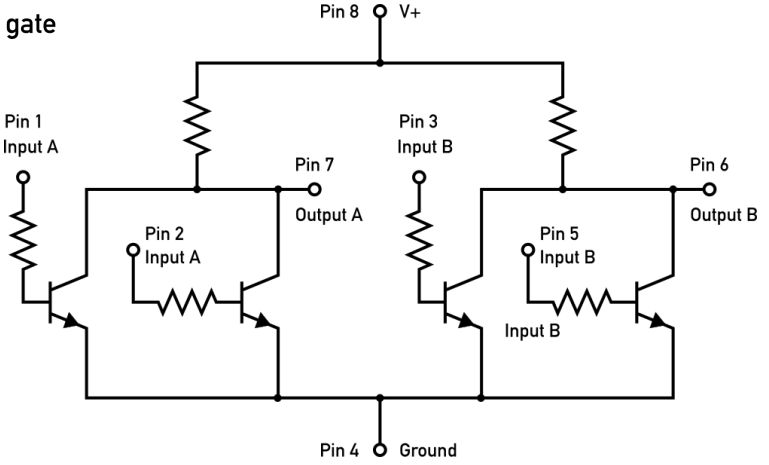
µL914
Dual 2-input NOR gate

Le composant est au centre,
connecté à des boutons poussoirs.



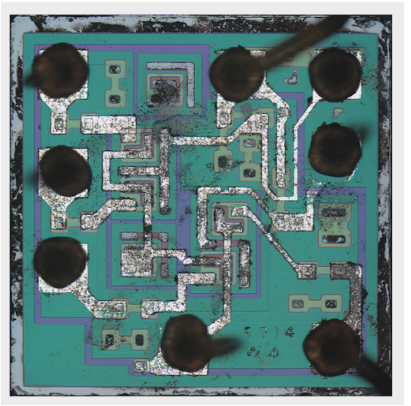


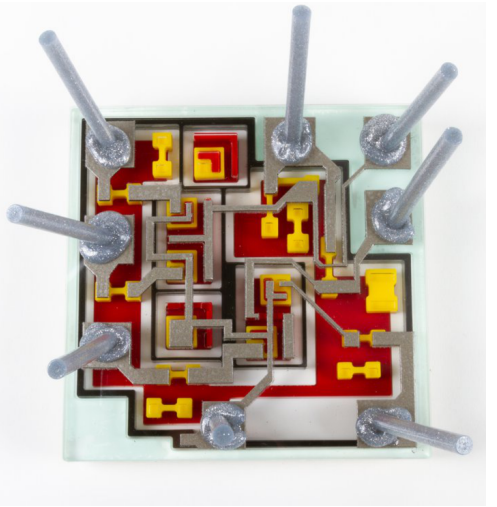
Agrandissement du composant.

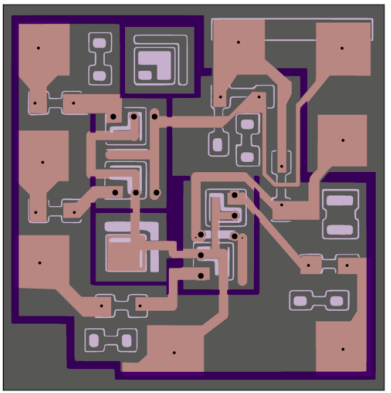


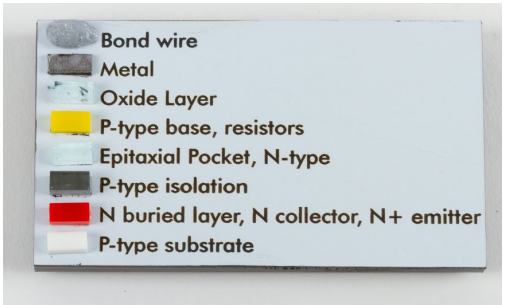
Et à l'intérieur ?

Le composant a été «*décapé*» : sa coque de protection a été enlevée par abrasion et utilisation d'acides :







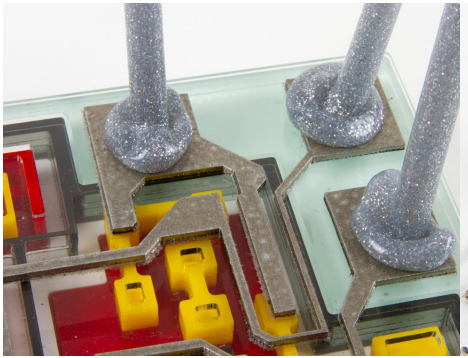


Le composant est constitué de différentes couches de matériaux différents, superposées les unes sur les autres.

On obtient chaque couche par dépôt de substrat ou par gravure (creusement d'une couche).

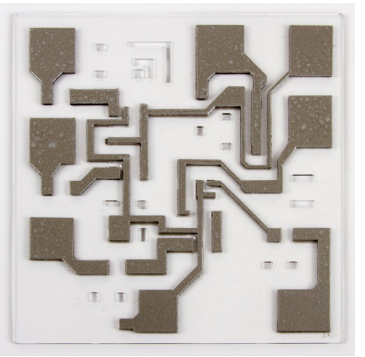
Et à l'intérieur ?

Fils de connexion vers l'extérieur :

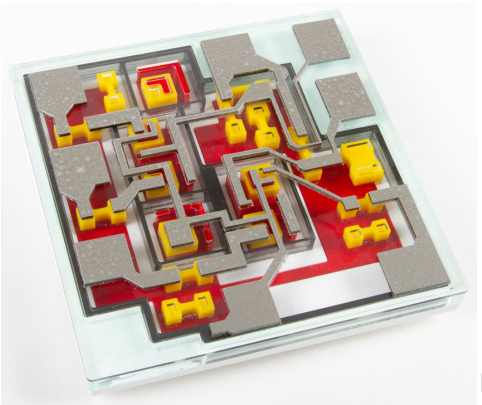


Connexion sur la couche conductrice :

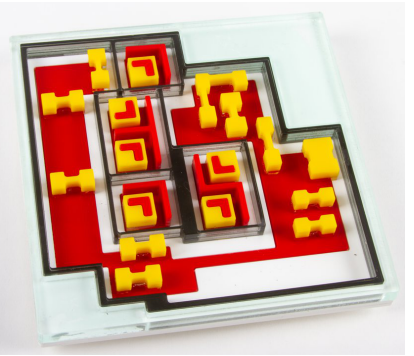
Couce conductrice :



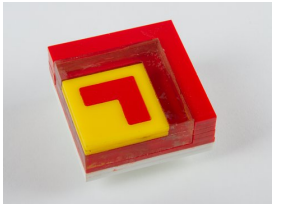
Toutes les couches :



Sans la couche conductrice :



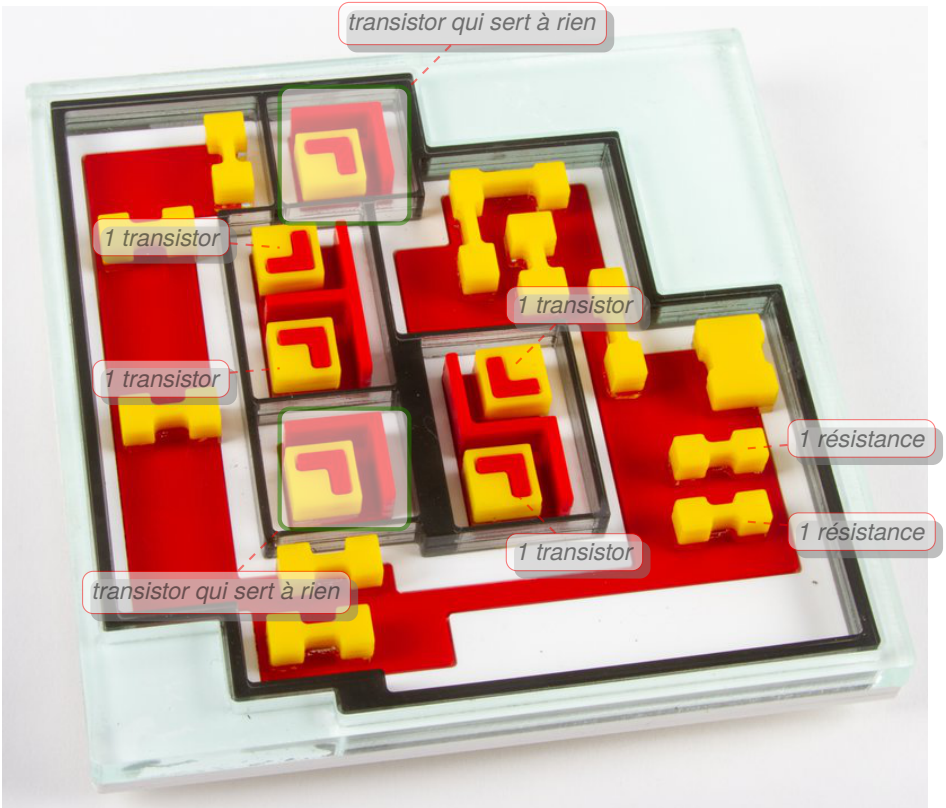
Les Transistors :



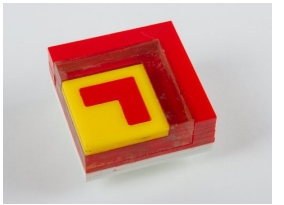
Les Résistances :

Deux transistors ne servent à rien.

Si on analyse...



Les Transistors :



Les Résistances :

On retrouve chaque transistor et résistance du circuit. Certains transistors ne servent à rien : ils ont été gravés/déposé mais ne sont pas connectés par la couche conductrice.

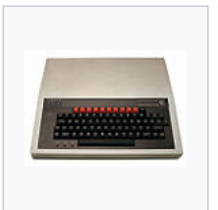
Et si on regardait
les premiers ordinateurs personnels ?

Processeur 6502

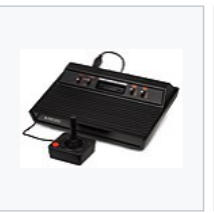
- processeur développé par Chuck Peddle pour la société MOS Technology ;
- introduit en 1975 ;
- très populaire :

Apple IIe

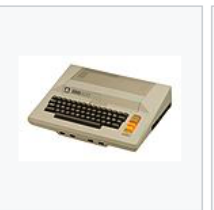
Commodore PET



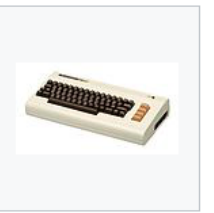
BBC Micro



Atari 2600



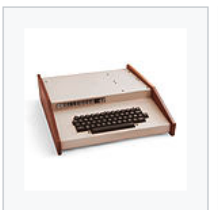
Atari 800



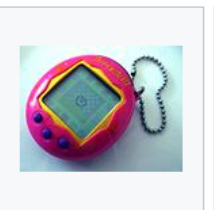
Commodore VIC-20

Commodore 64

Family Computer



Ohio Scientific Challenger 4P



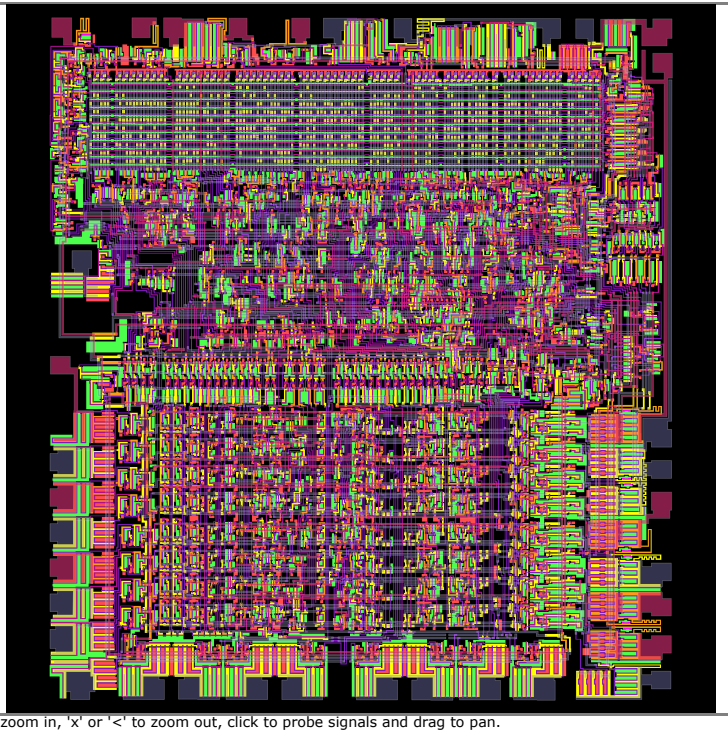
Tamagotchi digital pet[53]

Atari Lynx

- toujours en vente et utilisé dans les **systèmes embarqués** ;
- processeur 8bits, avec un bus d'adresse sur 16bits et «*little-endian*», cadencé de 1 à 2 MHz

Le processeur 6502 : représentation visuelle

FAQ Blog Links Source easy6502 assembler mass:werk disassembler



Use 'z' or '>' to zoom in, 'x' or '<' to zoom out, click to probe signals and drag to pan.

Show: ☑(diffusion) ☑(grounded diffusion) ☑(powered diffusion) ☑(polysilicon) ☑(metal) ☑(protection)

Find: | Clear Highlighting | Animate during simulation: ☑

Hide Chip Layout | Link to this location

halfcyc:372 phi0:0 AB:0015 D:69 RnW:1
PC:0015 A:12 X:07 Y:f9 SP:fb nv-BdIzc
Hz: 3.3 Exec: SEC(T0+T2)

```
0000: a9 00 20 10 00 4c 02 00 00 00 00 0
0010: e8 88 e6 0f 38 69 02 60 00 00 00 0
0020: 00 00 00 00 00 00 00 00 00 00 00 0
0030: 00 00 00 00 00 00 00 00 00 00 00 0
0040: 00 00 00 00 00 00 00 00 00 00 00 0
0050: 00 00 00 00 00 00 00 00 00 00 00 0
0060: 00 00 00 00 00 00 00 00 00 00 00 0
0070: 00 00 00 00 00 00 00 00 00 00 00 0
0080: 00 00 00 00 00 00 00 00 00 00 00 0
0090: 00 00 00 00 00 00 00 00 00 00 00 0
00a0: 00 00 00 00 00 00 00 00 00 00 00 0
00b0: 00 00 00 00 00 00 00 00 00 00 00 0
00c0: 00 00 00 00 00 00 00 00 00 00 00 0
00d0: 00 00 00 00 00 00 00 00 00 00 00 0
00e0: 00 00 00 00 00 00 00 00 00 00 00 0
00f0: 00 00 00 00 00 00 00 00 00 00 00 0
0100: 00 00 00 00 00 00 00 00 00 00 00 0
0110: 00 00 00 00 00 00 00 00 00 00 00 0
0120: 00 00 00 00 00 00 00 00 00 00 00 0
0130: 00 00 00 00 00 00 00 00 00 00 00 0
0140: 00 00 00 00 00 00 00 00 00 00 00 0
0150: 00 00 00 00 00 00 00 00 00 00 00 0
0160: 00 00 00 00 00 00 00 00 00 00 00 0
0170: 00 00 00 00 00 00 00 00 00 00 00 0
0180: 00 00 00 00 00 00 00 00 00 00 00 0
0190: 00 00 00 00 00 00 00 00 00 00 00 0
01a0: 00 00 00 00 00 00 00 00 00 00 00 0
01b0: 00 00 00 00 00 00 00 00 00 00 00 0
01c0: 00 00 00 00 00 00 00 00 00 00 00 0
```

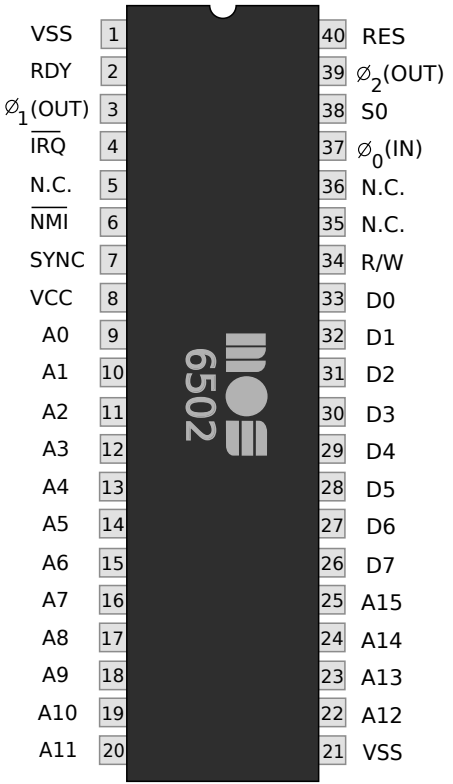
```
@AABBCCDDEEFFG
```

Trace more | Trace less | Trace these too: | Clear Log

```
cycle ab db rw Fetch pc a x y s p
```

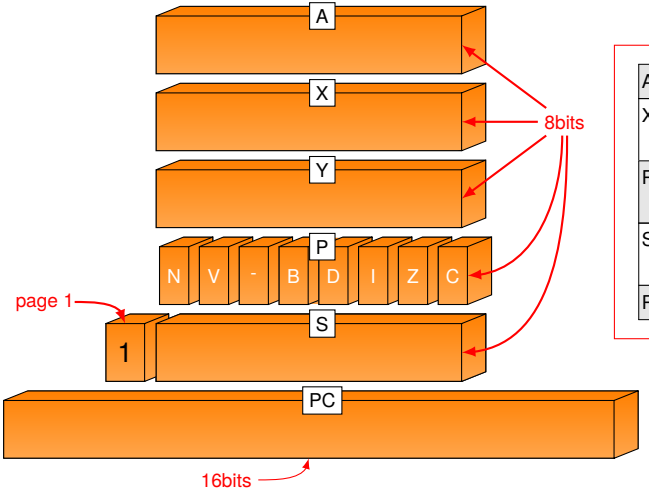
http://www.visual6502.org/JSSim/expert.html

Le 6502 : connexion avec l'extérieur



- Accès à la mémoire :
  - A_0, ..., A_{15} : 16 bits d'adresse ;
  - D_0, ..., D_7 : 8 bits de données ;
  - R/W : indique si c'est une opération de lecture ou d'écriture ;
- Interactions avec l'extérieur :
  - $Sync$: signal d'horloge : ryhtme le travail du processeur ;
  - NMI : «*Non Maskable Interruption*» : signal d'interruption ;
  - RES : «*reset*», réinitialise l'état du processeur et, si maintenue, le bloque ;

Le processeur 6502

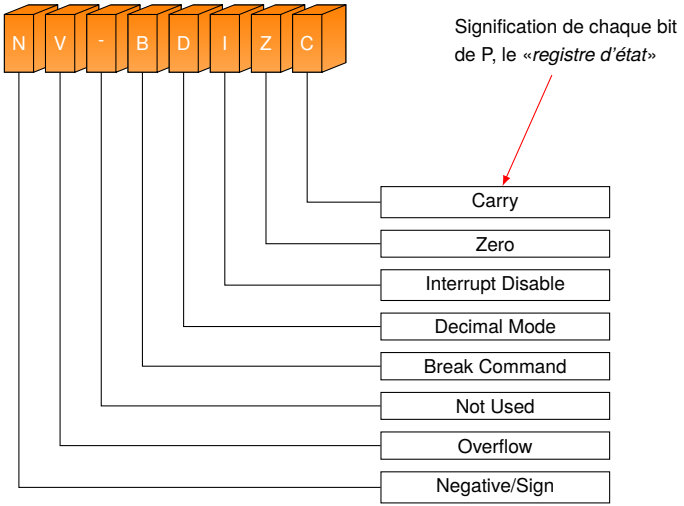


Registres

A	«*Accumulator*»	stockage depuis ou vers l'ALU
X & Y	«*Index register*»	utilisés dans certaines instructions pour calculer une adresse par décalage : `adresse+X`
P	«*Processor status register*»	chaque bit indique 1 état suite à l'exécution de l'instruction : nombre positif, nul, etc.
S	«*Stack pointer*»	contient l'adresse du dernier octet dans la pile le bit de préfixe à 1 place la pile sur la seconde page
PC	«*Program counter*»	indique l'adresse de la prochaine instruction à exécuter

Explications du registre d'état

Carry	indique un bit de retenu après opération de l'ALU (9ème bit...)
Zero	la valeur de X, Y ou A est devenue zéro
oVerflow	dépassement de capacité lors d'opération sur des nombres signés
Negative	vrai si le bit de rang 7 est à 1



mode	opérande
immédiat	la donnée
absolu	n'importe quelle adresse
page zéro	un octet correspondant au second octet d'adresse, le premier est fixé à zéro
indexé X	adresse+registre X
indexé Y	adresse+registre Y
implicite	pas d'opérande
relatif	un octet relatif en complément à deux, de -128 à 127

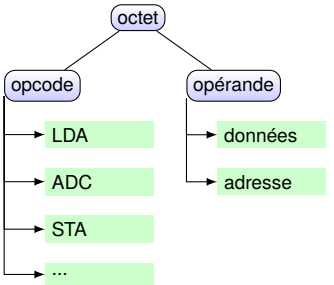
Chaque instruction est **codée sur un octet** en fonction du mode choisi.

Exemple : l'instruction ADC donne l'octet `69` *si la valeur à additionner est donnée en paramètre (mode immédiat).*

`69 01` *signifie additionner la valeur 1 dans l'accumulateur.*

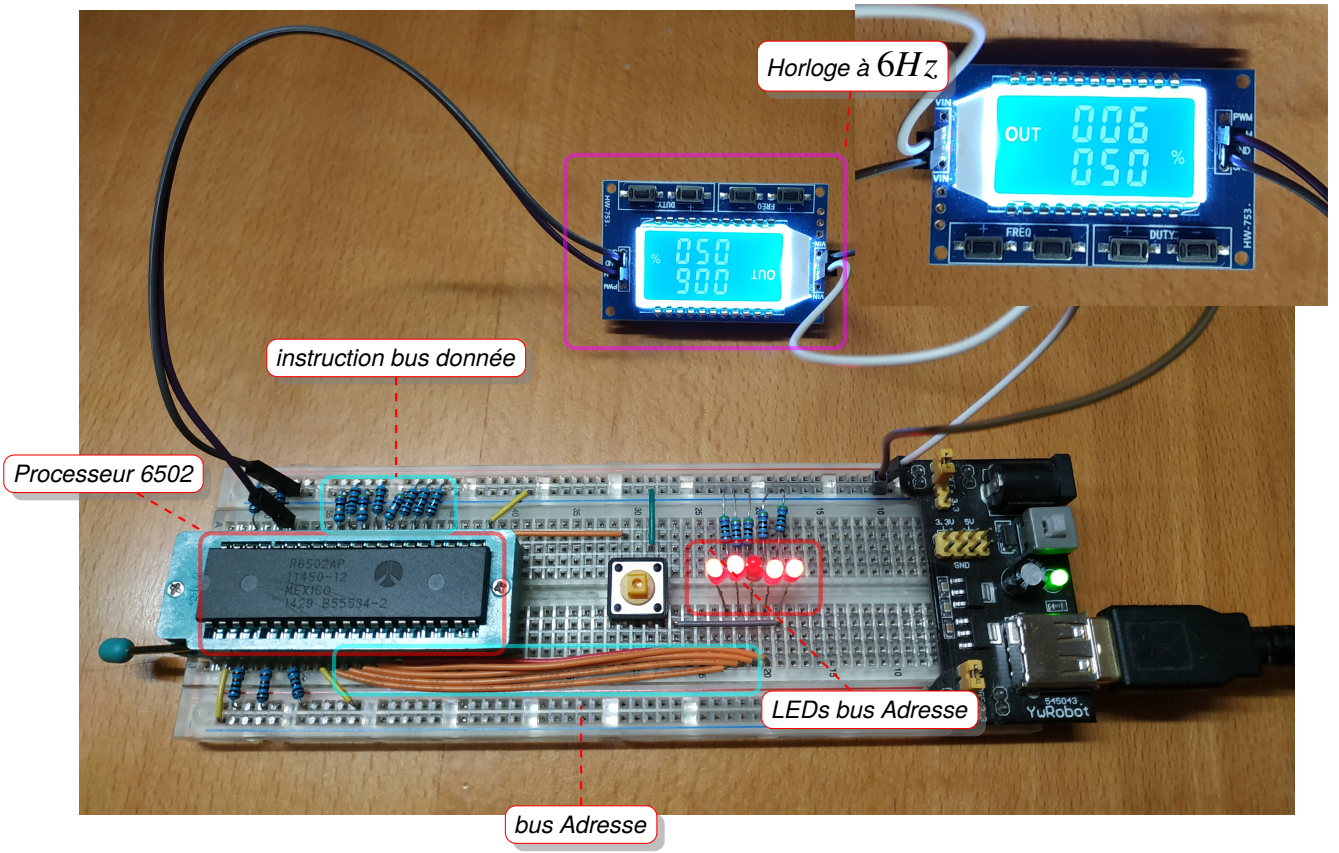
Certaines instructions ***modifient le registre d'état P*** *: Exemple pour faire un saut sur la condition que X soit égal à zéro :*
`CPX $0` *; compare la valeur du registre X avec 0 ⟹ positionne le bit Z à nul si les deux valeurs sont identiques (on fait une soustraction)*
`BEQ 0A` *; test la valeur du bit Z : 1 donne vrai et 0 donne faux*

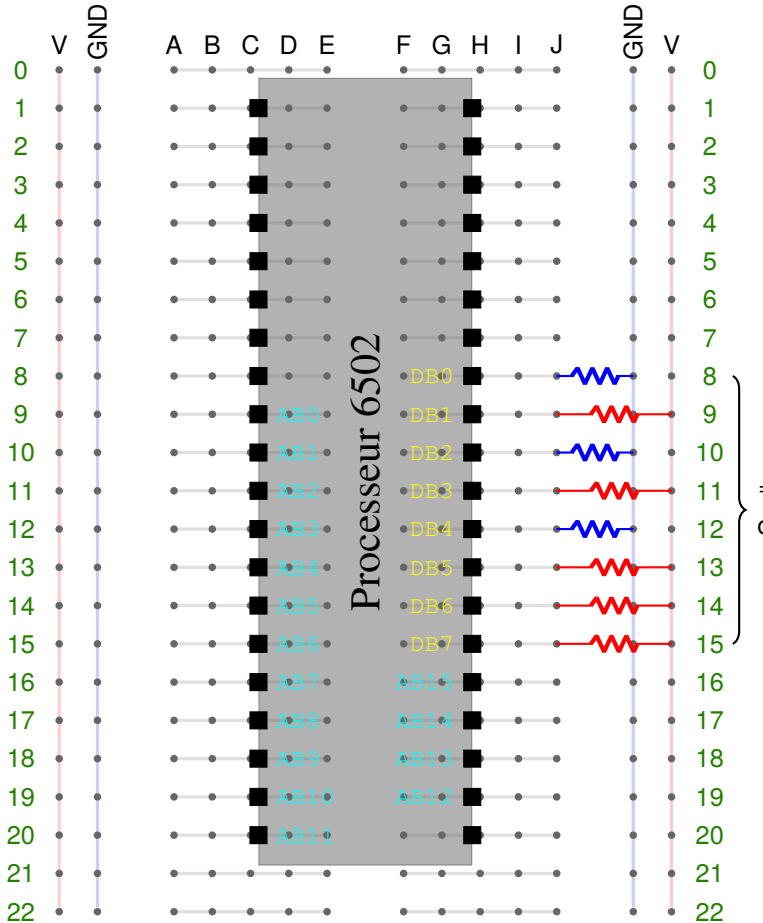
Un octet en mémoire peut être :



Ins	description	mode adressage						
		immédiat	absolu	page zéro	indexé X	indexé Y	implicite	relatif
ADC	ajoute un octet avec le bit de retenu dans l'accumulateur	69	6D	65	7D	79		
BEQ	«*Branch if EQual*», saut vers une adresse si vrai							F0
BNE	«*Branch if Not Equal*», saut vers une adresse si faux							D0
CPX	compare avec le registre X	E0	EC	E4				
INX	Incrémente la valeur dans le registre X						E8	
INY	Incrémente la valeur dans le registre Y						C8	
JMP	«*JuMP*», saut		4C					
JSR	«*Jump to SubRoutine*», saut vers un sous-programme		20					
LDA	charge un octet dans le registre A	A9	AD	A5	BD	B9		
LDX	charge un octet dans le registre X	A2	AE	A6		BE		
LDY	charge un octet dans le registre Y	A0	AC	A4	BC			
RTS	«*ReTurn from Subroutine*», retour d'un sous-programme						60	
STA	stocke l'accumulateur à une adresse donnée		8D	85	9D	99		
NOP	ne fait rien						EA	

6502 sur «*breadboard*» : un processeur, une horloge et deux bus





⟹11101010 de `DB7` à `DB0`
ce qui donne en hexa $\{1110\}_2\{1010\}_2 = \{E\}_{16}\{A\}_{16}$

`EA` en instruction 6052 ⟹ `NOP` pour «*No OPeration*»
Ce qui veut dire que le processeur ne fait rien
Il passe seulement à l'instruction suivante
⟹il incrémente le CO, «*Compteur ordinal*», ou «*instruction counter*»
⟹l'adresse est incrémentée sur le bus d'adresse `AB0` à `AB15`

Application d'un xor d'un texte avec un mot de passe

Le programme calcule $saisie_i \oplus mdp_i$ pour chaque caractère i de $saisie$ et de mdp.

```
 1 define sortie $200 ; on définie l'adresse de sortie à 0200
 2
 3 LDA saisie   ; on lit la taille de la chaîne saisie
 4 STA sortie   ; on la reporte dans la chaîne de sortie
 5 ADC #$1      ; on incrémente la valeur pour la comparaison utilisée pour arrêter la boucle
 6 STA $0      ; on la stocke dans la page zéro
 7 LDA mdp      ; on lit la taille de la chaîne mdp
 8 ADC #$1      ; on incrémente la valeur utilisée pour réinitialiser l'utilisation du mdp
 9 STA $1      ; on la stocke dans la page zéro
10
11 LDX #$1      ; on charge la valeur 1 dans le registre X
12 LDY #$1      ; on charge la valeur 1 dans le registre Y
13
14 boucle:      ; on définit une étiquette
15    LDA saisie,X   ; on charge dans l'accumulateur la valeur à l'adresse saisie+X
16    EOR mdp,Y      ; on réalise un xor entre le registre A et la valeur à l'adresse mdp+Y
17    STA sortie,X   ; on stocke le résultat à l'adresse sortie+X
18    INX            ; on incrémente la valeur contenu dans le registre X
19    CPX $0         ; on compare la valeur de la taille de la chaîne saisie
20    BEQ fin         ; si elle est identique, on a fini et on mets l'adresse fin dans le registre PC
21    INY            ; on incrémente la valeur contenue dans le registre Y
22    CPY $1         ; on compare avec la valeur de la taille de la chaîne mdp
23    BNE boucle      ; si elle n'est pas égale on recommence la boucle en sautant à l'adresse boucle
24    LDY #$1         ; sinon on réinitialise le registre Y à 1
25    JMP boucle      ; et on effectue un saut à l'adresse boucle
26 fin:         ; étiquette
27    BRK            ; instruction d'arrêt
28
29 saisie:
30    dcb 5,$68,$65,$6c,$6c,$6f ;hello
31 mdp:
32    dcb $9,$74,$6f,$70,$73,$65,$63,$72,$65,$74;topsecret
```

Utilisation du désassembleur

```
Address   Hexdump    Dissassembly
-------------------------------
$0600     ad 2e 06   LDA $062e
$0603     8d 00 02   STA $0200
$0606     69 01      ADC #$01
$0608     85 00      STA $00
$060a     ad 3c 06   LDA $063c
$060d     69 01      ADC #$01
$060f     85 01      STA $01
$0611     a2 01      LDX #$01
$0613     a0 01      LDY #$01
$0615     bd 2e 06   LDA $062e,X
$0618     59 3c 06   EOR $063c,Y
$061b     9d 00 02   STA $0200,X
$061e     e8         INX
$061f     e4 00      CPX $00
$0621     f0 0a      BEQ $062d
$0623     c8         INY
$0624     c4 01      CPY $01
$0626     d0 ed      BNE $0615
$0628     a0 01      LDY #$01
$062a     4c 15 06   JMP $0615
$062d     00         BRK
$062e     0d 68 65   ORA $6568
$0631     6c 6c 6f   JMP ($6f6c)
$0634     20 62 6f   JSR $6f62
$0637     6e 6a 6f   ROR $6f6a
$063a     75 42      ADC $42,X
$063c     09 74      ORA #$74
$063e     6f         ???
$063f     70 73      BVS $06b4
$0641     65 63      ADC $63
$0643     72         ???
$0644     65 74      ADC $74
```

```
0600: ad 2e 06 8d 00 02 69 01 85 00 ad 3c 06 69 01 85
0610: 01 a2 01 a0 01 bd 2e 06 59 3c 06 9d 00 02 e8 e4
0620: 00 f0 0a c8 c4 01 d0 ed a0 01 4c 15 06 00 0d 68
0630: 65 6c 6c 6f 20 62 6f 6e 6a 6f 75 72 09 74 6f 70
0640: 73 65 63 72 65 74
```

On note que :

`$062e`	adresse de la chaîne saisie
`$063c`	adresse de la chaîne mdp
`$062d`	adresse de l'instruction `brk`
`$062e`	le désassembleur trouve des instructions dans le contenu de la chaîne saisie $\Longrightarrow$ Interprétation automatique erronée
`$063e` `$0643`	Interprétation automatique impossible, il n'y a pas d'instruction reconnue

Et les communications avec le développeur ?

Transmission de données numériques

La transmission numérique consiste à faire transiter les informations sur le support physique de communication **sous forme de signaux numériques**.

Les informations numériques :

- ne peuvent pas circuler sous forme de 0 et de 1 directement ;
- doivent être **codées** sous forme d'un **signal** possédant deux états.

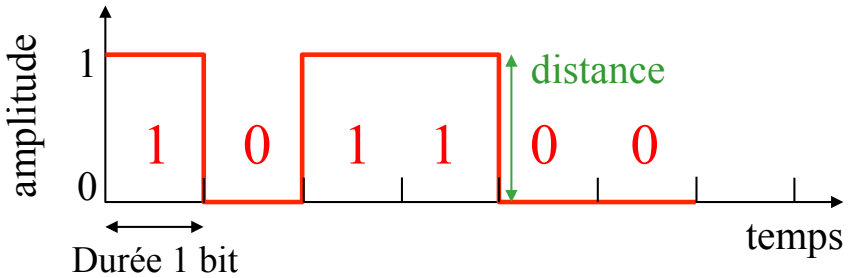
Un signal est une quantité mesurable variant au cours du temps ou dans l'espace.

Exemple :

- deux niveaux de tension par rapport à la masse ;
- la différence de tension entre deux fils ;
- la présence/absence de courant dans un fil ;
- la présence/absence de lumière ;
- *etc.*

La **transformation de l'information binaire sous forme d'un signal** à deux états est réalisée par l'interface.

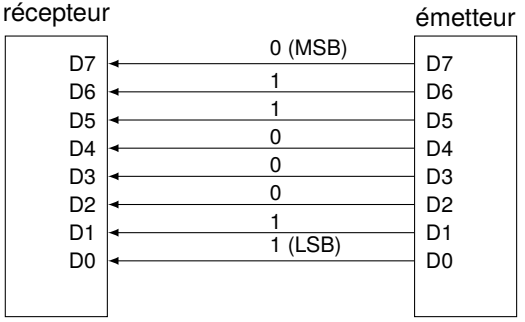
Exemple : bits codés suivant une différence de tension



L'interface réalise le «*codage en bande de base*».

On parlera de «transmission numérique» ou «transmission en bande de base», *baseband*.

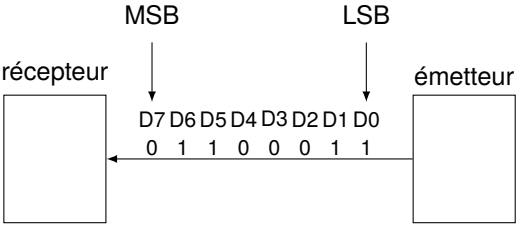
Transmission parallèle



- LSB : «*least significant bit*»
- MSB : «*most significant bit*»

Les bits sont émis **simultanément** sur autant de fils que de nombre de bits utilisé pour le codage.
Ce mode est employé pour les bus internes des ordinateurs (bus 16, 32 ou 64bits) parfois pour la communication vers des périphériques (imprimantes, bus SCSI, bus IDE…).
Exemple : on transmet un octet sur 8 fils, en envoyant en même temps chaque bit sur chaque fil.

Transmission série



Les bits sont transmis **séquentiellement** sur un seul fil.
Dans les réseaux, qu'ils soient locaux ou étendus, c'est la transmission série qui est utilisée.

C'est la **liaison série** qui est la **plus utilisée** (disque dur SATA, USB, ...)

Transmission série sur un seul fil pour une liaison synchrone

- *émetteur*, E, et *récepteur*, R, utilisent une **même base de temps** pour émettre les bits (horloge) ;
- il sont **cadencés** suivant la même horloge ;
- à chaque «top d'horloge», un bit est envoyé et R sait donc «quand» récupérer ce bit.

Le récepteur reçoit de façon continue les informations au rythme auquel l'émetteur les envoie.

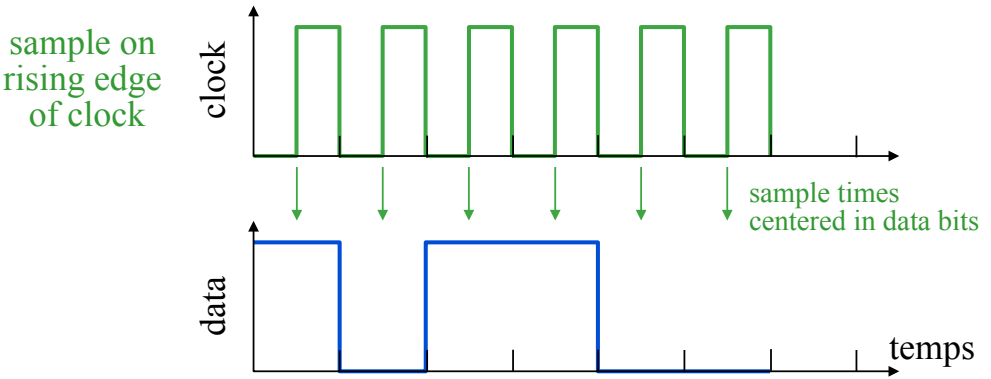
Inconvénient :

▷ la reconnaissance des informations au niveau du récepteur : il peut exister des différences entre les horloges de l'émetteur et du récepteur.

C'est pourquoi chaque envoi de bit doit se faire **sur une durée assez longue** pour que le récepteur la distingue.

Ainsi, la vitesse de transmission **ne peut pas être très élevée** *dans une liaison synchrone sans recourir à du matériel coûteux.*

Transmission série sur deux fils pour une liaison synchrone



Transmission série sur un seul fil pour une liaison asynchrone

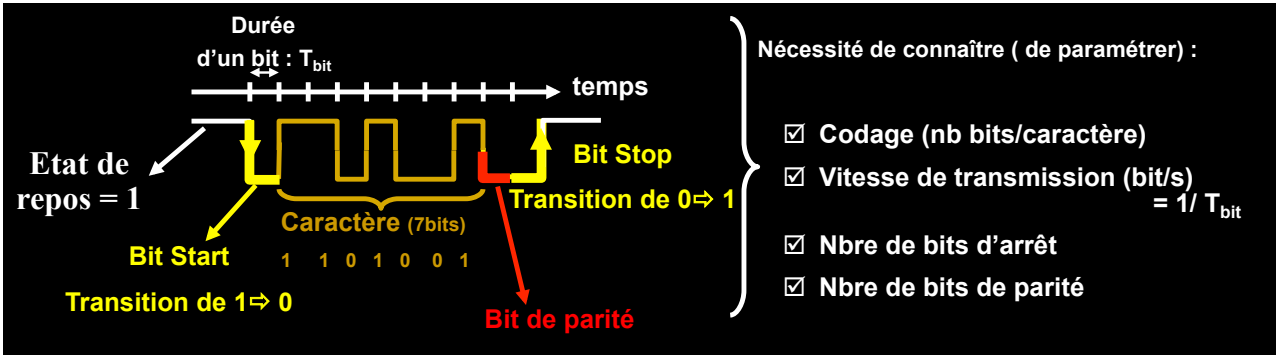
L'émetteur et le récepteur ne sont pas *synchronisés*.
Le récepteur doit détecter des **transitions** au sein des données reçues.

Problème Si un seul bit est transmis pendant une longue période de silence… le récepteur ne pourrait savoir s'il s'agit de `00010000`, ou `10000000` ou encore `00000100`…

Solution **Chaque caractère** est :

- **précédé d'une information** indiquant le début de la transmission du caractère (l'information de début d'émission est appelée bit START) ;
- **terminé par l'envoi** d'une information de fin de transmission (appelée bit STOP, il peut éventuellement y avoir plusieurs bits STOP).

Exemple



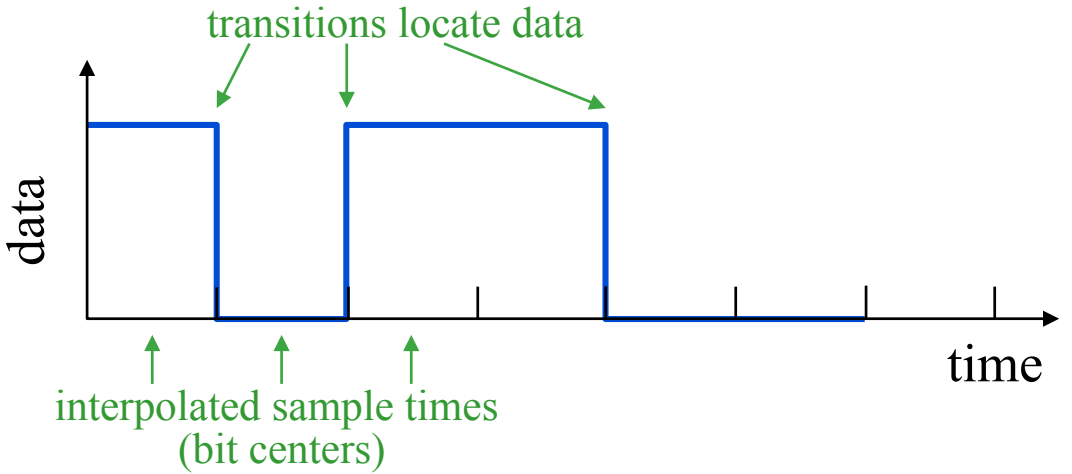
Ici, le codage consiste à passer d'une tension à l'autre seulement si on veut transmettre un bit de valeur différente.

Transmission série sur un seul fil pour une liaison asynchrone

L'émetteur et le récepteur ne sont pas *synchronisés*.

Le récepteur, pour se **synchroniser tout seul** :

- ▷ **connaît le débit** de transmission ;
- ▷ **recherche des transitions** pour se synchroniser et interpoler des mesures d'échantillonnage...
- ▷ **extrait l'horloge** des données :

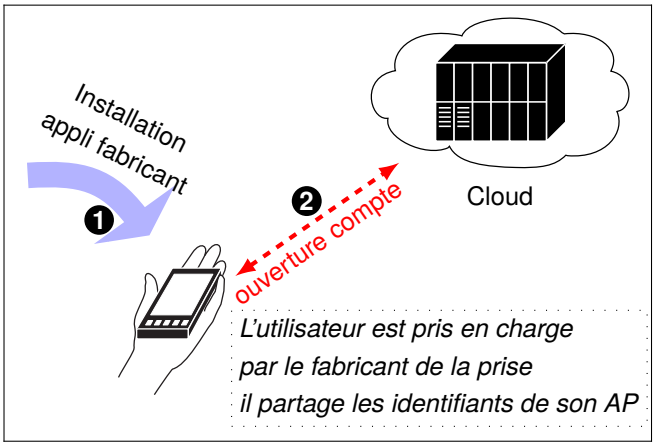


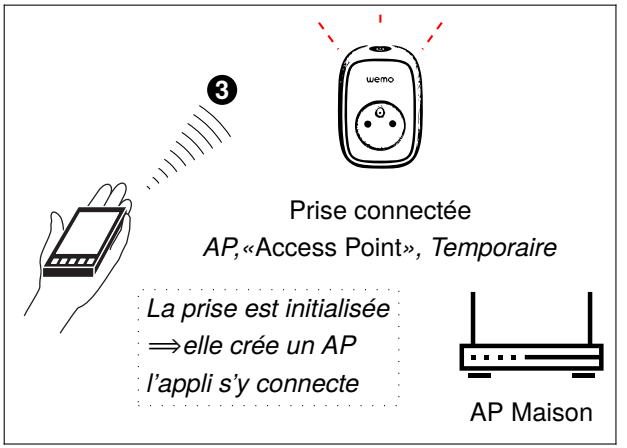
Importance du «*port série*»

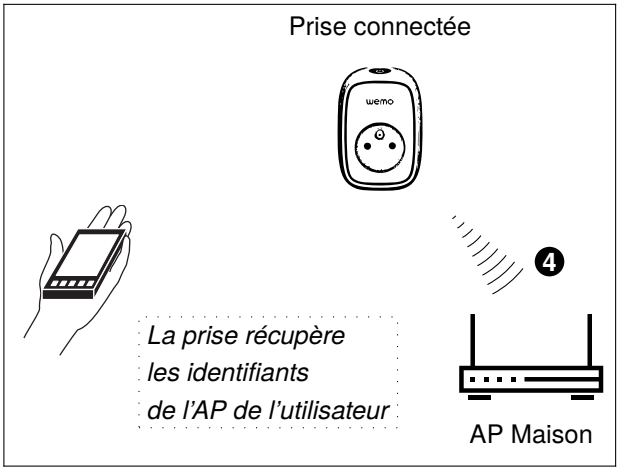
ou UART

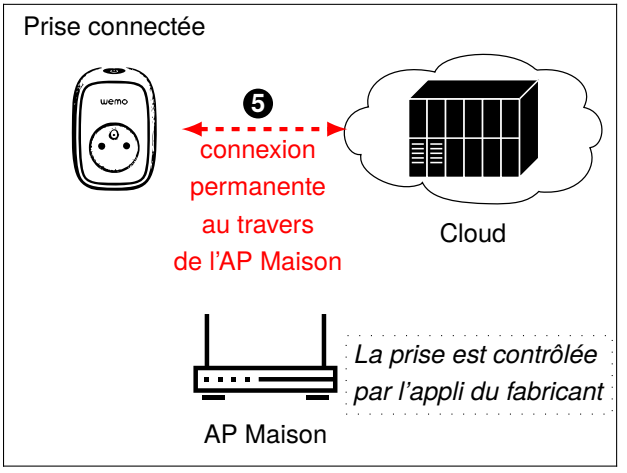
«*Universal Asynchronous Receiver/Transmitter*»

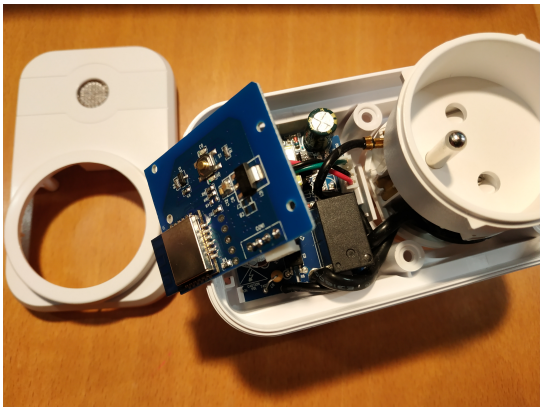
Scénario de l'intégration d'une prise connectée à la maison

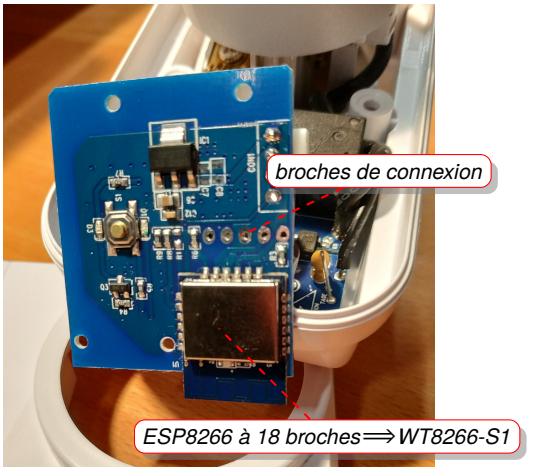






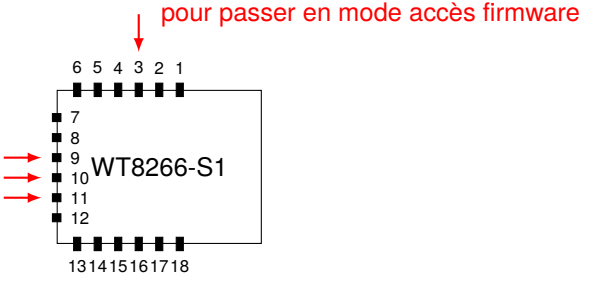






Identifier les broches du WT8266-S1 à l'aide d'un multimètre et de la doc constructeur

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 : communication par le port série, «*UART*»

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":"ffff8003ff000000ff000000ff00000000000000000","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"}
```

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.

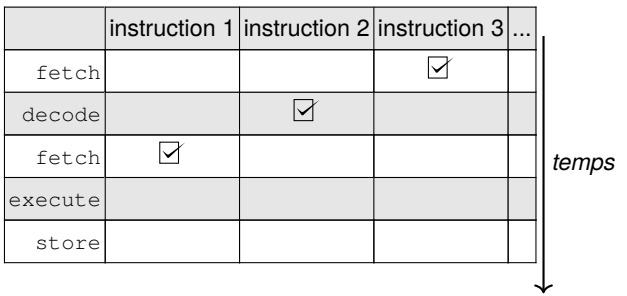
CPU Core	Unité programmable
MMU	Gestion de la mémoire virtuelle nécessaire pour un OS «*High End*»
DSP	Analyse de signal
Power Consumption	Batterie, génération de chaleur
Peripherals	A/D, UART, MAC, USB, Bluetooth, WiFi
Built-in RAM	vitesse et simplicité
Built-in cache	vitesse
Built-in EEPROM or FLASH	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.*)

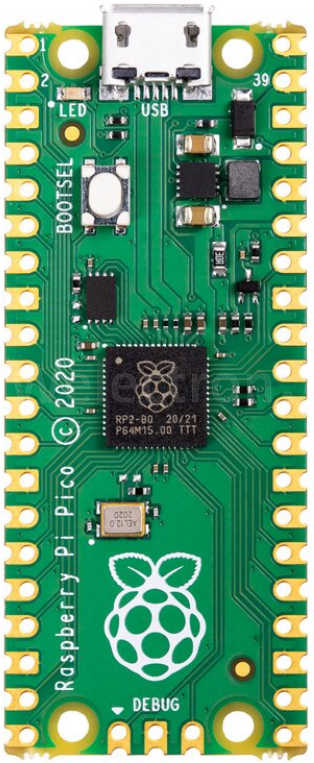
- **RTC**, «*Real Time Clock*» : maintenir l'heure et la date (utilisation d'une batterie séparée).
  Si le composant est «connecté» il peut utiliser un serveur NTP, «*Network Time Protocol*».

- **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.

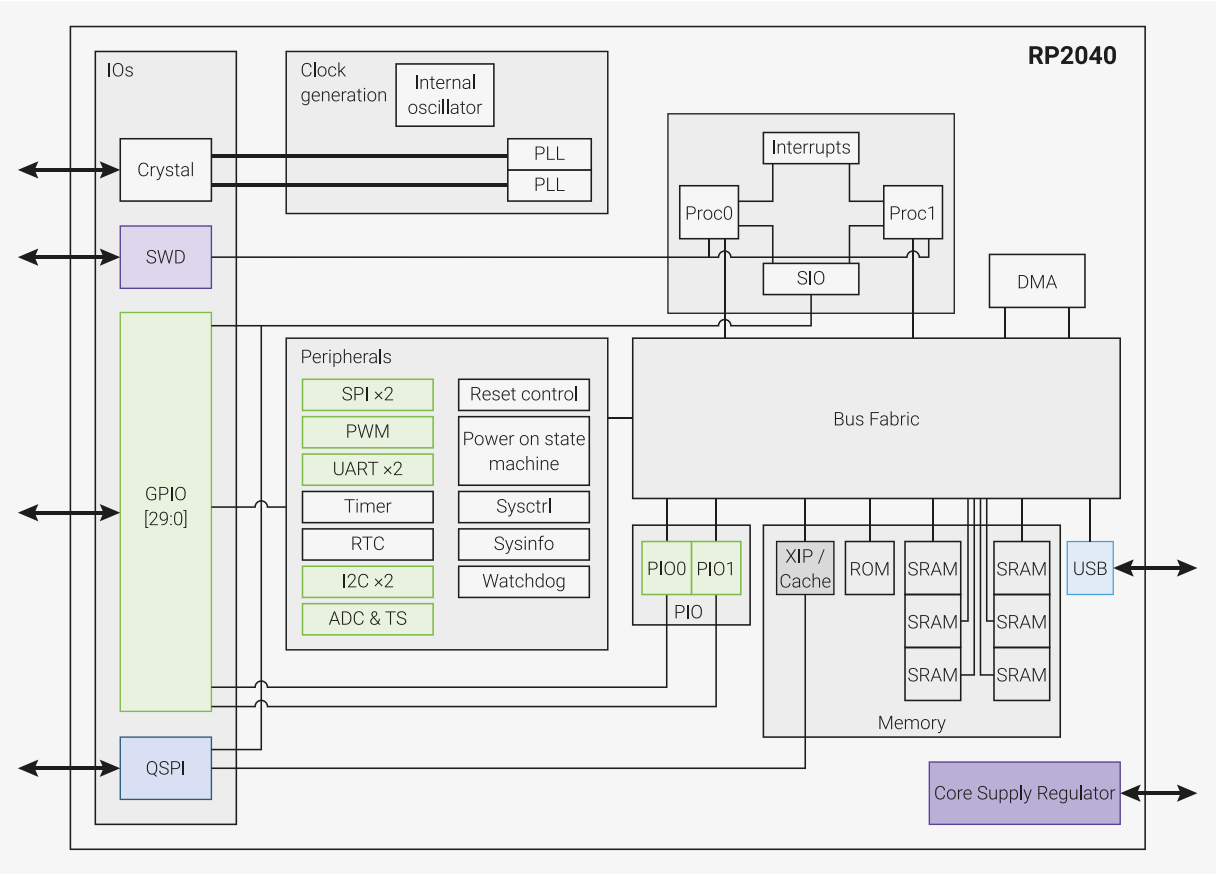


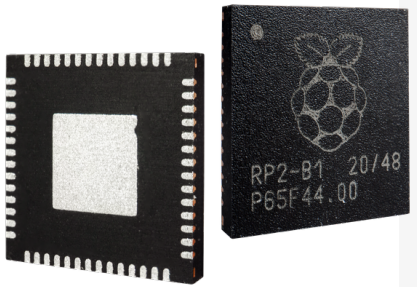
RP2040 is a low-cost, **high-performance microcontroller** device with **flexible** digital interfaces.

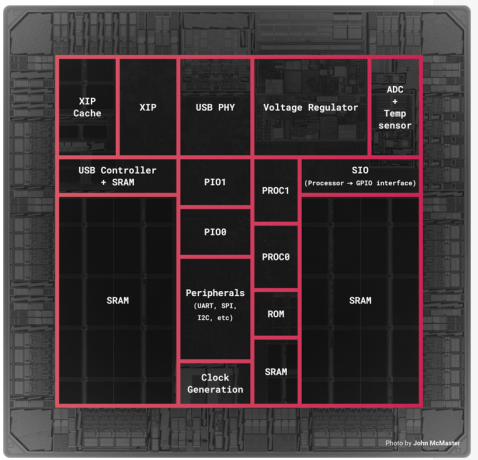
- **Dual Cortex M0+** processor cores, up to 133MHz ;
- **264kB** of embedded SRAM in 6 banks ;
- 30 multifunction **GPIO** ;
- 6 dedicated IO for **SPI Flash** (supporting XIP, «*eXecute-In-Place*» : considérer la mémoire Flash externe comme de la mémoire interne) ;
- Dedicated hardware for **commonly used peripherals** ;
- Programmable IO for **extended peripheral support** ;
- **4 channel ADC** with internal temperature sensor, 500ksps, 12-bit conversion ;
- USB 1.1 **Host**/**Device**.

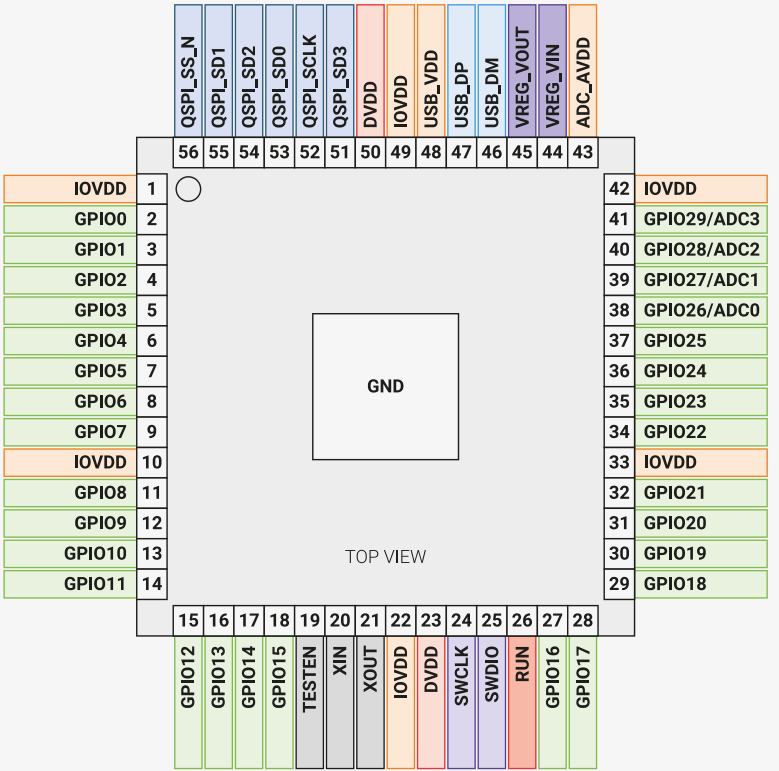
- **Code** may be executed **directly from external memory** through a dedicated SPI, DSPI or QSPI interface.
  A small cache improves performance for typical applications.
- **Debug** is available via the **SWD interface**.
- **Internal SRAM** can contain code or data. It is addressed as a single 264 kB region, but physically partitioned into 6 banks to allow **simultaneous parallel access** from different masters.
- **DMA** bus masters are available to offload repetitive data transfer tasks from the processors.
  GPIO pins can be driven directly, or from a variety of dedicated logic functions.
- **Dedicated hardware** for fixed functions such as SPI, I2C, UART.
- **Flexible configurable PIO** controllers can be used to provide a wide variety of IO functions.
- A **USB controller** with embedded PHY can be used to provide FS, «*Full Speed, 12Mbps*»/LS, «*Low Speed, 1,5Mbps*» Host or Device connectivity under software control.
  Four ADC inputs which are shared with GPIO pins.
- **Two PLLs** to provide a fixed 48MHz clock for USB or ADC, and a flexible system clock up to 133MHz.
- An **internal Voltage Regulator** to supply the core voltage so the end product only needs supply the IO voltage.

Présentation du Raspberry Pico : représentation «*schématique*»

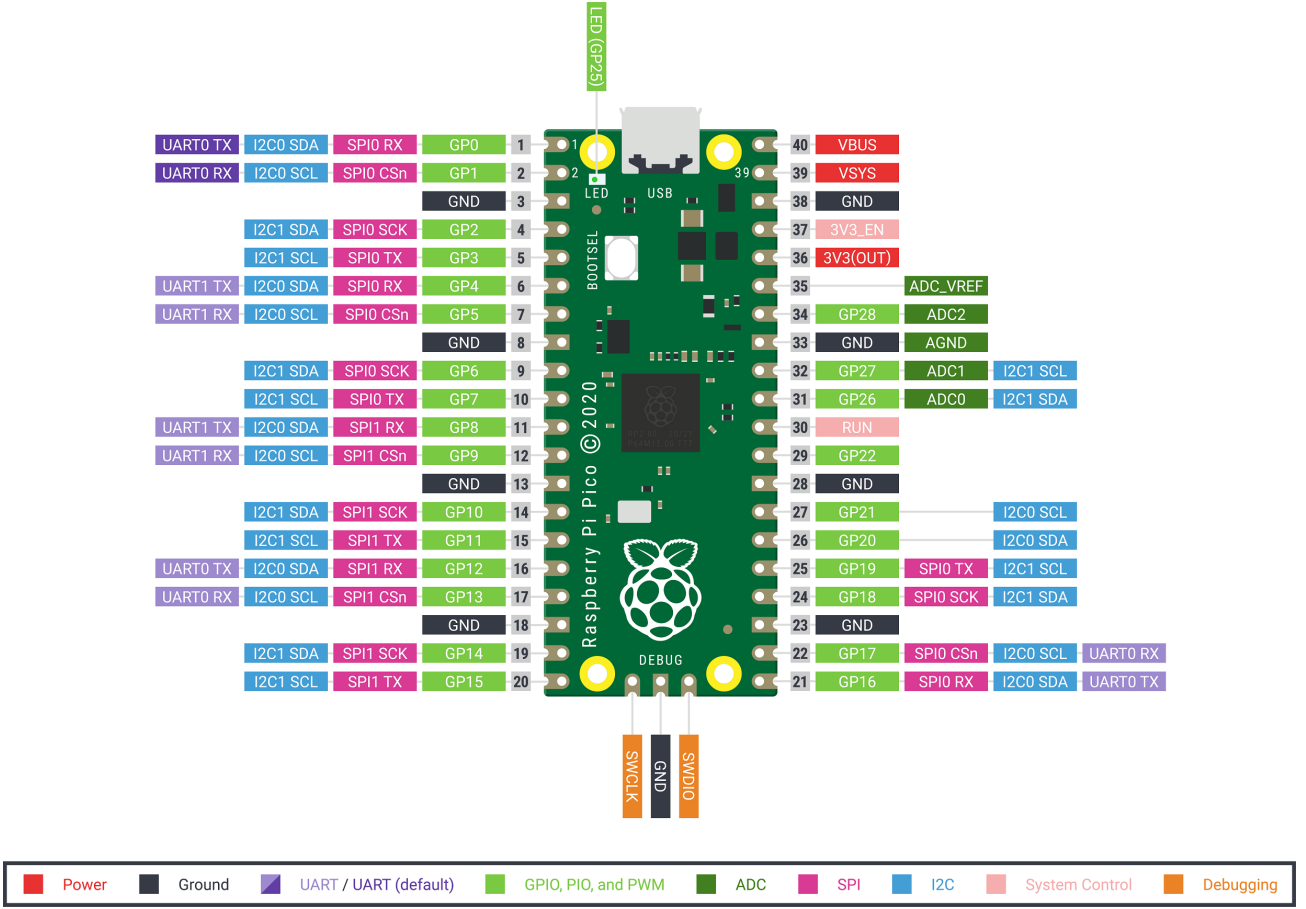



RP2-B1 20/48
P65F44.QO


XIP Cache
XIP
USB PHY
Voltage Regulator
ADC + Temp sensor
USB Controller + SRAM
PIO1
PROC1
SIO
(Processor → GPIO interface)
PIO0
PROC0
SRAM
Peripherals
(UART, SPI, I2C, etc)
ROM
SRAM
Clock Generation
SRAM
Photo by John McMaster

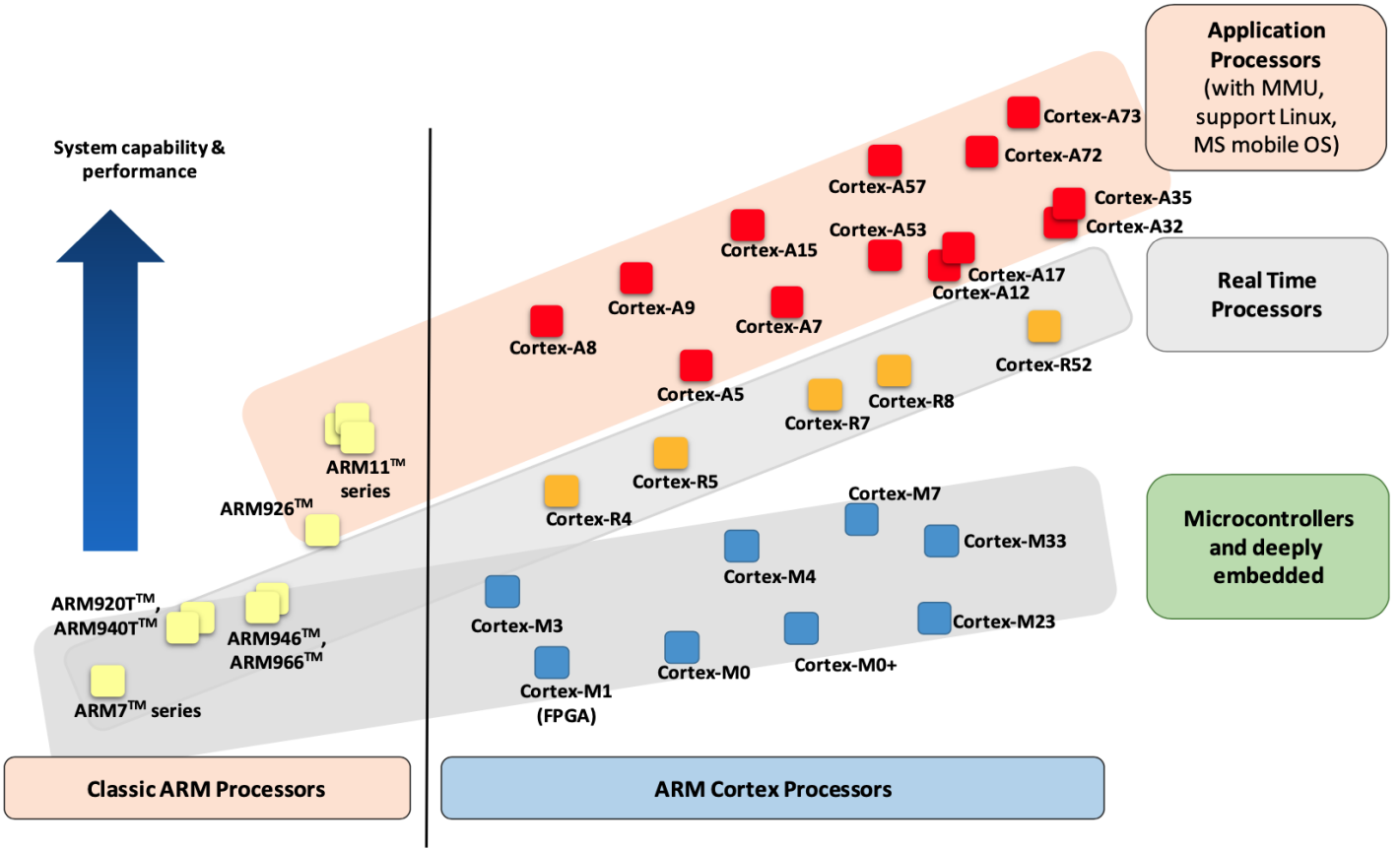

QSPI_SS_N
QSPI_SD1
QSPI_SD2
QSPI_SD0
QSPI_SCLK
QSPI_SD3
DVDD
IOVDD
USB_VDD
USB_DP
USB_DM
VREG_VOUT
VREG_VIN
ADC_AVDD
56 55 54 53 52 51 50 49 48 47 46 45 44 43
IOVDD 1
GPIO0 2
GPIO1 3
GPIO2 4
GPIO3 5
GPIO4 6
GPIO5 7
GPIO6 8
GPIO7 9
IOVDD 10
GPIO8 11
GPIO9 12
GPIO10 13
GPIO11 14
42 IOVDD
41 GPIO29/ADC3
40 GPIO28/ADC2
39 GPIO27/ADC1
38 GPIO26/ADC0
37 GPIO25
36 GPIO24
35 GPIO23
34 GPIO22
33 IOVDD
32 GPIO21
31 GPIO20
30 GPIO19
29 GPIO18
GND
TOP VIEW
15 16 17 18 19 20 21 22 23 24 25 26 27 28
GPIO12
GPIO13
GPIO14
GPIO15
TESTEN
XIN
XOUT
IOVDD
DVDD
SWCLK
SWDIO
RUN
GPIO16
GPIO17

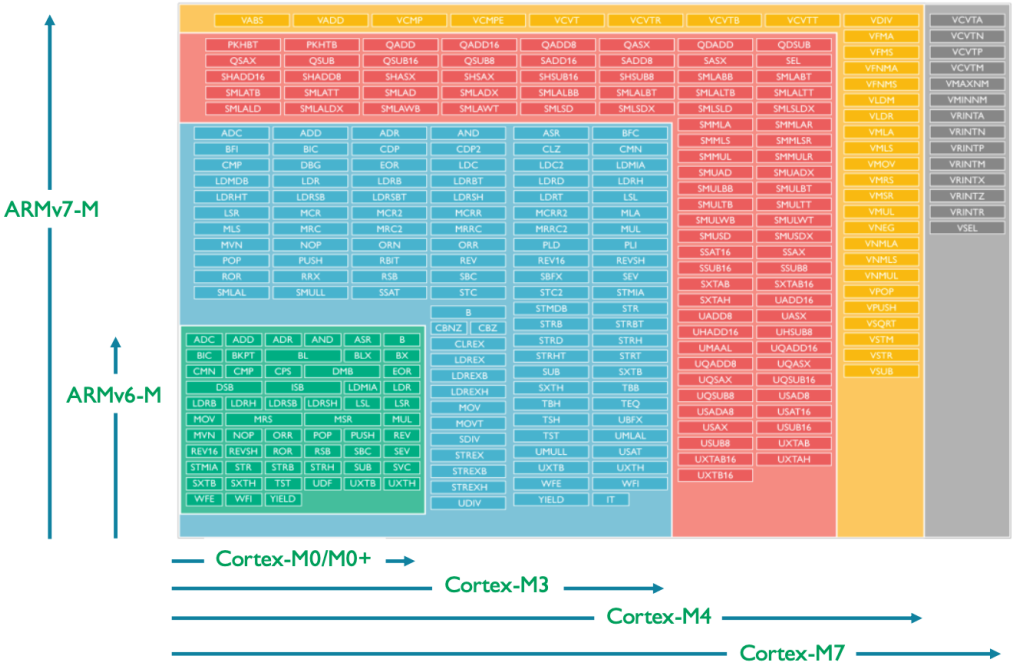
Présentation du Raspberry Pico : le brochage ou «*pinout*»



The ARM Cortex-M0+ processor features and benefits are:

- **Tight integration** of system peripherals reduces area and development costs.
- **Thumb instruction** set combines high code density with 32-bit performance.
- Support for **single-cycle I/O** access.
- **Power control** optimization of system components.
- **Integrated sleep modes** for low-power consumption.
- **Fast code execution** enables running the processor with a slower clock or increasing sleep mode time.
- **Optimized code fetching** for reduced flash and ROM power consumption.
- Hardware **multiplier**.
- **Deterministic**, high-performance «interrupt handling» for time-critical applications.
- **Deterministic instruction cycle timing**.
- Support for **system level debug authentication**.
- **Serial Wire Debug** reduces the number of pins required for debugging.


System capability & performance
Application Processors (with MMU, support Linux, MS mobile OS)
Real Time Processors
Microcontrollers and deeply embedded
Cortex-A73
Cortex-A72
Cortex-A57
Cortex-A35
Cortex-A32
Cortex-A53
Cortex-A15
Cortex-A17
Cortex-A12
Cortex-A9
Cortex-A7
Cortex-A8
Cortex-R52
Cortex-A5
Cortex-R8
Cortex-R7
ARM11™ series
Cortex-R5
ARM926™
Cortex-R4
Cortex-M7
Cortex-M33
Cortex-M4
ARM920T™, ARM940T™
ARM946™, ARM966™
Cortex-M3
Cortex-M23
Cortex-M0
Cortex-M0+
Cortex-M1 (FPGA)
ARM7™ series
Classic ARM Processors
ARM Cortex Processors

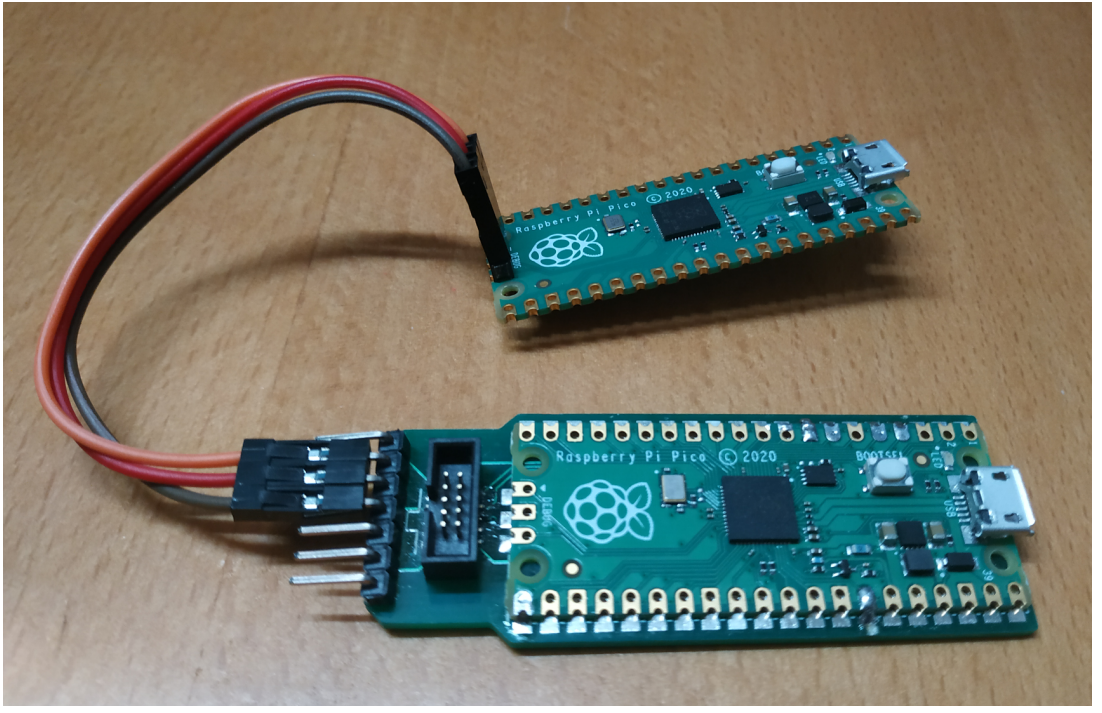


Floating Point

DSP (SIMD, fast MAC)

Advanced data processing
bit field manipulations

General data processing
I/O control tasks



- ▷ le premier Raspberry Pi Pico sert de **sonde**, «*probe*», de débogage ;
- ▷ le second est sous contrôle matériel du premier :
  - ◇ on peut interrompre/reprendre l'exécution de son firmware ;
  - ◇ on peut connaître le contenu des registres et de la mémoire lors de l'arrêt de l'exécution.

On liste les périphériques USB connectés :

xterm
```
$ lsusb
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 007: ID 2e8a:0005 MicroPython Board in FS mode
```

La connexion du Raspberry Pico crée un «*device*» série :

xterm
```
$ ls /dev
... ttyACM0 ...
```

On peut voir les événements «*hardware*» :

xterm
```
$ sudo dmesg
...
[ 2049.165618] usb 1-9.4: new full-speed USB device number 7 using xhci_hcd
[ 2049.271582] usb 1-9.4: New USB device found, idVendor=2e8a, idProduct=0005, bcdDevice= 1.00
[ 2049.271596] usb 1-9.4: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[ 2049.271602] usb 1-9.4: Product: Board in FS mode
[ 2049.271607] usb 1-9.4: Manufacturer: MicroPython
[ 2049.271611] usb 1-9.4: SerialNumber: e6614103e76d1f22
[ 2049.309954] cdc_acm 1-9.4:1.0: ttyACM0: USB ACM device
[ 2049.310040] usbcore: registered new interface driver cdc_acm
[ 2049.310045] cdc_acm: USB Abstract Control Model driver for USB modems and ISDN adapters
[ 3319.456183] usb 1-9.4: USB disconnect, device number 7
[ 3492.738513] usb 1-9.4: new full-speed USB device number 8 using xhci_hcd
[ 3492.840166] usb 1-9.4: New USB device found, idVendor=2e8a, idProduct=0004, bcdDevice= 1.00
[ 3492.840181] usb 1-9.4: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[ 3492.840187] usb 1-9.4: Product: Picoprobe
[ 3492.840192] usb 1-9.4: Manufacturer: Raspberry Pi
[ 3492.840195] usb 1-9.4: SerialNumber: E6614C30930D512E
[ 3492.855791] cdc_acm 1-9.4:1.0: ttyACM0: USB ACM device
```

nouveau périphérique USB connecté

création du device série lié au Raspberry Pico branché en USB

xterm

```
$ minicom -D /dev/ttyACM0

Welcome to minicom 2.8

OPTIONS: I18n
Port /dev/ttyACM0, 20:23:10

Press CTRL-A Z for help on special keys

>>>
MPY: soft reboot
MicroPython v1.19.1 on 2022-06-18; Raspberry Pi Pico with RP2040
Type "help()" for more information.
>>> help()
Welcome to MicroPython!

For online help please visit https://micropython.org/help/.

For access to the hardware use the 'machine' module.  RP2 specific commands
are in the 'rp2' module.

Quick overview of some objects:
  machine.Pin(pin) -- get a pin, eg machine.Pin(0)
  machine.Pin(pin, m, [p]) -- get a pin and configure it for IO mode m, pull mode p
    methods: init(..), value([v]), high(), low(), irq(handler)
...
  machine.Timer(freq, callback) -- create a software timer object
    eg: machine.Timer(freq=1, callback=lambda t:print(t))

Pins are numbered 0-29, and 26-29 have ADC capabilities
Pin IO modes are: Pin.IN, Pin.OUT, Pin.ALT
Pin pull modes are: Pin.PULL_UP, Pin.PULL_DOWN

Useful control commands:
  CTRL-C -- interrupt a running program
  CTRL-D -- on a blank line, do a soft reset of the board
  CTRL-E -- on a blank line, enter paste mode

For further help on a specific object, type help(obj)
For a list of available modules, type help('modules')
```

On se connecte sur le port série à l'aide de la commande `minicom`

Faire clignoter une LED

xterm

```
>>> import machine
>>> led = machine.Pin(25, machine.Pin.OUT)
>>> led.toggle()
>>> led.toggle()
>>> def blink(timer):
...     led.toggle()
...
...
...
>>> timer.init(freq=2.5, mode=machine.Timer.PERIODIC,callback=blink)
```

On allume la LED

On éteint la LED

Utiliser le port série

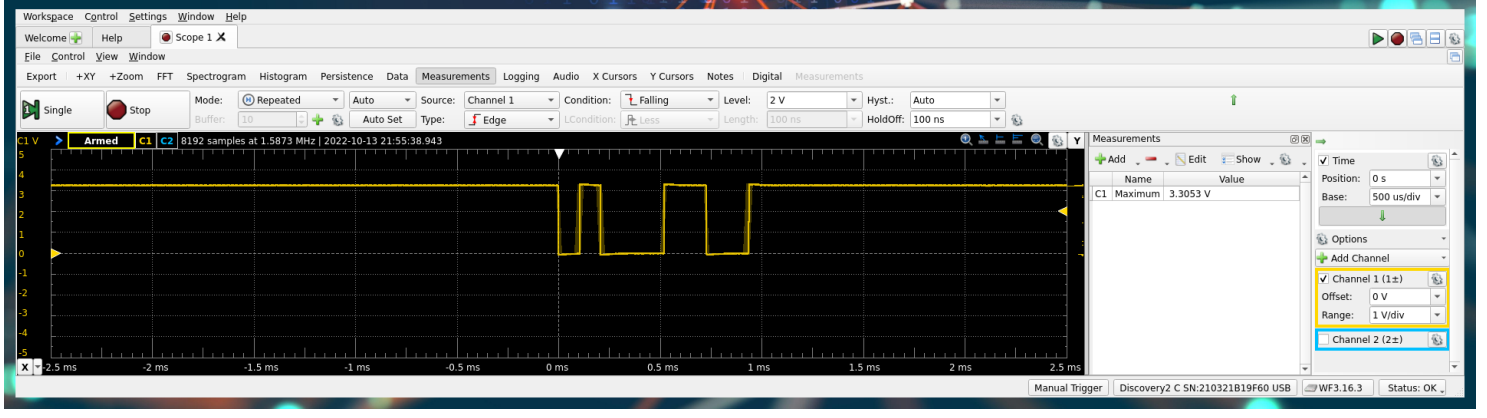
xterm

```
MPY: soft reboot
MicroPython v1.19.1 on 2022-10-11; Raspberry Pi Pico with RP2040
Type "help()" for more information.
>>> from machine import Pin, Timer, UART
>>> uart = UART(0, baudrate=9600, tx=Pin(0), rx=Pin(1))
>>> def envoi(timer):
...     uart.write(b'1')
...
...
...
>>> timer = Timer()
>>> timer.init(freq=.5, mode=Timer.PERIODIC, callback = envoi)
```

On utilise les broches 0 et 1 du Raspberry Pico

À l'aide d'un oscilloscope on peut «*intercepter*» la transmission série sur la broche 0 :

```
MPY: soft reboot                                                    [22/8976]
MicroPython v1.19.1 on 2022-10-11; Raspberry Pi Pico with RP2040
Type "help()" for more information.
>>> from machine import Pin, Timer, UART
>>> uart = UART(0, baudrate=9600, tx=Pin(0), rx=Pin(1))
>>> def envoi(timer):
...     uart.write(b'1')
...
...
...
>>> timer = Timer()
>>> timer.init(freq=.5, mode=Timer.PERIODIC, callback = envoi)
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
```



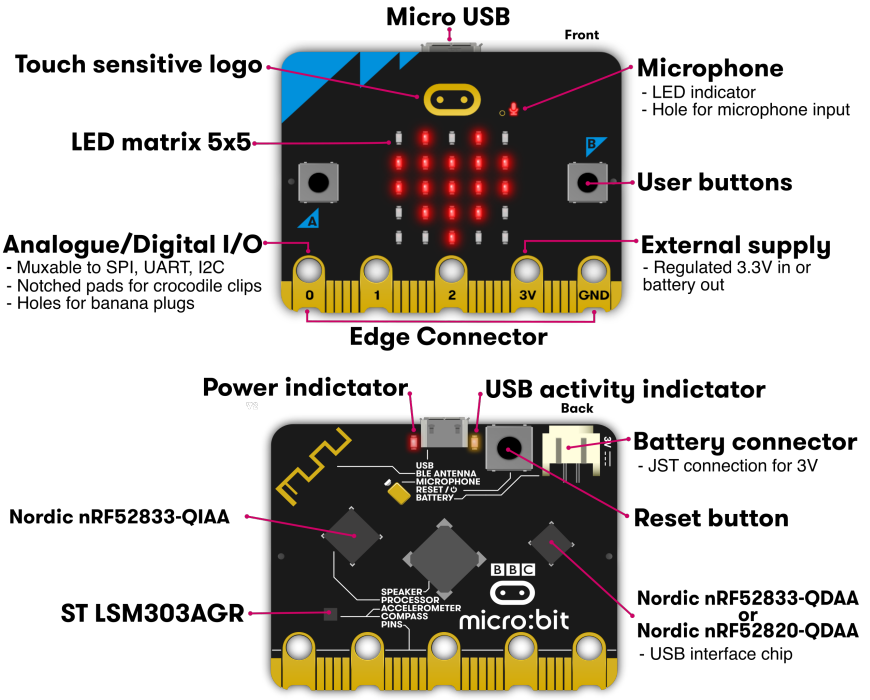
Lorsque le port série ne transmet rien, la broche est au niveau haut, c-à-d à 3,3v.

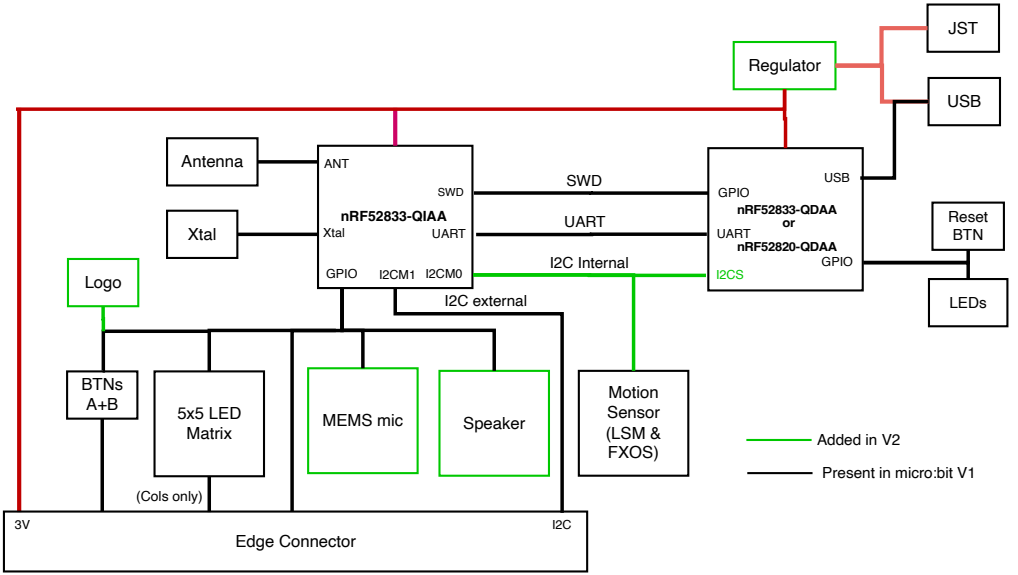
Grâce à **l'oscilloscope**, on peut déterminer la **vitesse de transmission** :

Ici, on voit que $1/\Delta X = 9,6KHz$ ce qui est le cas : on a configuré le port série pour une transmission à 9600bps :

xterm

```
...
>>> uart = UART(0, baudrate=9600, tx=Pin(0), rx=Pin(1))
...
```

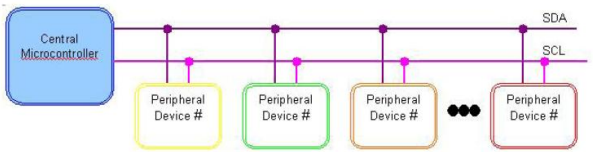

Micro USB
Front
Touch sensitive logo
Microphone
- LED indicator
- Hole for microphone input
LED matrix 5x5
User buttons
Analogue/Digital I/O
- Muxable to SPI, UART, I2C
- Notched pads for crocodile clips
- Holes for banana plugs
External supply
- Regulated 3.3V in or battery out
0
1
2
3V
GND
Edge Connector
Power indictator
USB activity indictator
Back
Battery connector
- JST connection for 3V
Reset button
Nordic nRF52833-QIAA
ST LSM303AGR
Nordic nRF52833-QDAA
or
Nordic nRF52820-QDAA
- USB interface chip
micro:bit



Model	Nordic nRF52833
Core variant	Arm Cortex-M4 32 bit with FPU
Flash ROM	512KB
RAM	128KB
Speed	64MHz
Debug	SWD, J-Link/OB
More Info	Software, nRF52 datasheet

Bluetooth Wireless Communication	Bluetooth 5.1/Bluetooth Low Energy(BLE)
Low level radio communications	2.4 GHz, 1Mbps or 2Mbps
Buttons	2 tactile user buttons
Display	5x5 matrix surface mount red LED
Motion sensor	3 magnetic field and 3 acceleration axes
Other	Temperature sensor, Speaker, Mic.
GPIOs	I2C, PWM, Serial, SPI, *etc.*

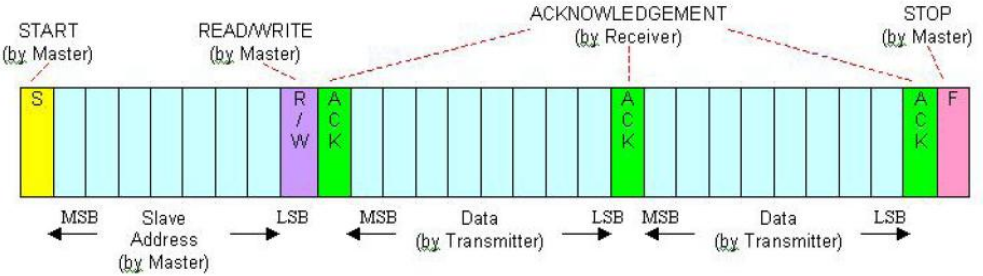
Et les communications entre les composants ?



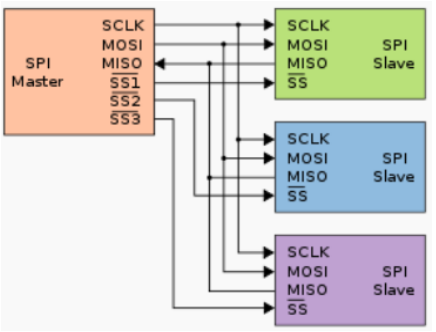
Le bus I2C, «*Inter-Integrated Circuit*» :

- ⋆ 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 ;

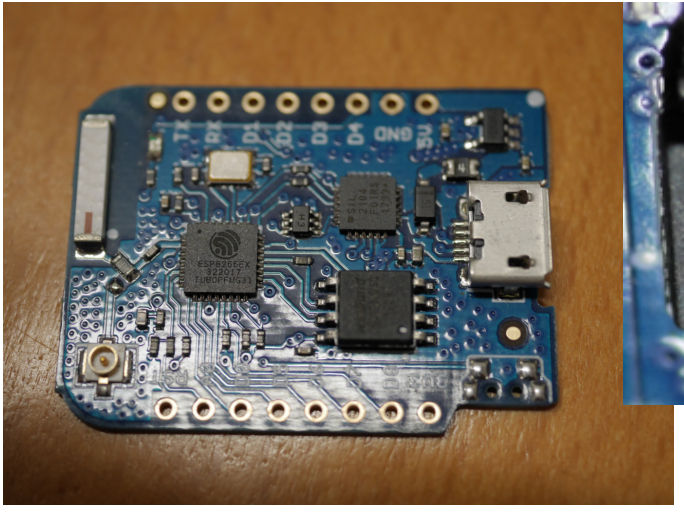
- 4 signaux :
  - `SCLK`, «*Clock*» : l'horloge obligatoire pour la transmission synchrone ;
  - `MOSI`, «*Master Out Slave Input*» : communication Master ⟹ Slave ;
  - `MISO`, «*Master Input Slave Output*» : communication Slave ⟹ Master ;
  - `SS`, «*Slave Select*» : un fil par Slave pour pouvoir le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

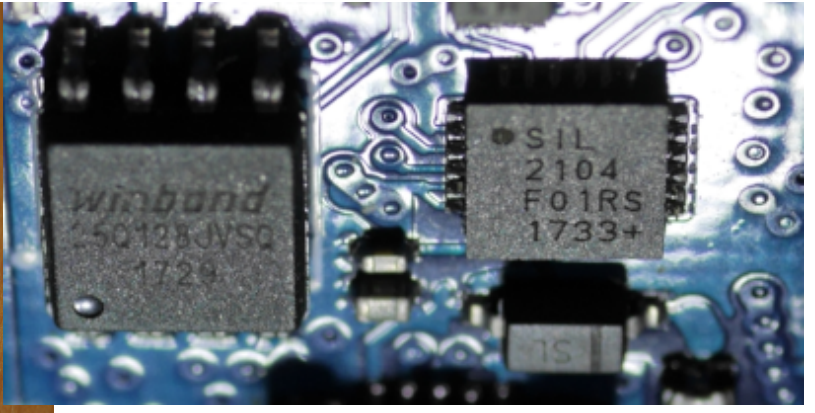
«SPI *vs* I2C» : Quel bus choisir ?

- 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

Et concrètement une communication SPI
ça donne quoi ?



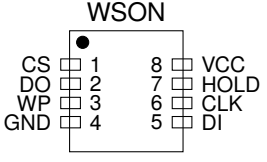


On peut lire la référence du composant mémoire :
Winbond 25Q128JVSQ

On peut récupérer la documentation à :

`https://www.winbond.com/resource-files/W25Q128JV%20RevI%2008232021%20Plus.pdf`

On obtient la description des broches du composant :

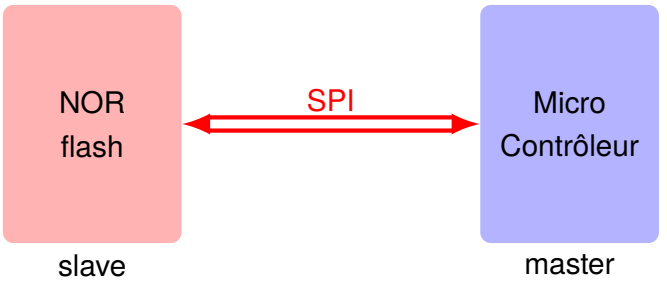


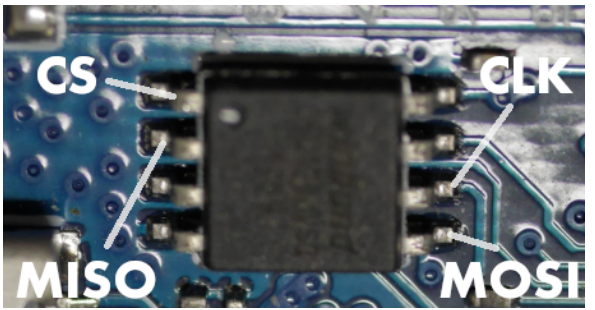
Ce qui permet de le connecter au **Bus Pirate**.

WSON	Bus Pirate
VCC	3.3v
GND	GND
CS	CS
CLK	CLK
DO	MISO
DI	MOSI

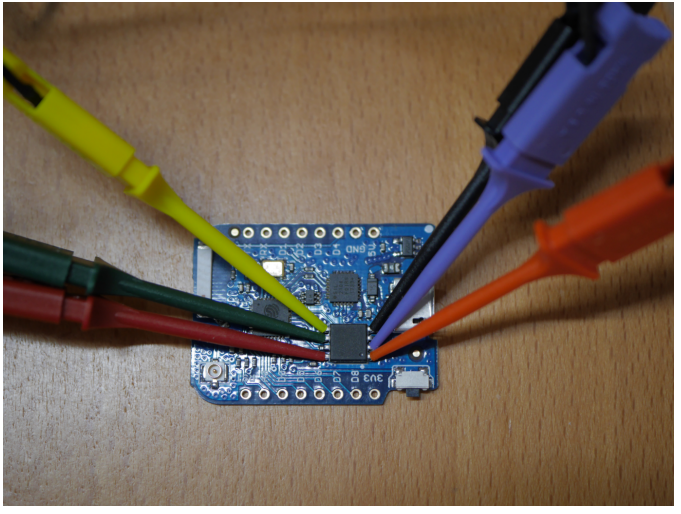
NOR Flash : mémoire accessible par SPI

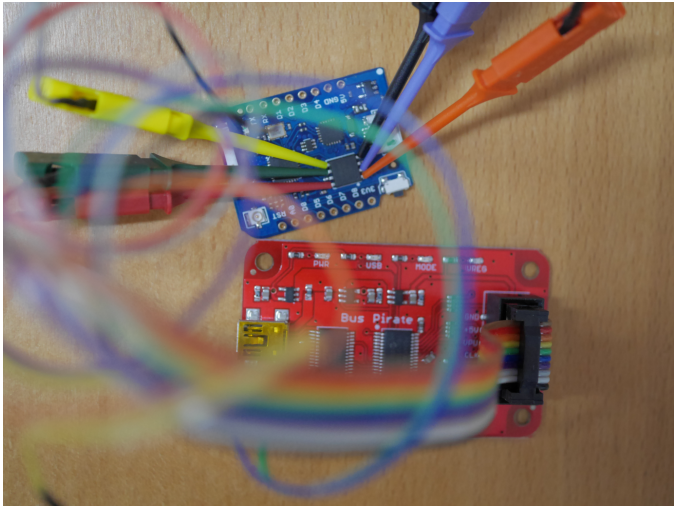
- Support de stockage pour données **non volatiles** : les données restent dans le composant jusqu'à ce qu'elles soient réécrites et elles sont conservées même si le circuit embarqué est éteint ;
- Les données peuvent être lues **octet par octet** : avec de la mémoire NAND, on ne peut lire qu'un, dix ou 100 octets à la fois : par page de 4096 ;
- Mémoire **sans défaut** : pas de système de correction d'erreur nécessaire contrairement à la mémoire de type NAND ;
- **faible latence** d'accès : utilisable pour l'exécution directe de code ;
- utilisée en général pour stocker des données en faible quantités ;
- communication par le bus SPI : communication **synchrone**, **full duplex** avec 4 signaux : CS, CLK, MISO et MOSI.



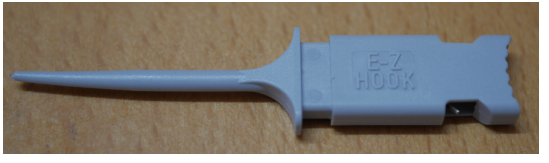


On utilise des sondes, «*probes*» pour se connecter à chaque broche du composant mémoire :





Chacune de ces sondes est reliées au **Bus Pirate**.



Le **Bus Pirate** est un circuit opensource qui permet :

- ▷ d'alimenter le composant à tester en 3,3v ou 5v ;
- ▷ de disposer d'une interface USB/série ;
- ▷ de disposer d'un **micro-contrôleur** dédié à :
  - ◇ lire des **ordres** en provenance de la machine de l'utilisateur par l'intermédiaire du port série simulé par USB ;
  - ◇ **interpréter** ces ordres avec un firmware spécialisé opensource ;
  - ◇ de gérer **différents protocoles de communication** inter-composants : I2C, SPI, 1-Wire, ... ;
  - ◇ **d'envoyer et recevoir des données** du protocole choisi sur des broches connectées au composant à tester ;
- ▷ *Ici, on va utiliser le bus SPI pour communiquer avec le composant mémoire.*

Identification du composant en utilisant le bus SPI :

8.1 **Device ID and Instruction Set Tables**

8.1.1 Manufacturer and Device Identification

MANUFACTURER ID	**(MF7 - MF0)**	
Winbond Serial Flash	EFh	
Device ID	**(ID7 - ID0)**	**(ID15 - ID0)**
Instruction	**ABh, 90h, 92h, 94h**	**9Fh**
W25Q128JV-IN/IQ/JQ	17h	4018h
W25Q128JV-IM*/JM*	17h	7018h

Note: For DTR, QPI supporting, please refer to W25Q128JV-M DTR datasheet.

8.1.2 Instruction Set Table 1 (Standard SPI Instructions)(1)

Data Input Output	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Number of Clock(1-1-1)	**8**	**8**	**8**	**8**	**8**	**8**	**8**
Write Enable	**06h**						
Volatile SR Write Enable	**50h**						
Write Disable	**04h**						
Release Power-down / ID	**ABh**	Dummy	Dummy	Dummy	(ID7-ID0)(2)		
Manufacturer/Device ID	**90h**	Dummy	Dummy	00h	(MF7-MF0)	(ID7-ID0)	
JEDEC ID	**9Fh**	(MF7-MF0)	(ID15-ID8)	(ID7-ID0)			
Read Unique ID	**4Bh**	Dummy	Dummy	Dummy	Dummy	(UID63-0)	

La commande SPI est `0x9f` et la réponse devrait être : `(MF)(ID)(ID)` où :

- `MF` vaut `EF` ;
- `ID` soit `4018` ou `7018`.

xterm

```
$ minicom
Welcome to minicom 2.8
OPTIONS: I18n
Port /dev/ttyUSB0, 17:08:40
Press CTRL-A Z for help on special keys
HiZ>m
1. HiZ
2. 1-WIRE
3. UART
4. I2C
5. SPI
6. 2WIRE
7. 3WIRE
8. KEYB
9. LCD
10. PIC
11. DIO
x. exit(without change)
(1)>5
Set speed:
 1.  30KHz
 2. 125KHz
 3. 250KHz
 4.   1MHz
 5.  50KHz
 6. 1.3MHz
 7.   2MHz
 8. 2.6MHz
 9. 3.2MHz
10.   4MHz
11. 5.3MHz
12.   8MHz
(1)>4
Clock polarity:
 1. Idle low *default
 2. Idle high
(1)>
```

xterm

```
Output clock edge:
 1. Idle to active
 2. Active to idle *default
(2)>
Input sample phase:
 1. Middle *default
 2. End
(1)>
CS:
 1. CS
 2. /CS *default
(2)>
Select output type:
 1. Open drain (H=Hi-Z, L=GND)
 2. Normal (H=3.3V, L=GND)
(1)>2
Clutch disengaged!!!
To finish setup, start up the power
supplies with command 'W'
Ready
SPI>W
POWER SUPPLIES ON
Clutch engaged!!!
SPI>[0x9f r:3]
/CS ENABLED
WRITE: 0x9F
READ: 0xEF 0x40 0x18
/CS DISABLED
SPI>
```

On :

- ▷ se connecte au **Bus Pirate** avec `minicom` ;
- ▷ envoie la commande `[0x9f r:3]` qui envoie la commande `0x9f` SPI et lit 3 octets en réponse ;
- ▷ reçoit les 3 octets `0xEF 0x40 0x18`

⟹ **le composant est bien identifié** !

Utilisation de l'outil `flashrom`

xterm
```
$ flashrom -p buspirate_spi:dev=/dev/ttyUSB0,spispeed=1M
flashrom v1.2 on Linux 5.15.0-47-generic (x86_64)
flashrom is free software, get the source code at https://flashrom.org

Using clock_gettime for delay loops (clk_id: 1, resolution: 1ns).
No EEPROM/flash device found.
Note: flashrom can never write if the flash chip isn't found automatically.
```

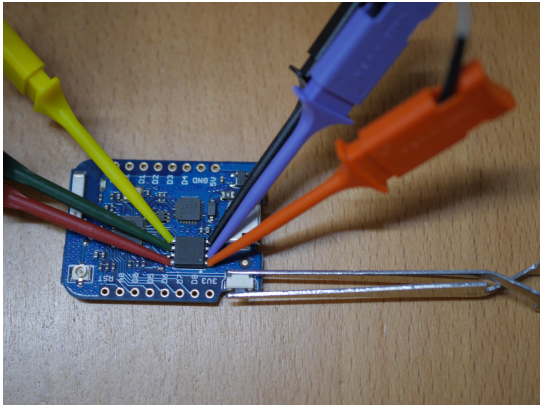
Il y a un **conflit d'accès** au composant mémoire :

- le processeur accède à la mémoire pour **exécuter le firmware** ;
- le Bus Pirate accède à la mémoire pour **l'identifier**.

$\Longrightarrow$ il faut **bloquer le processeur** en le maintenant dans l'état RESET !

*Heureusement, sur la carte de développement un **bouton RESET** permet de le faire.*

$\Longrightarrow$On va utiliser une **pince** pour le bloquer.



xterm
```
$ flashrom -p buspirate_spi:dev=/dev/ttyUSB0
flashrom v1.2 on Linux 5.15.0-47-generic (x86_64)
flashrom is free software, get the source code at https://flashrom.org

Using clock_gettime for delay loops (clk_id: 1, resolution: 1ns).
Found Winbond flash chip "W25Q128.V" (16384 kB, SPI) on buspirate_spi.
No operations were specified.
```

Le composant est correctement identifié !

Récupération finale du firmware

xterm

```
$ flashrom -p buspirate_spi:dev=/dev/ttyUSB0 -r dump_esp8266.bin
flashrom v1.2 on Linux 5.15.0-47-generic (x86_64)
flashrom is free software, get the source code at https://flashrom.org

Using clock_gettime for delay loops (clk_id: 1, resolution: 1ns).
Found Winbond flash chip "W25Q128.V" (16384 kB, SPI) on buspirate_spi.
Reading flash... done.
```

Contenu du firmware

xterm

```
$ strings dump_esp8266.bin
...
Access denied
WebREPL connected
>>>
Password:
wH&@wH&@SH&@#H&@#H&@;H&@wH&@
9(@_
...
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept:
258EAFA5-E914-47DA-95CA-C5AB0DC85B11
Sec-WebSocket-Key:
Not a websocket request
Sec-WebSocket-Key
...
Welcome to MicroPython!
For online docs please visit http://docs.micropython.org/en/latest/esp8266/ .
For diagnostic information to include in bug reports execute 'import port_diag'.
Basic WiFi configuration:
```

la commande `strings` permet d'extraire les chaînes de caractères

Peut-être un token secret...

Le firmware est basé sur microPython !

Comment fonctionne l'outil `flashrom` ?

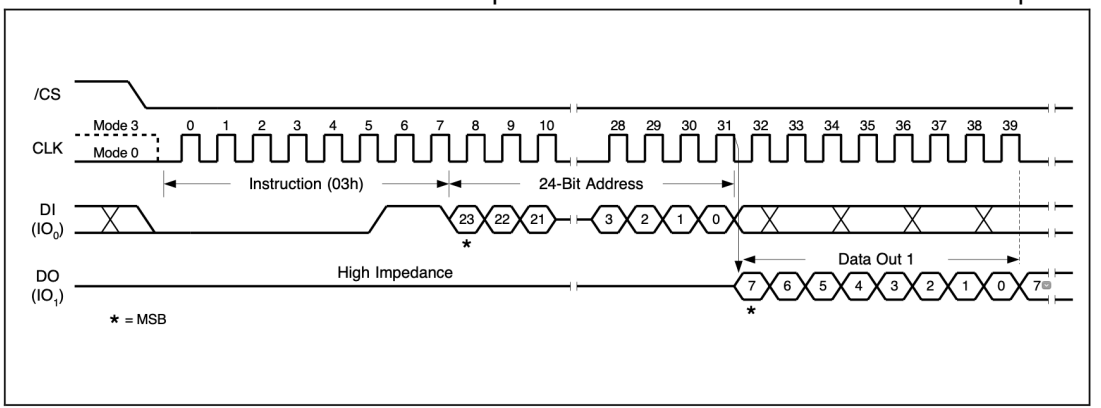
Il utilise le **Bus Pirate** pour effectuer les échanges sur le bus SPI.

Il utilise la commande de lecture `0x03` pour lire la mémoire octet par octet :

Read Data	03h	A23-A16	A15-A8	A7-A0	(D7-D0)	

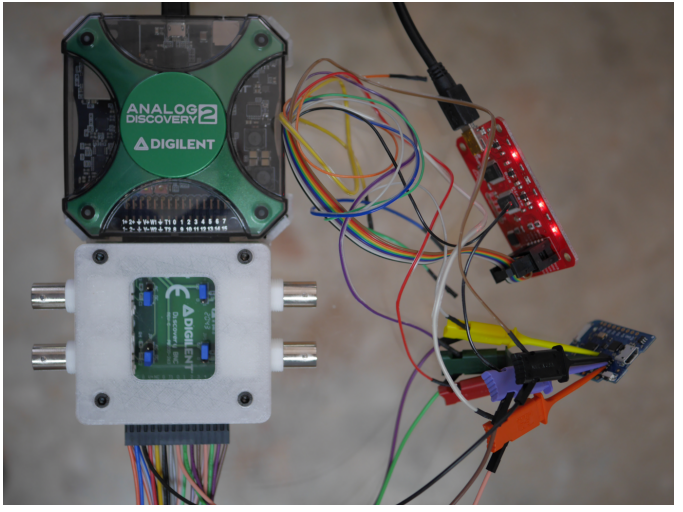
Il transmet `0x03 0x00 0x00 0x00` pour obtenir l'octet à l'adresse `0x000000`.

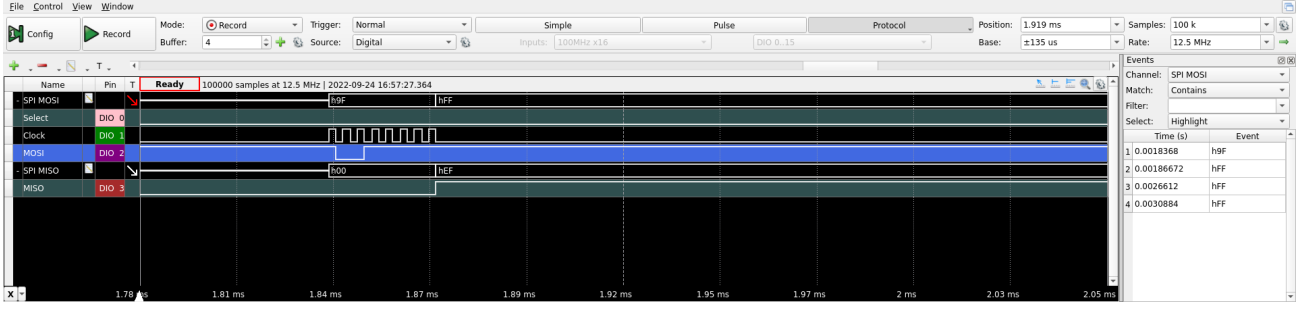
Il transmet la commande sur le bus SPI et récupère l'octet de mémoire à l'adresse indiquée :



Ici, il balaye l'ensemble des 16Mo de mémoire pour récupérer l'intégralité du firmware.

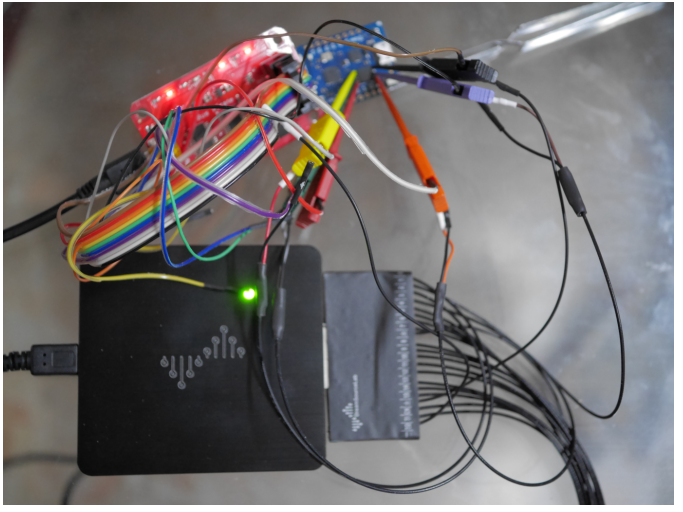
Et dans un analyseur logique ?



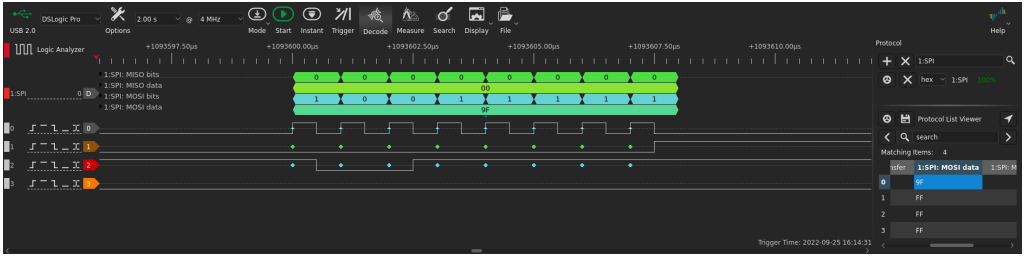


Ici on peut voir le `0x9f` transmis sur la broche MOSI et durant la réponse du composant, le master transmets `0xff`.

Et dans un autre analyseur logique ?

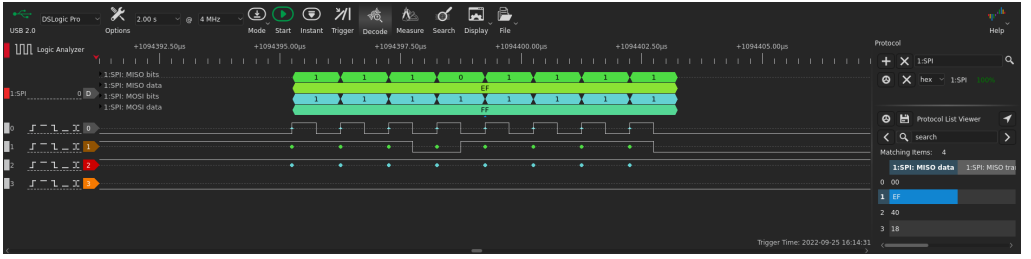


Ici on peut voir le `0x9f` transmis sur la broche MOSI :

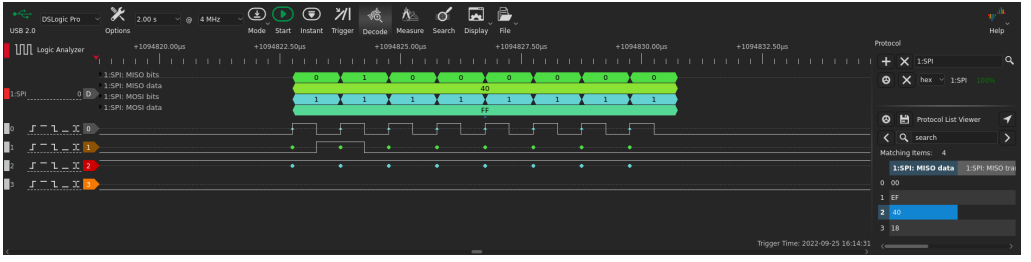


La réponse du composant mémoire :

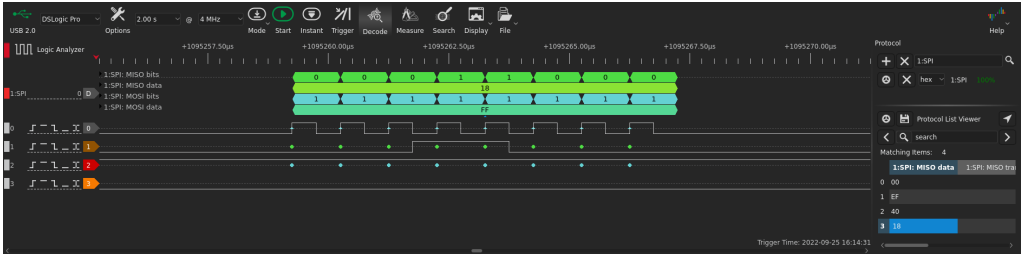
- le `0xEF` :



- le `0x40` :



- et enfin le `0x18` :



Et 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

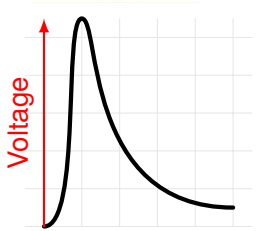
Boucle de «*polling*» utilisée pour surveiller une entrée qui ne s'arrête que lorsque l'entrée passe d'un état à un autre :

- ▷ *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*»

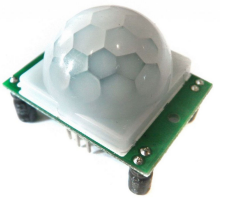
Capteur PIR : capteur électronique qui mesure la **quantité de lumière infrarouge** diffusée par les objets dans son champs de vision.

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.



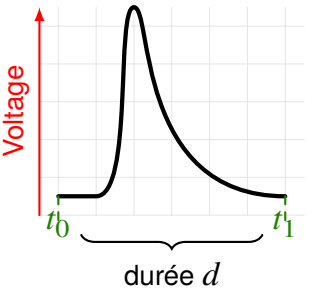
$\Longrightarrow$ application à la détection de mouvement : on détermine la différence entre la mesure «*habituelle*», température de la pièce, et un changement soudain, température du corps.

$\Longrightarrow$ 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*» :

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

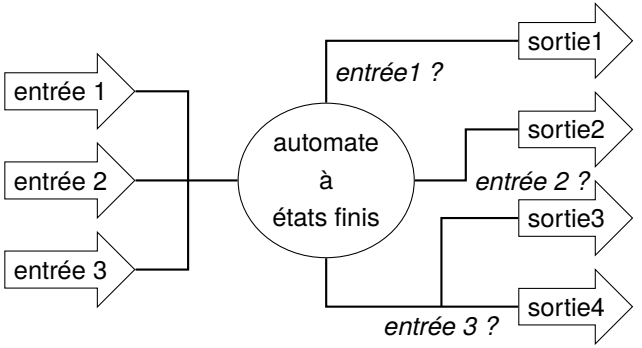
La **lecture régulière** par le micro-contrôleur correspond à faire du «*polling*».
$\Longrightarrow$cette lecture doit être intégrée précisément dans une fonction de `travail` dont on garantit la **fréquence d'appel** en fonction de d_m.

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

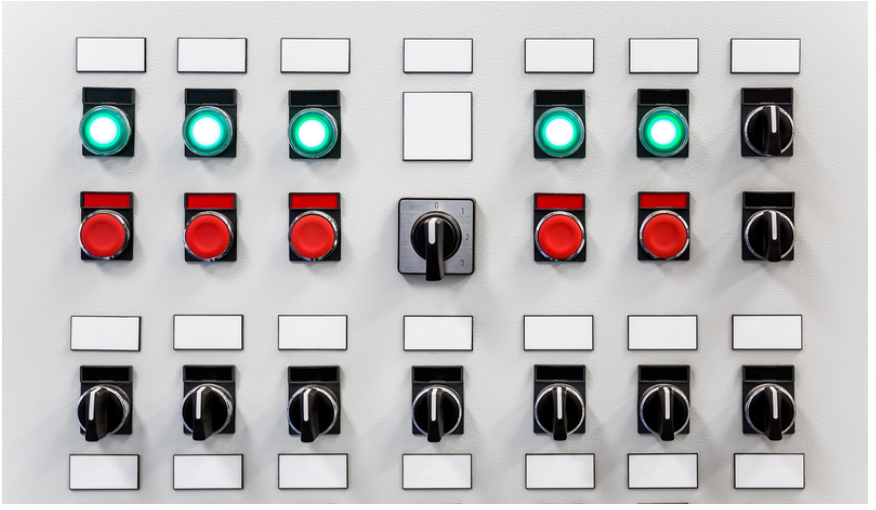


- □ événement sur «*entrée 1*» ⟹ «sortie 1» ;
- □ événement sur «*entrée 2*» ⟹ «sortie 2» ;
- □ événement sur «*entrée 3*» ⟹ «sortie 3» et «sortie 4» ;

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
```

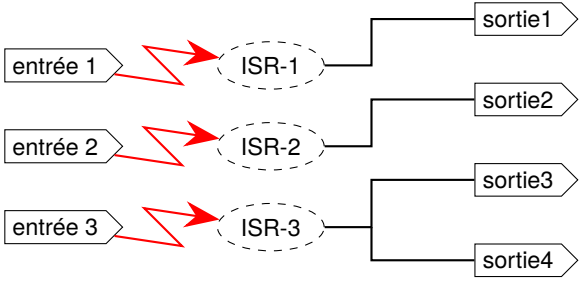
L'utilisation d'une alarme, `tmr.alarm` *donne du temps à l'OS.*

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.



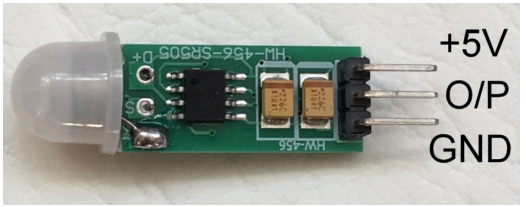
Ici, on a un traitement purement piloté par les interruptions :

- *__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

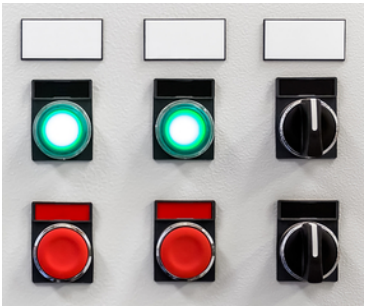


Le capteur PIR est associé à un circuit :

- ▷ 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 :

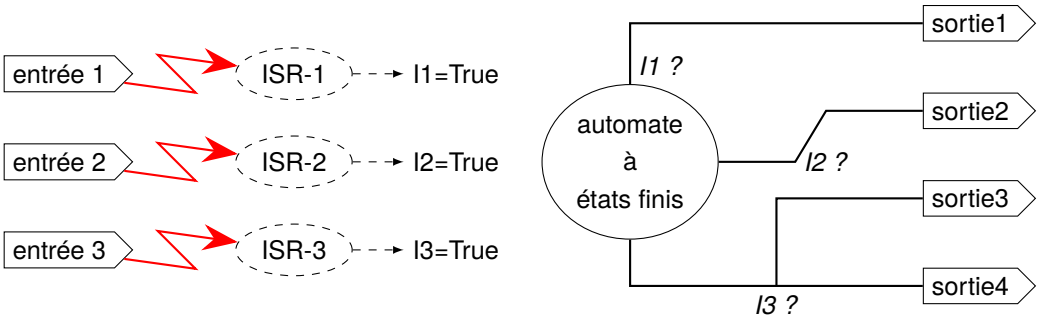
- ▷ associer une **entrée digitale** du micro-contrôleur à chaque bouton ;
- ▷ associer une **interruption** pour détecter la modification de chacun de ces boutons.

⟹ 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

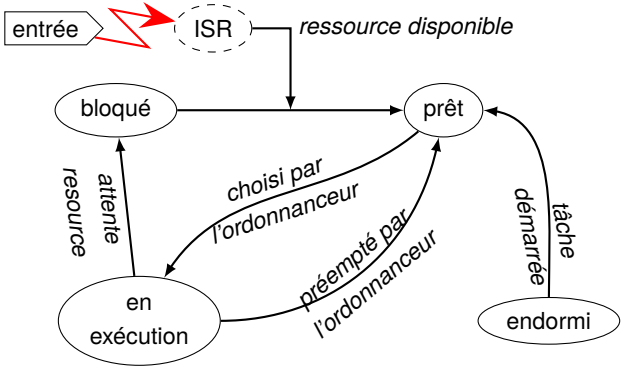


- ▷ *le traitement de l'interruption est rapide :*
  - ◇ *juste positionner un drapeau,* «flag»*, à vrai ;*
  - ◇ *faible latence pour le traitement des autres interruptions.*
- ▷ *c'est l'automate qui gère les priorités s'il y en a besoin.*

Attention

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

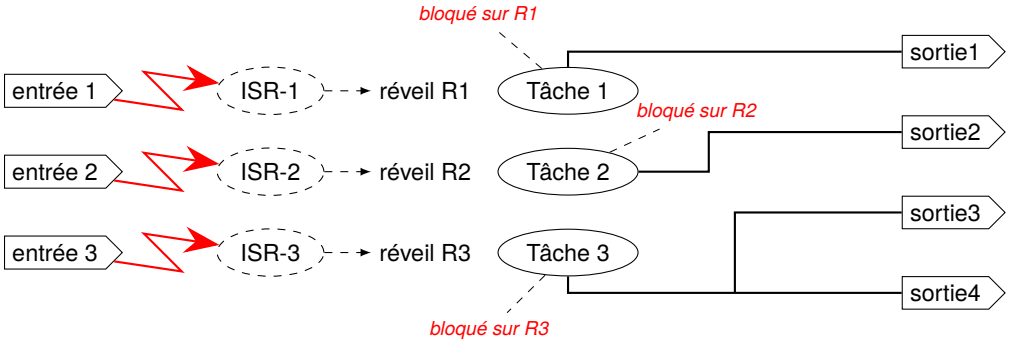
- ☐ «*level triggered*» : l'interruption se déclenche **tant que** le niveau (haut ou bas) sur la broche, ou «*pin*», est maintenu dans l'état associé à l'interruption $\Longrightarrow$ soit le matériel change le niveau au déclenchement de l'interruption, soit l'ISR doit changer le niveau, sinon l'interruption 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.

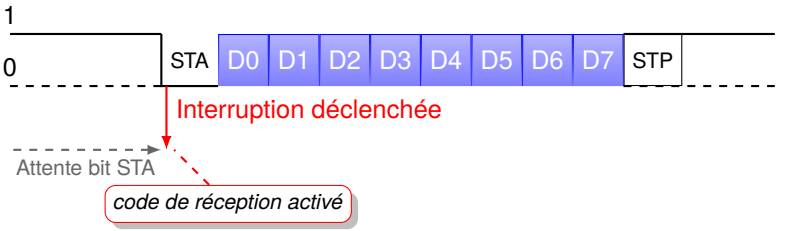
- ordonnanceur et préemption : si le noyau supporte la préemption, une tâche est **suspendue** après un «*timeslice*» : tous les registres sont sauvegardés et un **changement de contexte** est réalisé (dans le cas d'une interruption, on peut ne sauvegarder que les registres utilisés par l'ISR).

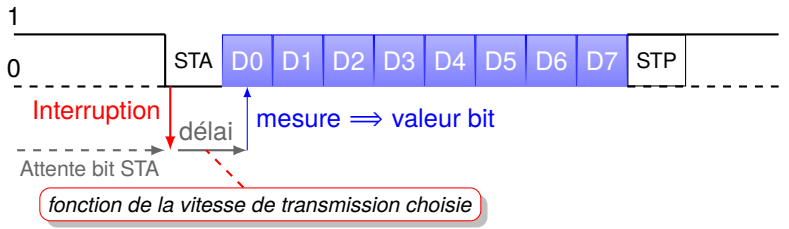


Chaque tâche est suspendue jusqu'à ce que le SE la réveille lorsque un événement se produit au travers d'une ISR.
Les tâches peuvent êtres synchronisées par des **sémaphores**, et communiquées entre elles par des «***message queues***».

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 :

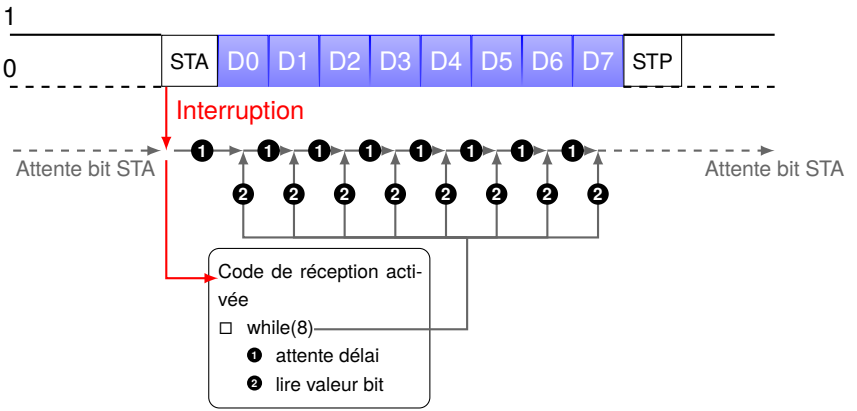




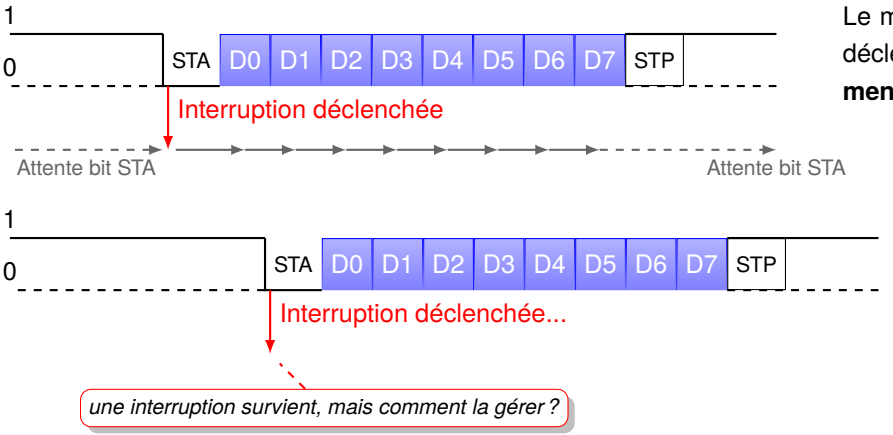
Le **code de réception** :

- ▷ 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

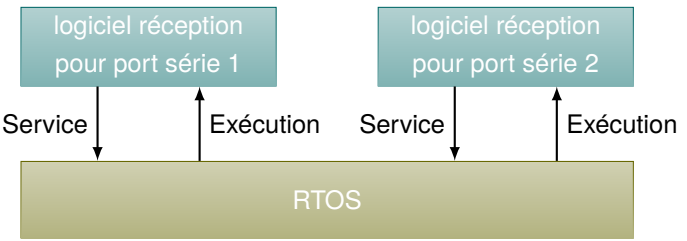


Le modèle de gestion basé sur les routines déclenchées par «*interruption*» est **difficilement extensible** !

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

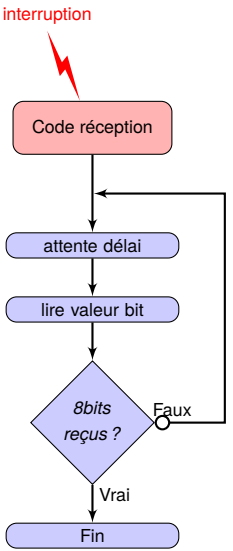
Avec RTOS, la configuration se simplifie :

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



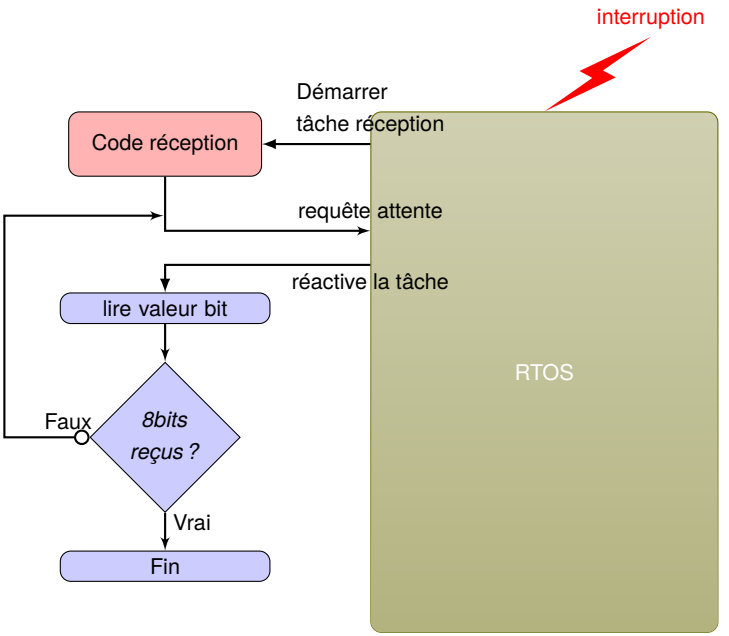
Le code de réception pour une communication asynchrone

Sans RTOS :



*Le délai est **interne** au code de réception.*

Avec RTOS :



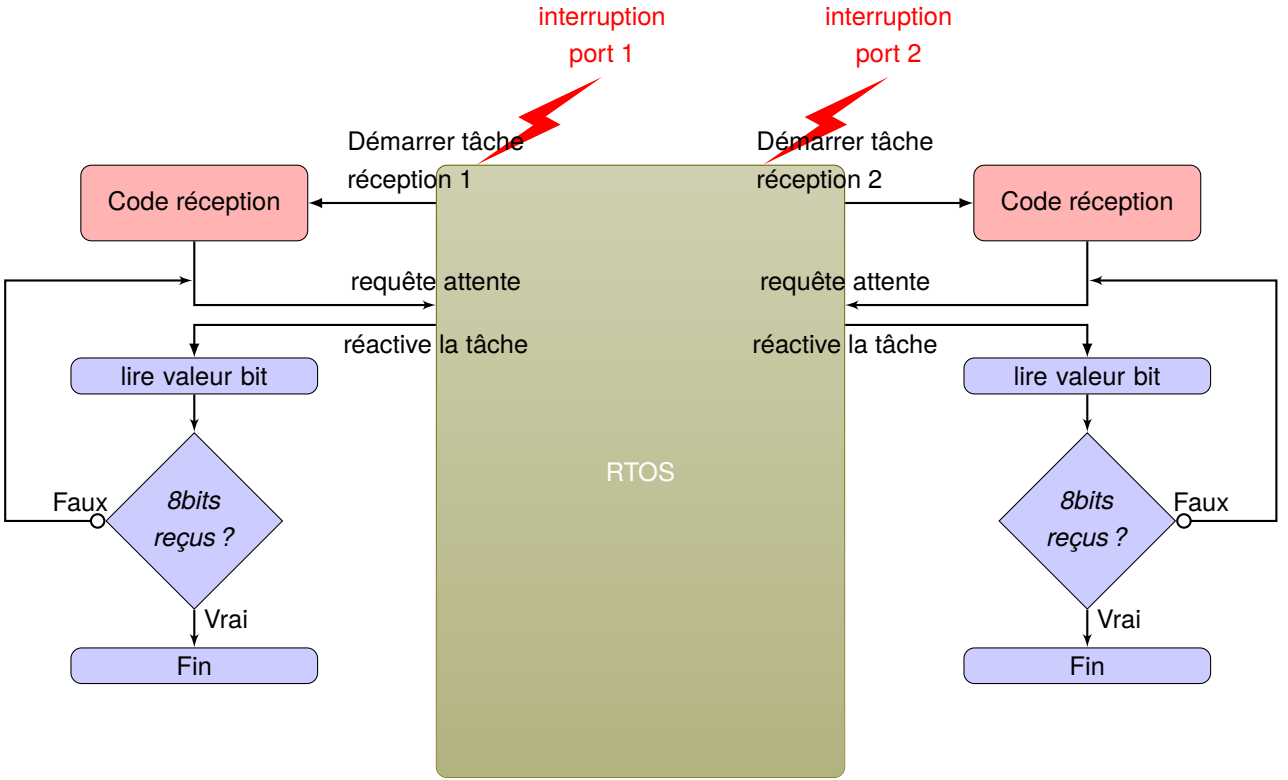
*Le délai d'attente de la tâche de réception est du **temps rendu** à RTOS.*

RTOS :

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

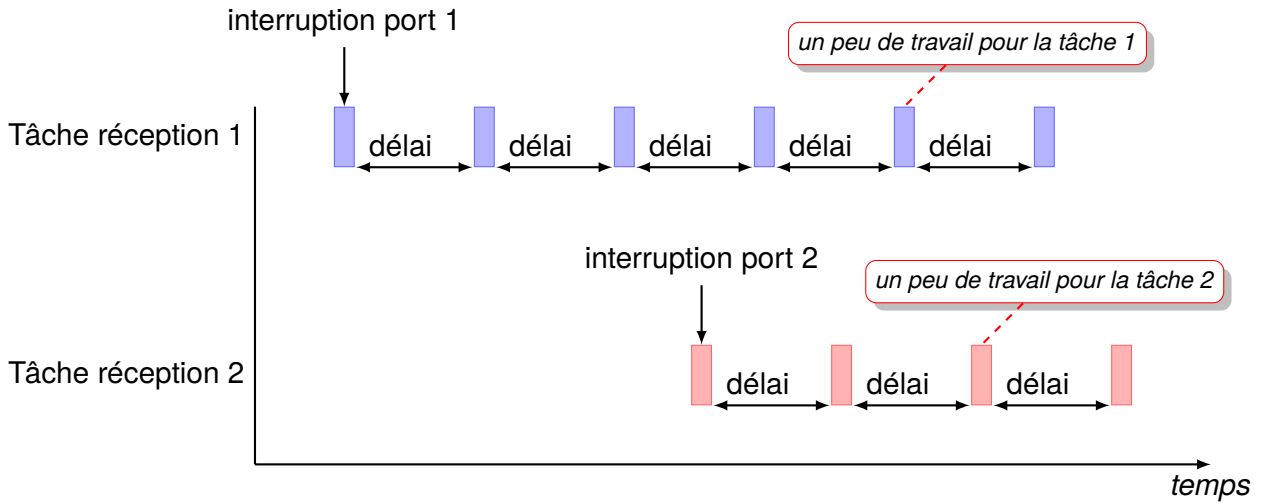
Gestion de deux canaux de réception asynchrone

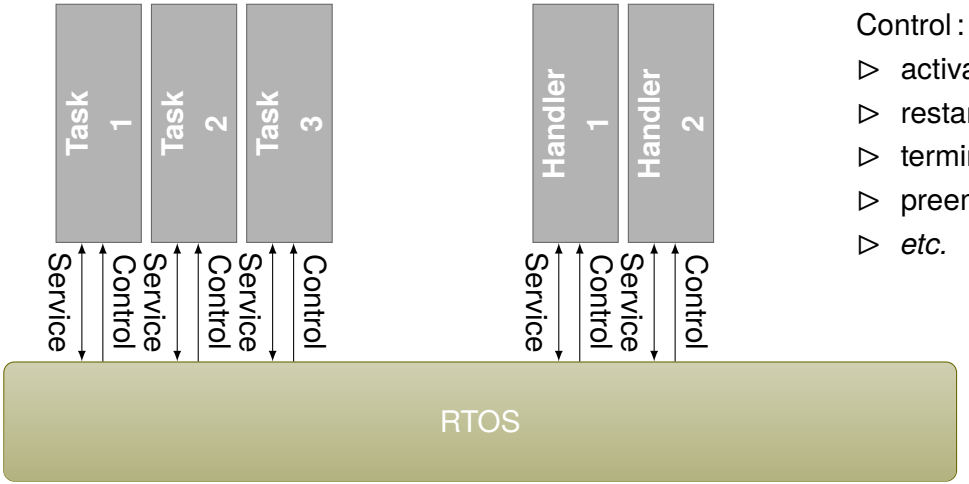
- On demande **l'allocation de deux tâches** de réception à RTOS ;
- RTOS **partage le temps** entre les deux tâches.



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.

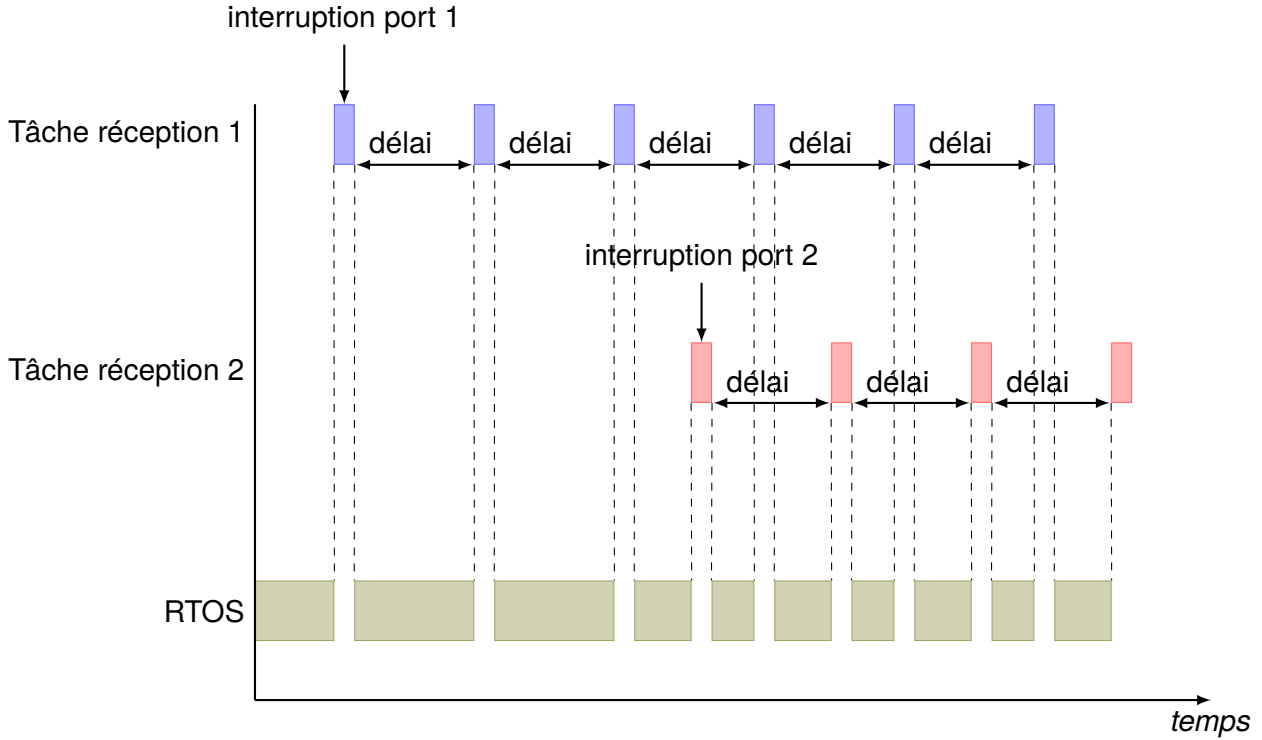




Control :
- ▷ activate,
- ▷ restart,
- ▷ terminate,
- ▷ preempt,
- ▷ *etc.*

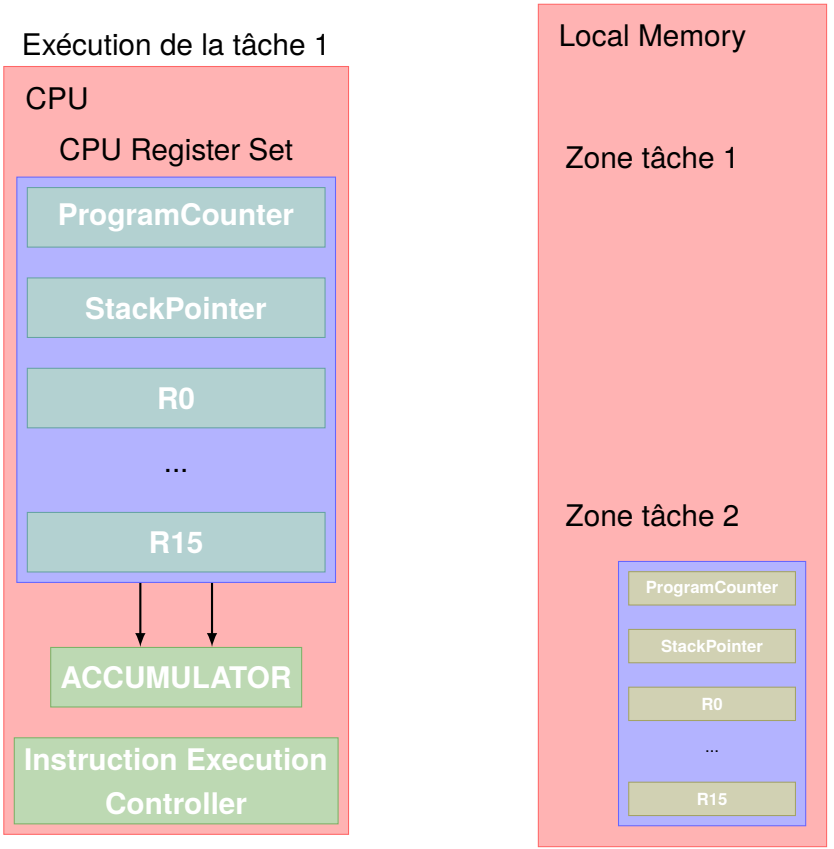
- □ **Task** : application gérée par RTOS :
  - ◇ RTOS en contrôle le lancement, la terminaison, *etc.* ;
  - ◇ une tâche peut également faire des appels systèmes vers RTOS, des APIs propres à RTOS ;
- □ **Handler** : code exécuté lors d'une interruption de l'exécution du code courant ;
  - ◇ examples : 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



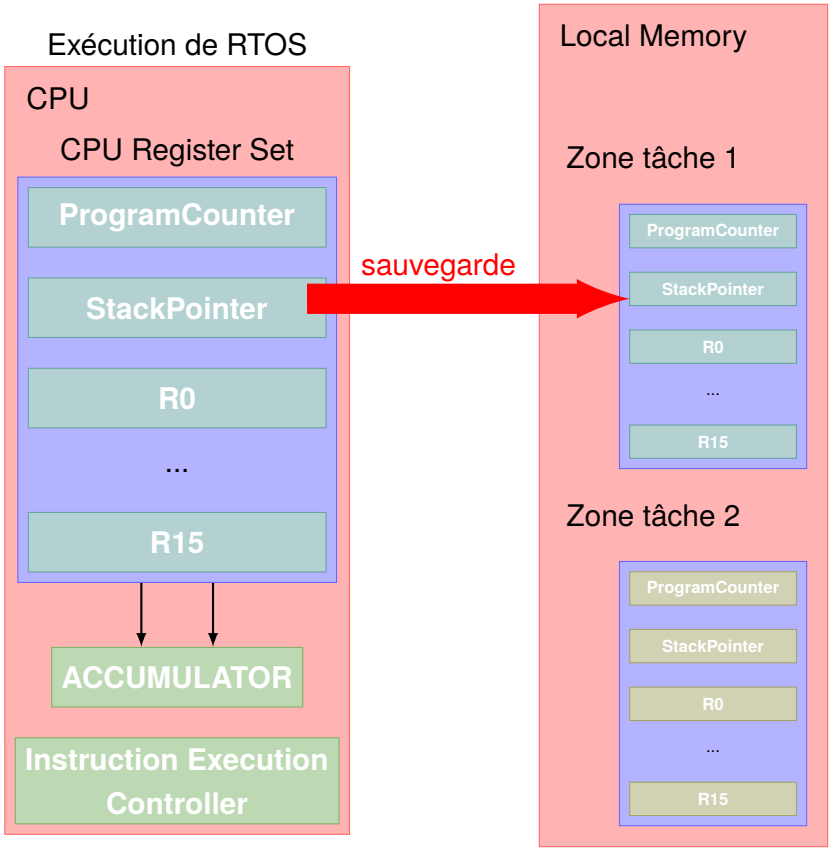
Lors d'un «*délai*», la tâche associée est «*suspendue*» et RTOS passe en mode «*Idle*» s'il n'a rien à faire ou peut réaliser du travail nécessaire au bon fonctionnement de l'OS.

Changement de tâche ou «*task switching*»



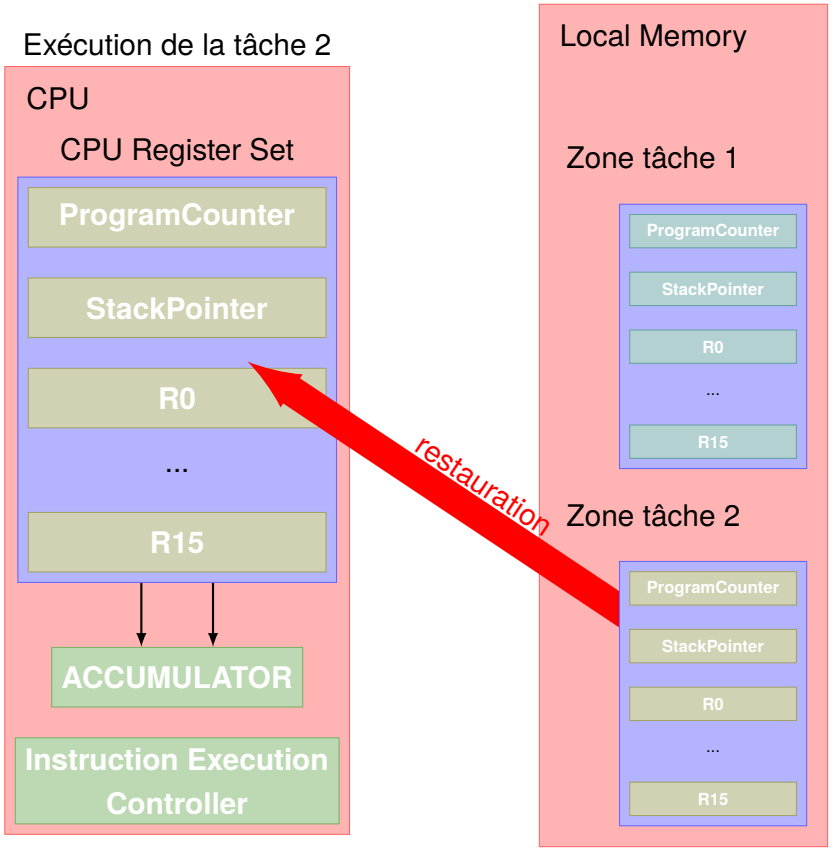
- La tâche 1 est en cours d'exécution sur le processeur ;
- on va passer à l'exécution de la tâche 2...

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.

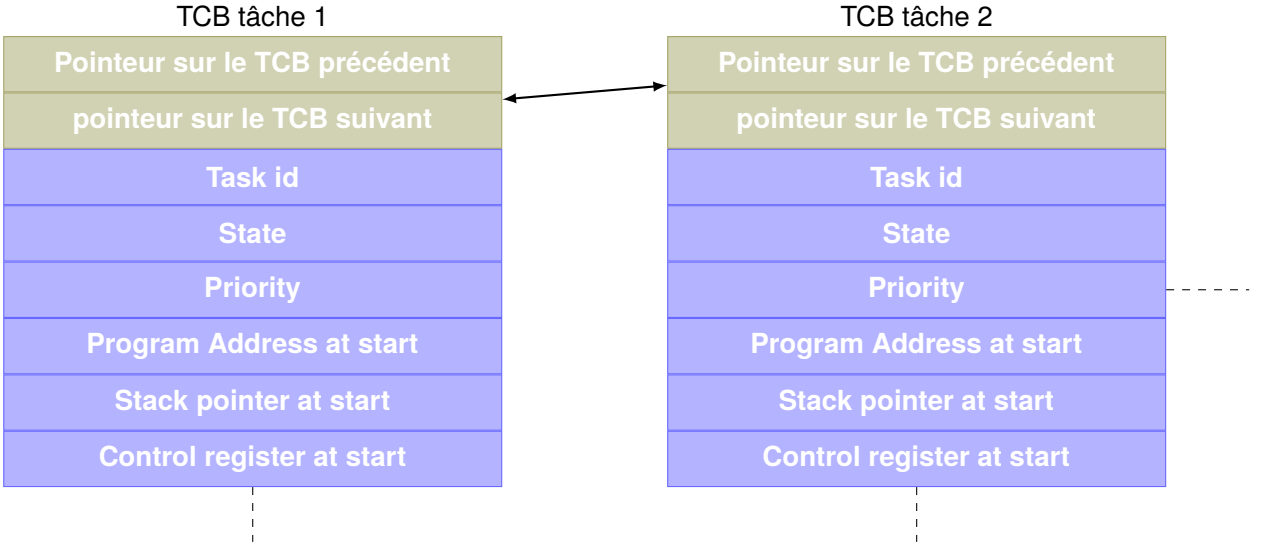
En particulier le registre «ProgramCounter» *pointe sur les instrutions de la tâche 2 à la place de celles de la tâche 1, une fois que RTOS a fini de réaliser l'échange.*

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 contexte $\Longleftrightarrow$ 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 :

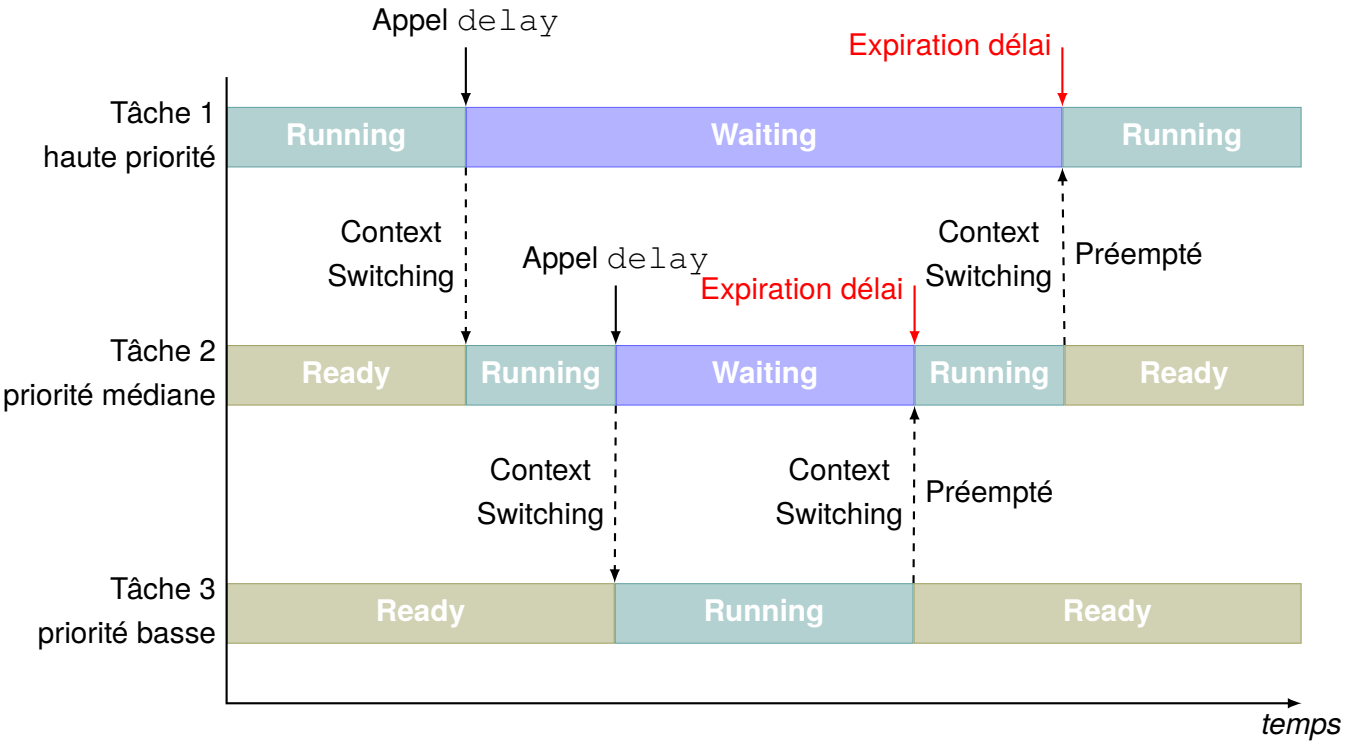
Prête, «*ready*» | **En exécution, «*running*»**

En attente, «*waiting*» | **dormant**

- ☐ «*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*» ;
  - ◇ seule la tâche de **plus haute priorité** est dans 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**.

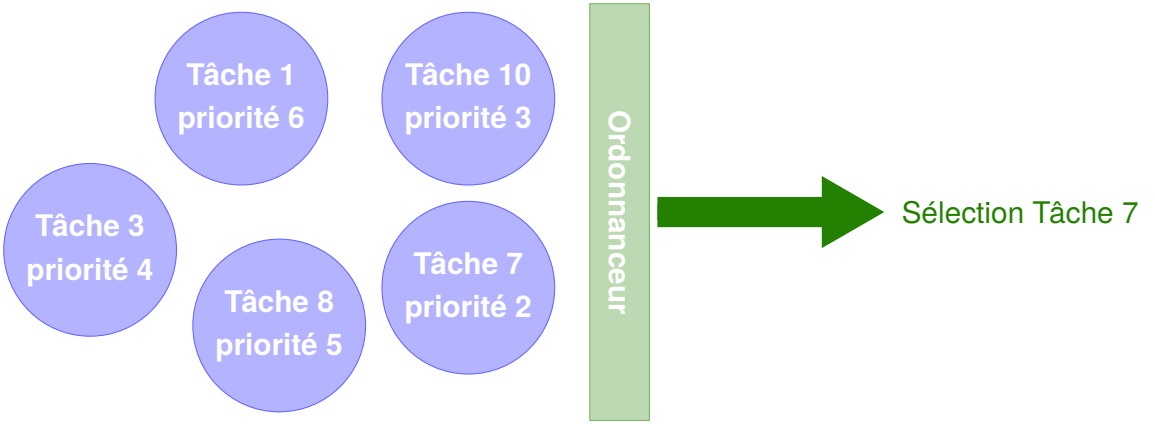
Changement de tâche et priorités

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



Ordonnancement, «*Scheduling*»

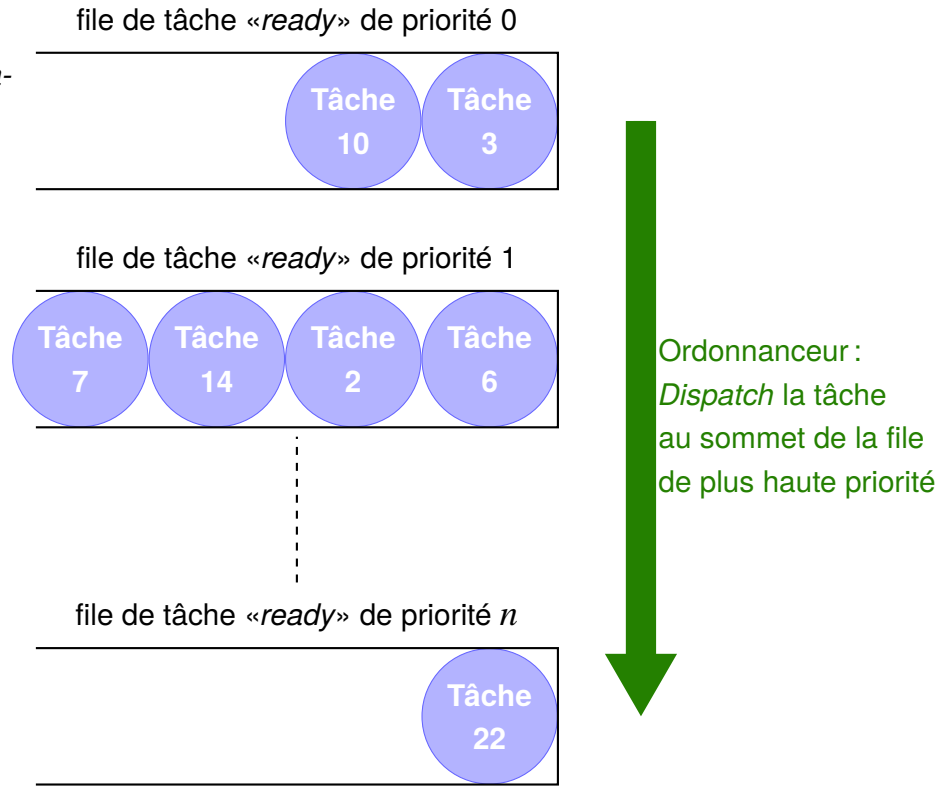
- chaque tâche possède une **priorité fixée lors de l'initialisation** ;
- RTOS **sélectionne** le tâche de **priorité la plus élevée** parmi celles en état «*ready*» ;



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

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*».



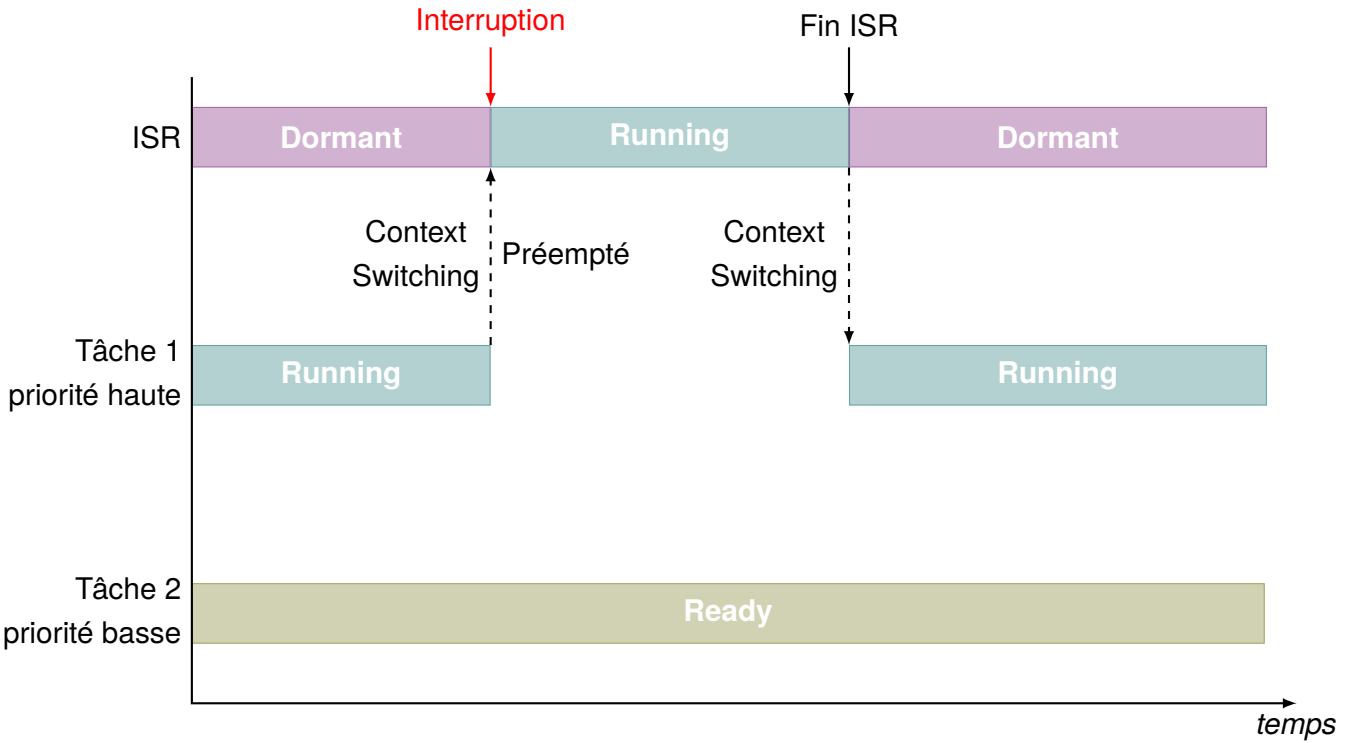
Retour sur les «*Handlers*»

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*, **supérieure** à la priorités des différentes tâches.
- Les **interruptions** sont, *en général*, **désactivées** pendant l'exécution d'un «*handler*».

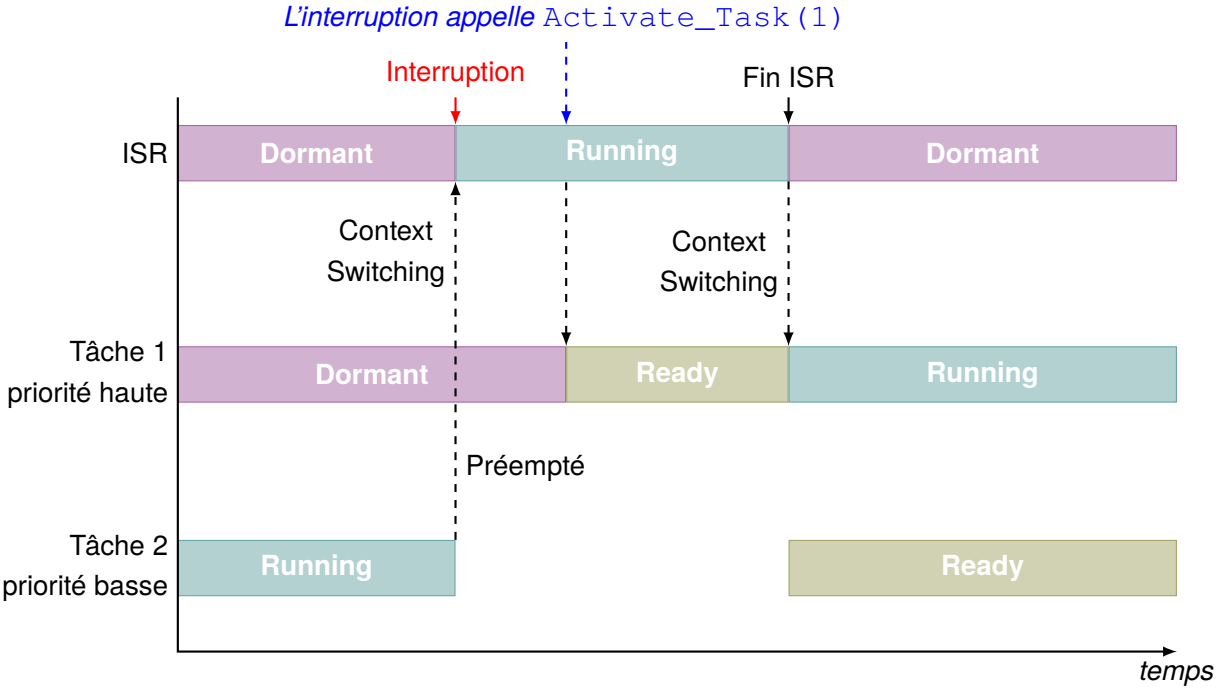
ISR, «*Interrupt Service Routine*»

- 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 ;

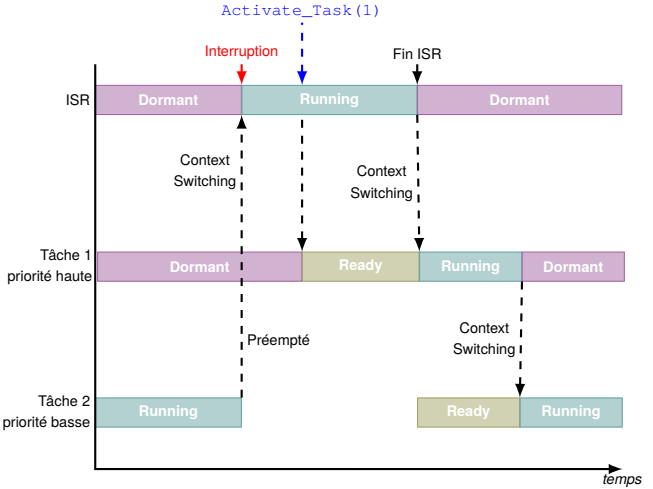


- ▷ L'ISR appelle l'API `Activate_Task(1)` qui passe la tâche 1 de l'état «*Dormant*» à «*Ready*» ;
- ▷ Lors de la fin de l'ISR, l'ordonnanceur choisit la tâche 1 qui est de **priorité supérieure** à tâche 2 qui passe en «*Ready*» ⟹ l'ISR «*communique*» vers la tâche 1.

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 :

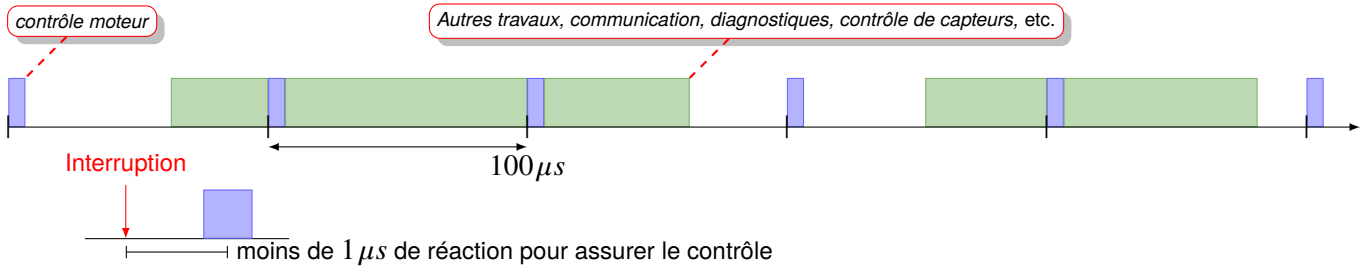


En effet, l'API `Activate_Task` *ne peut être appelée que depuis RTOS.*

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

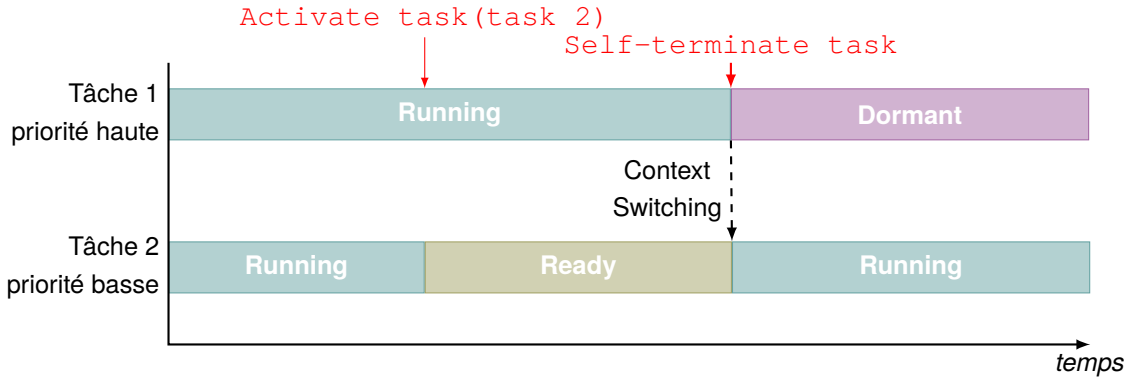


- ▷ on utilise un «*cyclic handler*» pour garantir que le contrôle s'effecture régulièrement : ici toutes les $100\,\mu s$;
- ▷ l'interruption doit être prise en compte dans un délai garanti : ici moins de $1\,\mu s$;
- ▷ entre le traitement de ces interruptions d'autres tâches peuvent être réalisées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

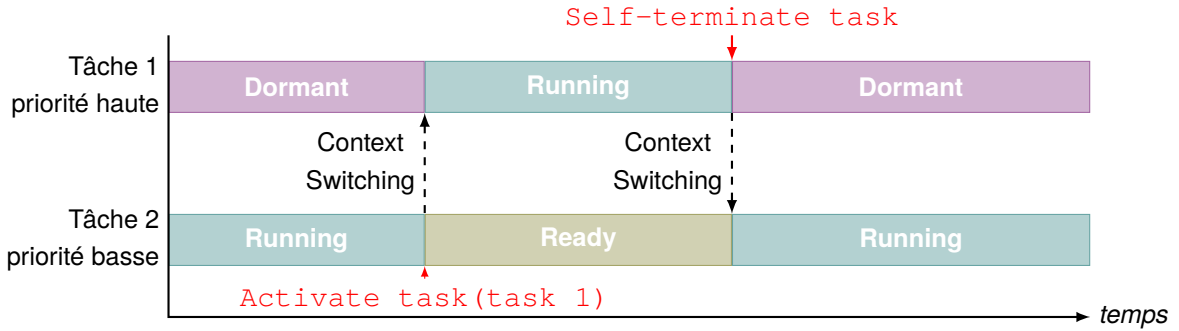
Le travail du «*cyclic handler*» est en relation directe avec des événements extérieurs/contrôleur (ici le moteur de la direction assistée du véhicule) qui sont «*temps réel*».

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.*

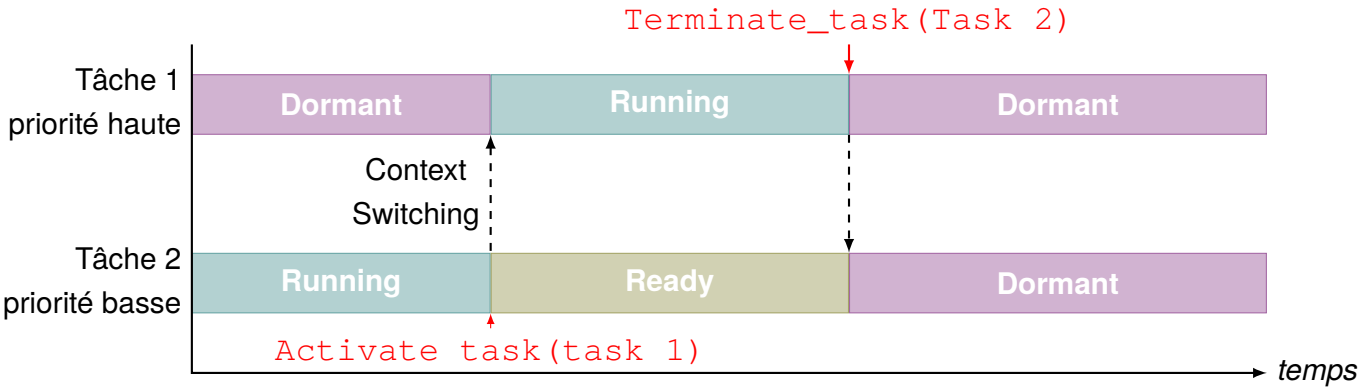
- `Activate_task(Task ID)`
  - RTOS change l'état de la tâche indiquée en argument de «*Dormant*» à «*Ready*» ;
- `Self-terminate_task`
  - RTOS change l'état de la tâche qui l'appelle de l'état «*Running*» à «*Dormant*» ;



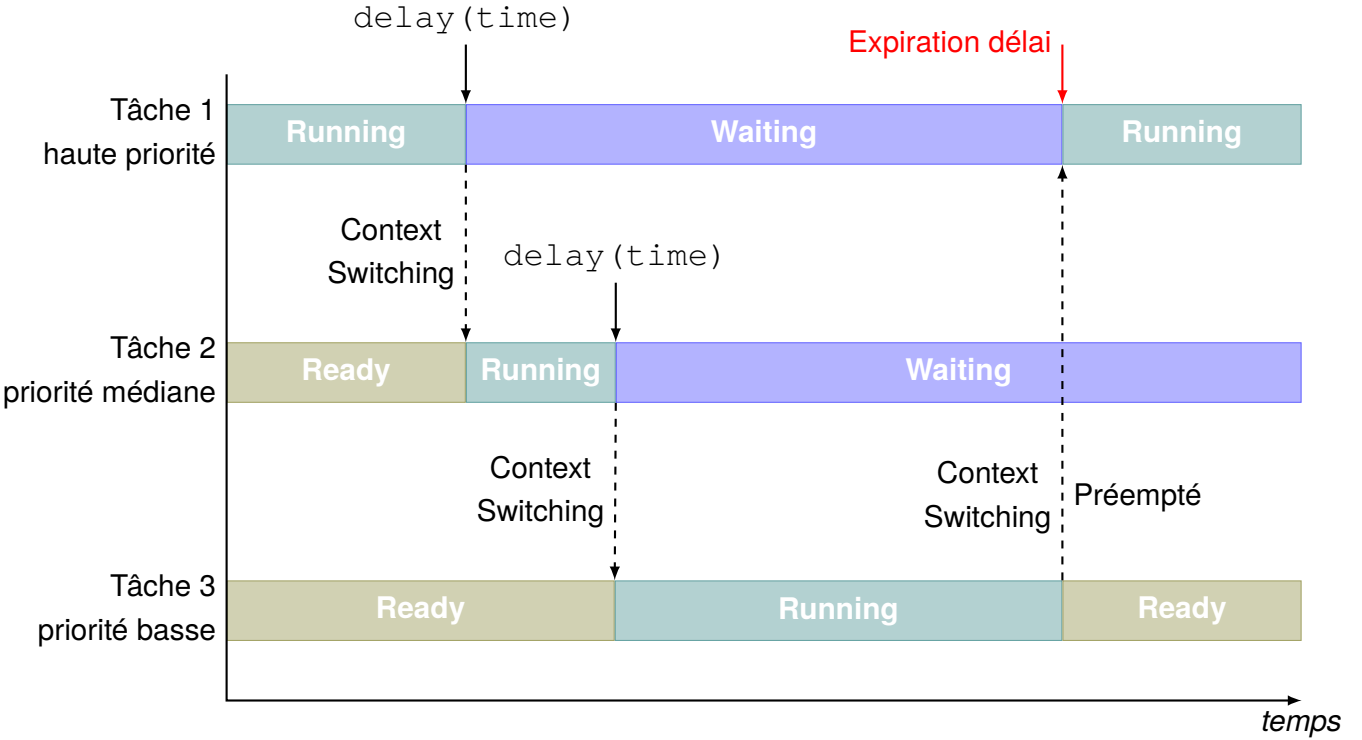
Exemple où une tâche de priorité inférieure active une tâche de priorité supérieure



- `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

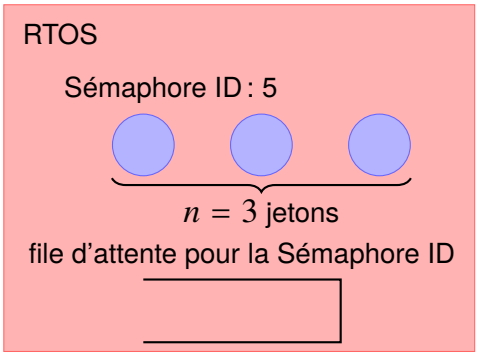
- ▷ fournit un contrôle :
  - ◇ acquérir une sémaphore pour autoriser :
    - ⋆ des opérations ;
    - ⋆ un accès ;
    - ⋆ *etc.*

    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*».

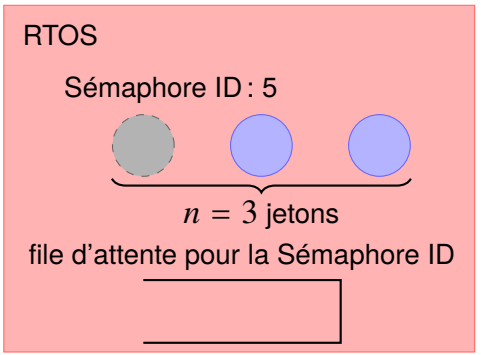
    RTOS change également l'état de cette tâche vers «*Ready*».

Sémaphore avec compteur



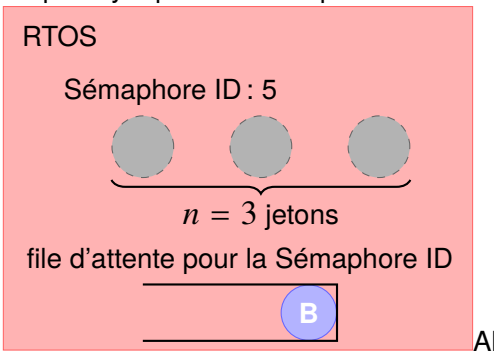
- le nombre de sémaphore est n ;
- les autres conditions sont les mêmes que pour les sémaphores binaires :
  - ◇ il existe plusieurs sémaphores pouvant être récupérées par plusieurs tâches ;
  - ◇ une sémaphore peut être libérer par une tâche que ne l'a pas acquise à l'origine ;
  - ◇ une ISR peut libérer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

Si une tâche appelle `Acquire_semaphore(5)` :



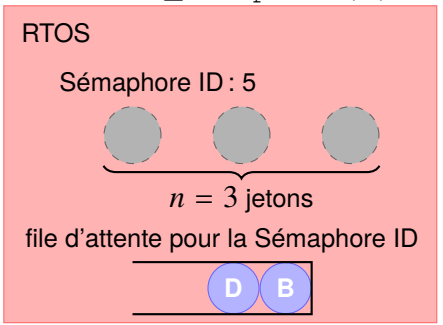
Un jeton/sémaphore est retiré/donné à la tâche.

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



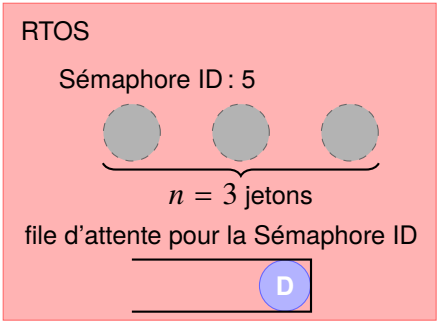
Alors la tâche B est mise dans la file d'attente $\Longrightarrow$ elle est en état «*Waiting*».

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

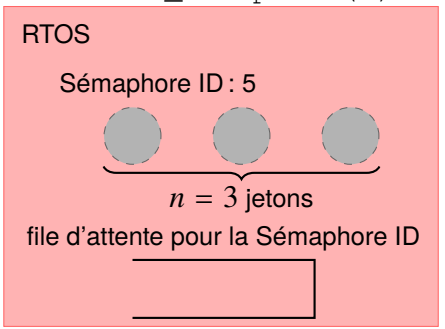


⟹RTOS va :

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

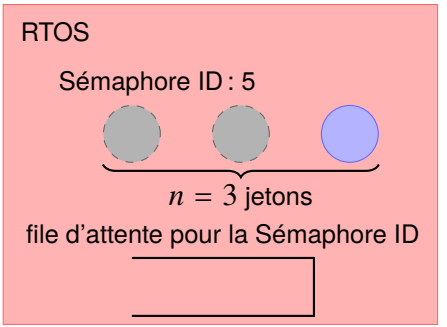


- ▷ une tâche va libérer la sémaphore avec `Release_sémaphore(5)` :



⟹RTOS va :

- ▷ augmenter le compteur de la sémaphore.

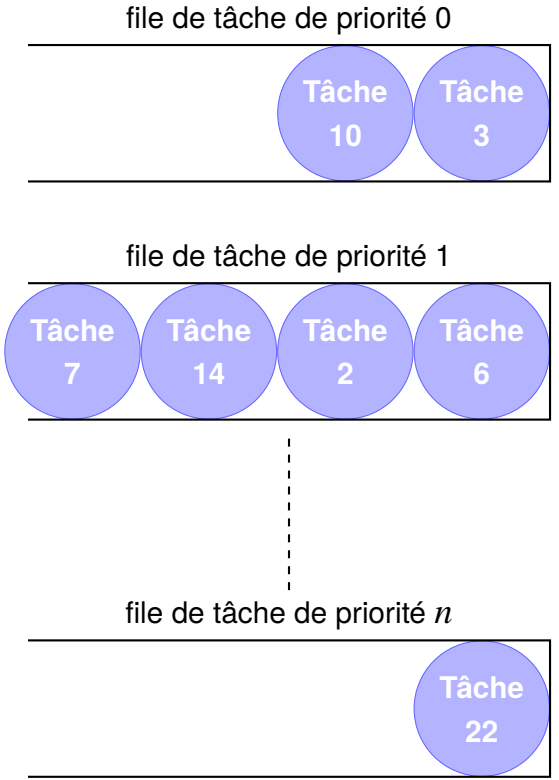


Il n'y a pas de tâches dans la file d'attente...

File d'attente de Sémaphore

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

File d'attente de la sémaphore ID=0



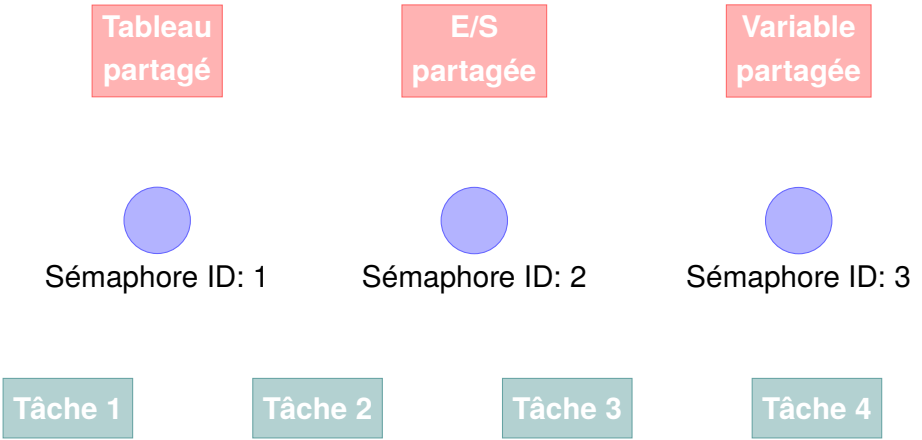
Chaque sémaphore a le même nombre de files que de priorités :

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.

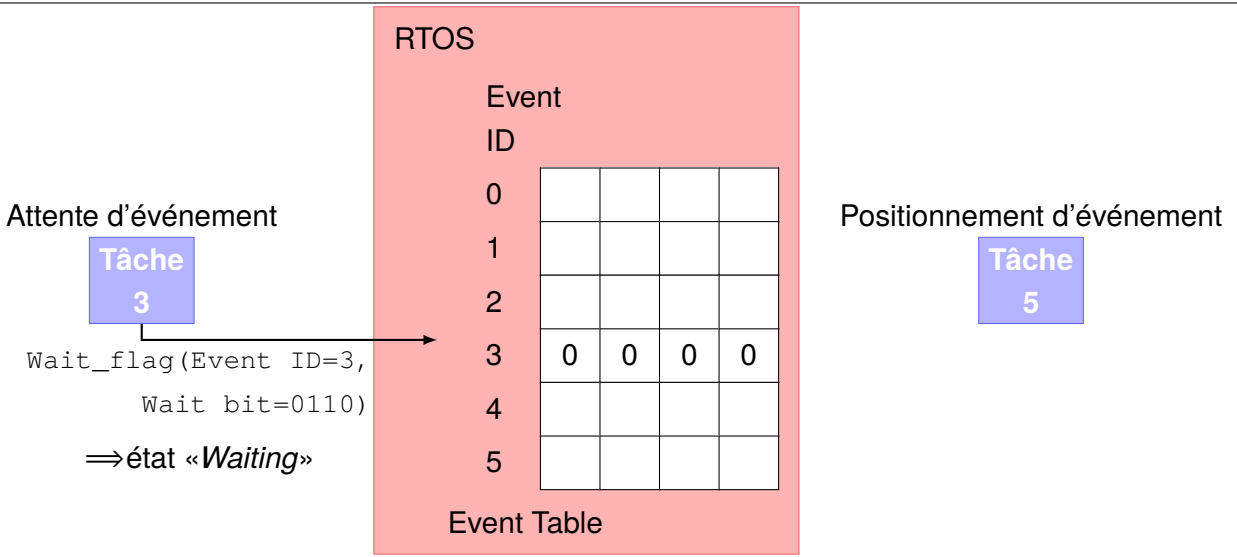
Ordonnanceur :
FCFS basé sur la priorité
la tâche au sommet de la file
de plus haute priorité est libérée
de l'état «*Waiting*»

Accès synchronisé à des ressources partagées

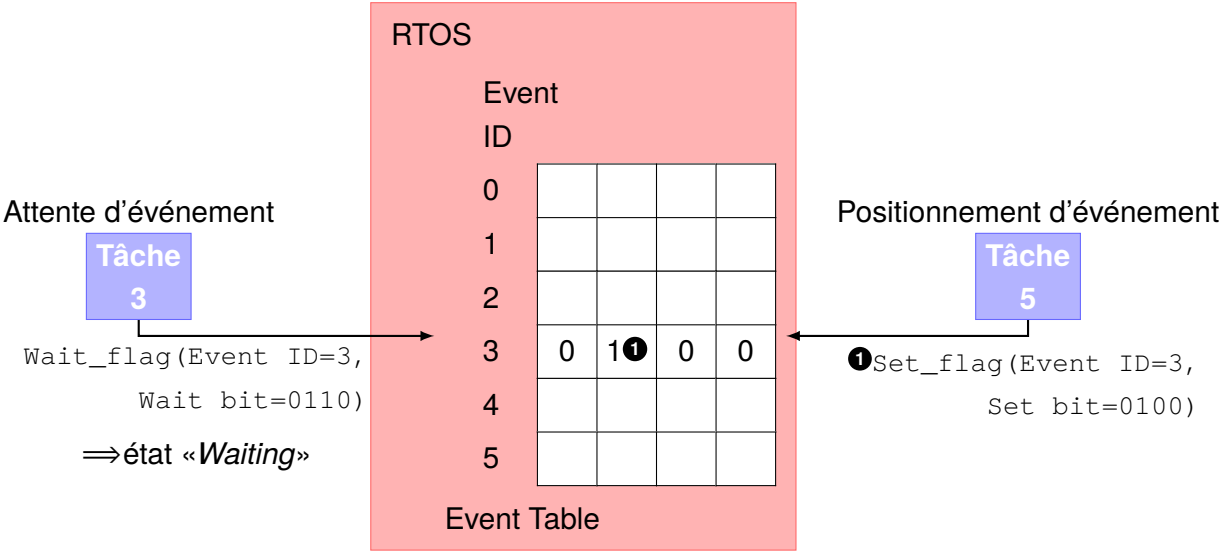


- «*raise a flag*» pour envoyer un signal à une autre tâche :
  - communication entre tâches ;
  - synchronisation entre tâches ;
- positionne ou lit un drapeau (bit) dans un registre particulier de la table «*Event*».
- `Wait_flag(Event ID, Flag pattern:`
  - si le «*flag pattern*» indiqué en argument **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

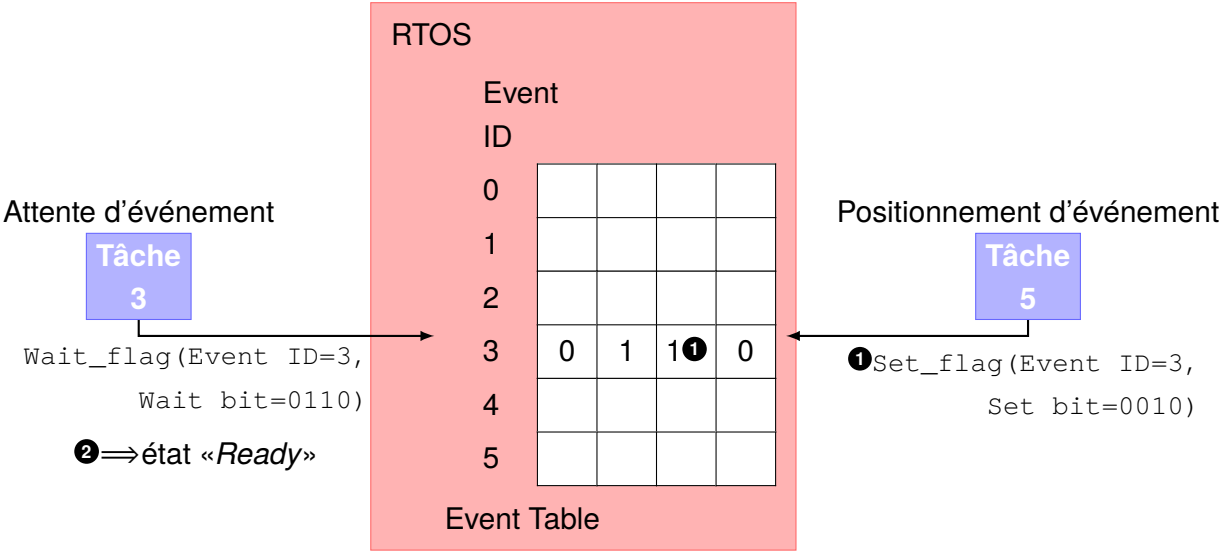


- la tâche 3 appelle `Wait_flag(Event ID=3, Wait bit=0110);`
- l'entrée 3 de la table d'événement ne correspond pas au motif demandé
  ⟹RTOS passe la tâche 3 en état «*Waiting*».



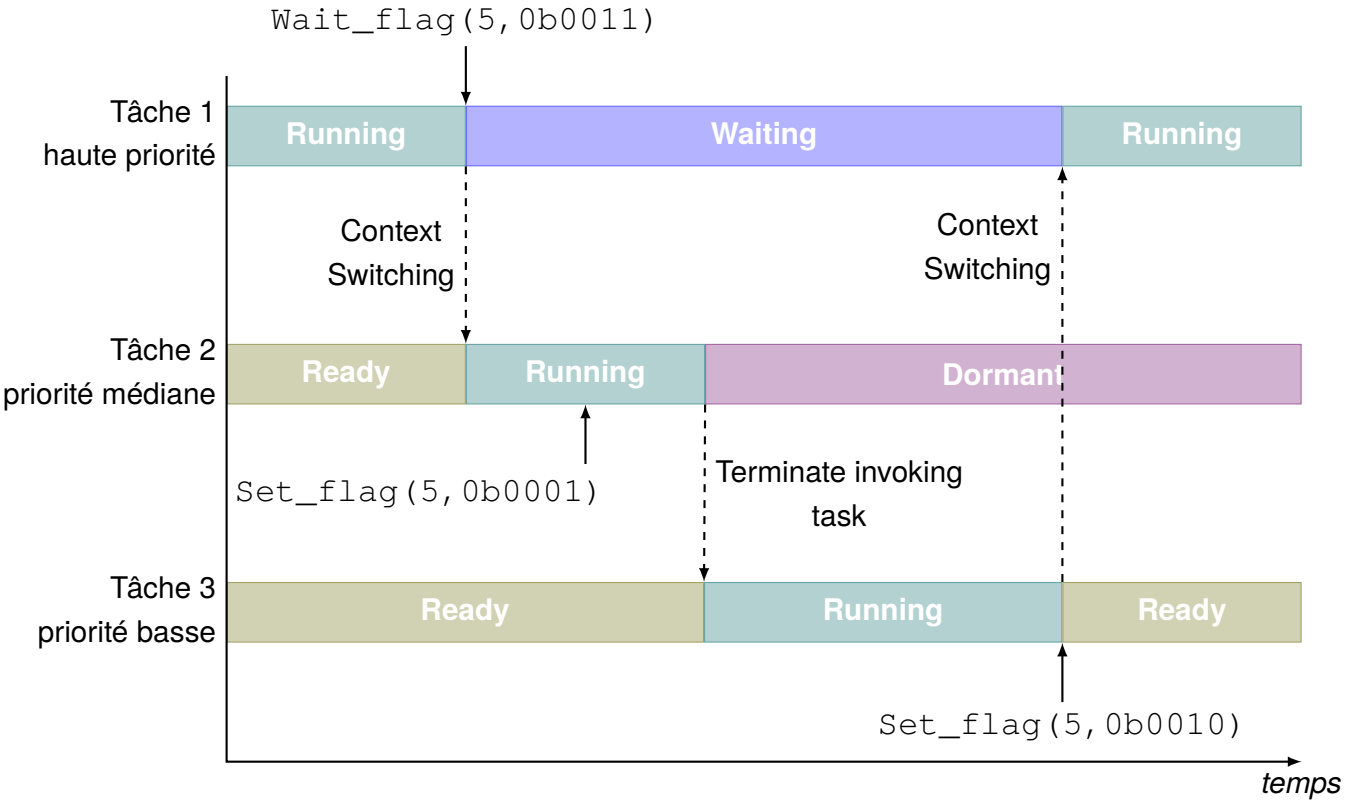
- ▷ la tâche 5 appelle `Set_flag(Event ID=3, Set bit=0100)`❶;
- ▷ l'entrée 3 de la table d'événement devient `0100`;
- ▷ *ce motif ne correspond pas au motif attendu par la tâche 3*

⟹ la tâche 3 reste dans l'état «*Waiting*».



- ▷ 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

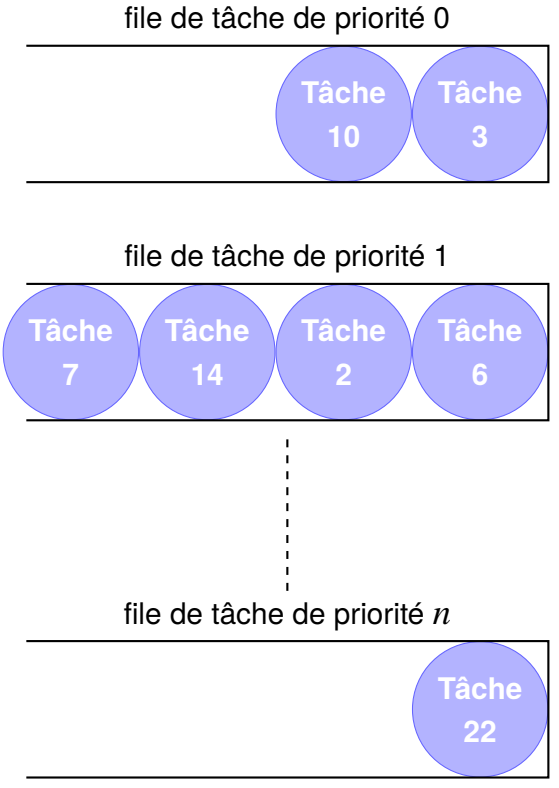
Les tâches 3 et 5 se sont **synchronisées** sur la construction d'un motif.


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 «*flag 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*» ⟹ 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 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 code 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_semaphore` :
  - 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*» :

    $\Longrightarrow$*l'essai d'acquisition d'une sémaphore n'est pas bloquante*.

- 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*».
      Enfin, 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*».

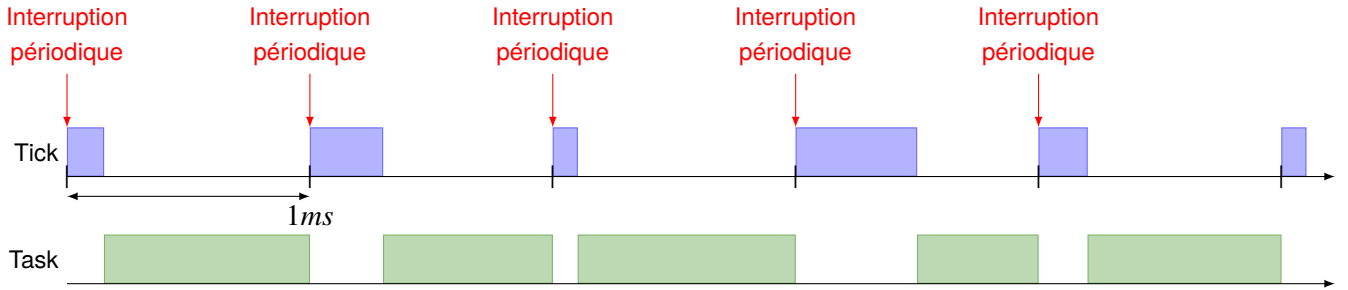
Attention

- 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** ↗.



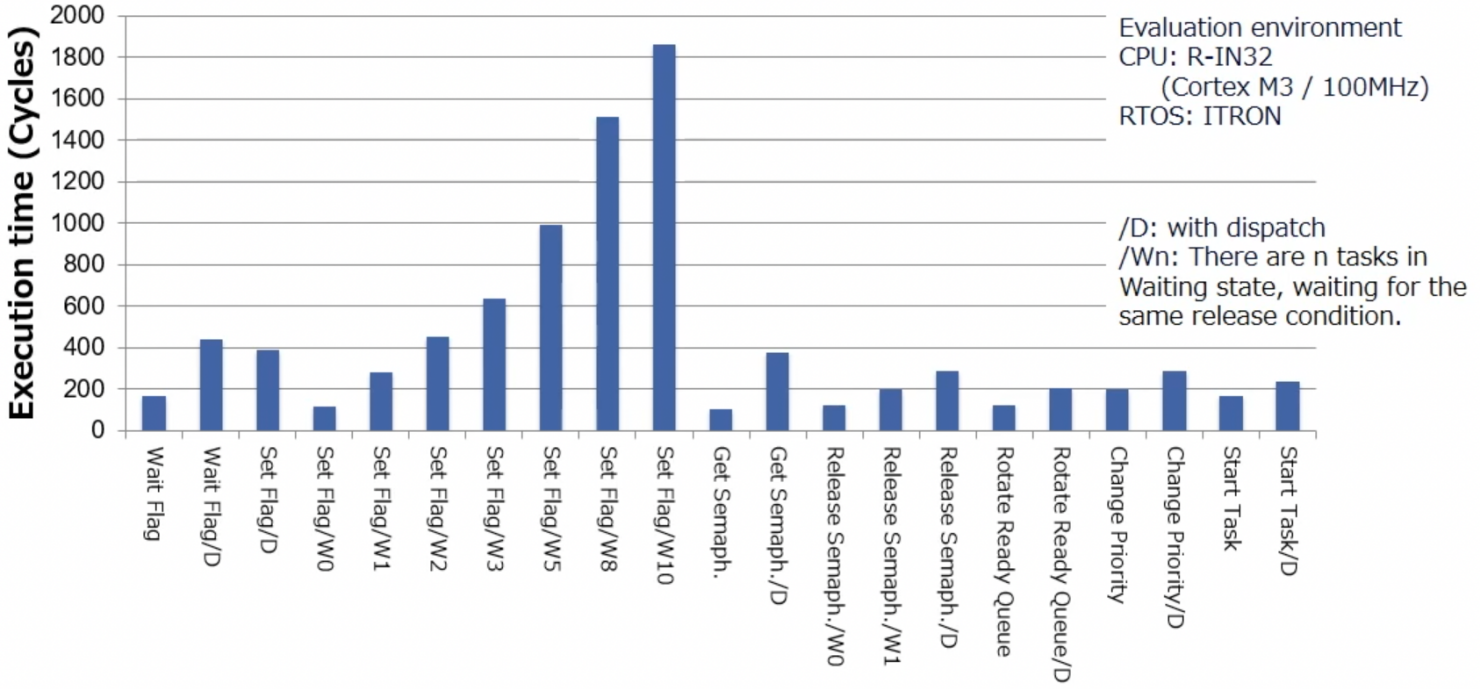
Du point de vue des tâches :

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

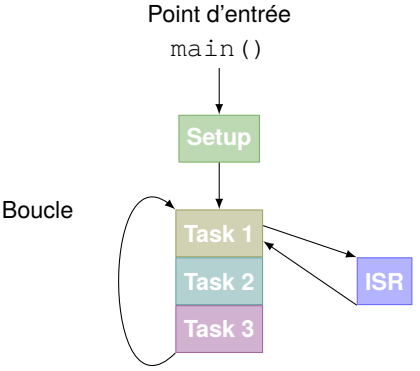


L'unité de temps est le nombre de **cycle processeur** : une instruction prend en général de 100 à cycles du CPU.

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*»



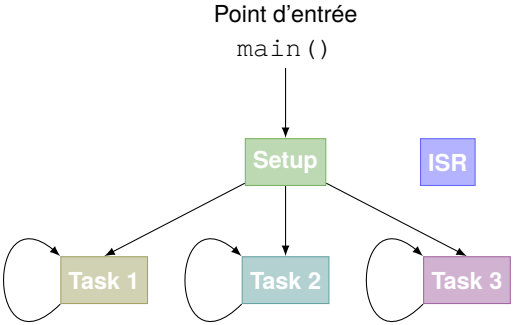
Avantages :

- 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 faire 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

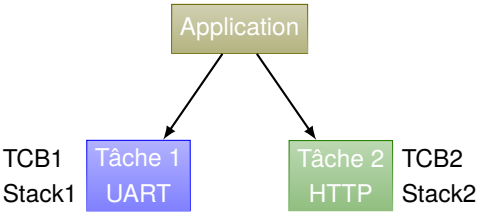
Developed in partnership with the world's leading chip companies over a 15-year period, and now downloaded every 175 seconds, FreeRTOS is a market-leading real-time operating system (RTOS) for microcontrollers and small microprocessors. Distributed freely under the MIT open source license, FreeRTOS includes 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 :

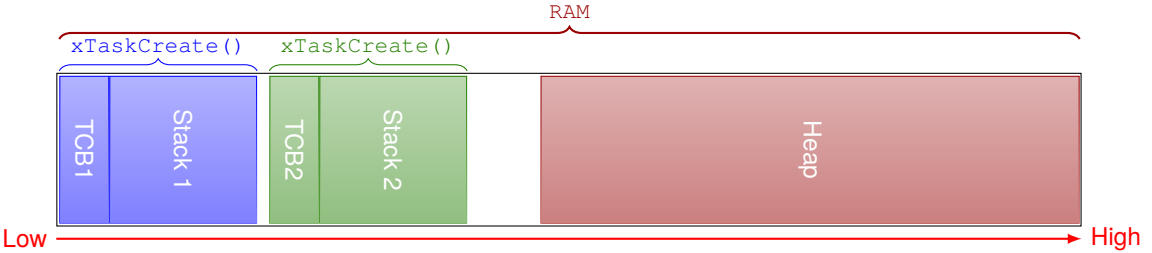
- est gérée par un TCB, «*Task Control Block*» ;
- dispose d'une pile, «*stack*», pour ses variables et ses appels de fonction.




```
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.

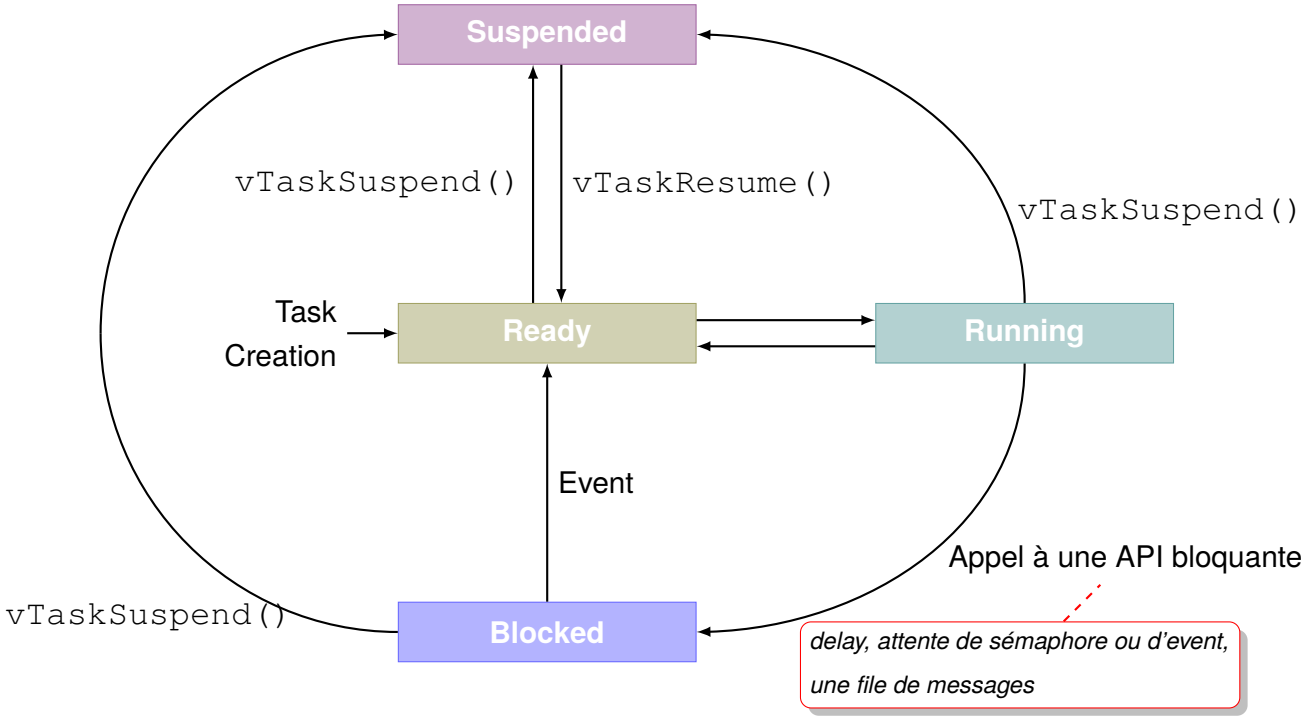
⟹ un **arrêt de l'application** survient si la tâche **dépasse** cette taille de pile allouée.



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

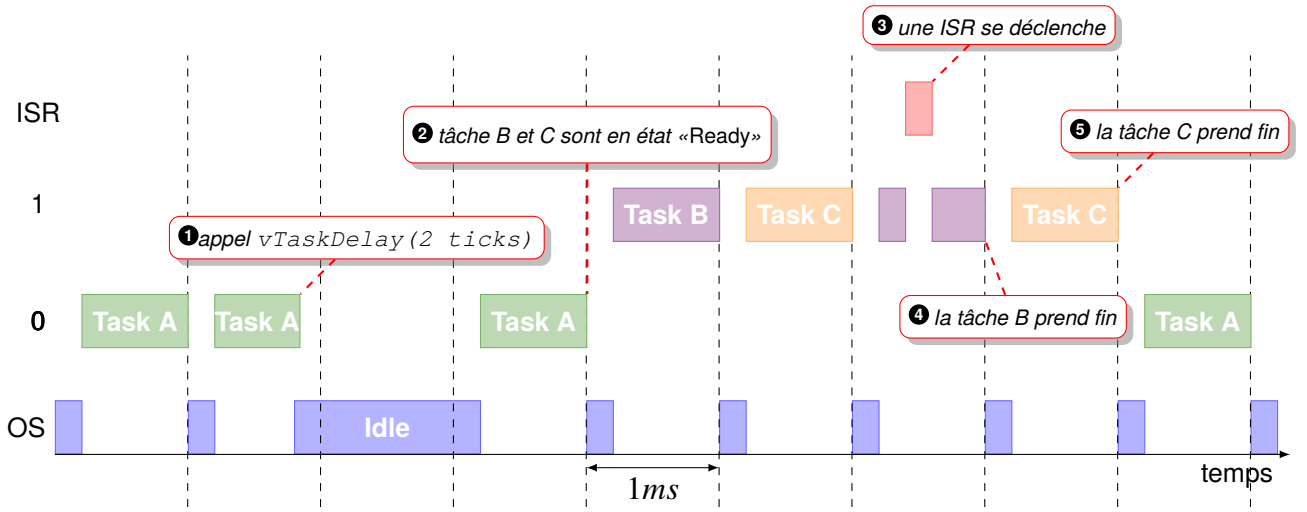
- une allouée aux différentes **tâches/piles** ;
- la seconde affectée au **tas**, «*heap*» de l'application complète.

Les différents états d'une tâche en FreeRTOS

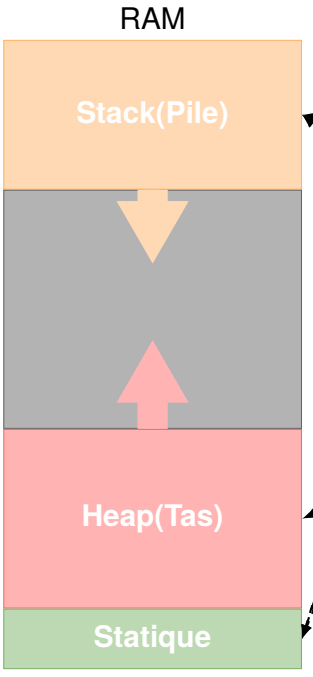


Par rapport au RTOS générique

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



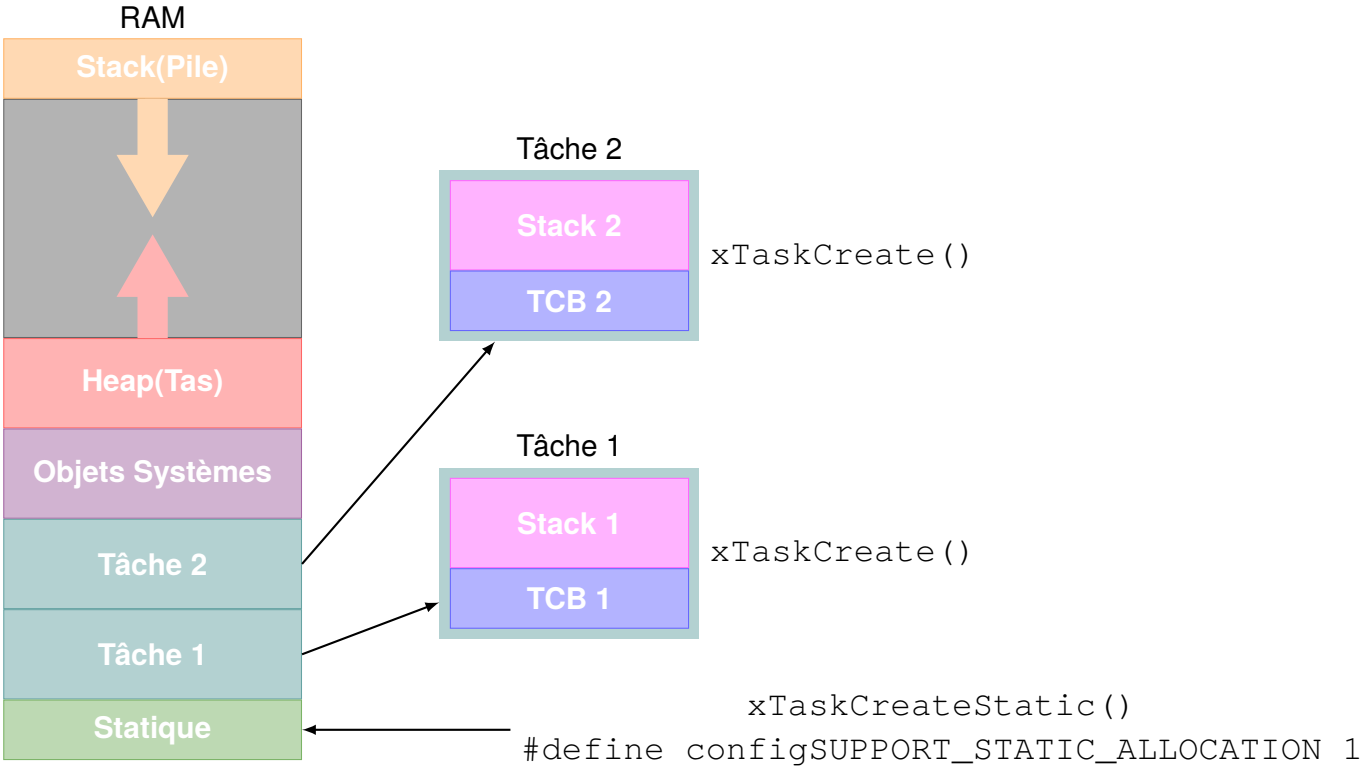
- le temps est découpé en «*slice*» de $1ms$: un «*tick*» ;
- à chaque «*slice*», l'OS tourne pour sélectionner la prochaine tâche à exécuter⟹ordonnanceur :
  - il choisit la tâche de plus haute priorité : «*Task A*» est de faible priorité mais elle est seule ;
  - en ❶, la tâche se met en attente et 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.




```
int variable_globale;
static int variable_statique;

void une_fonction(int un_argument) {
  static int variable_static_locale;
  int variable_locale;
  int *pointeur;

  pointeur = (int *)malloc(32 * sizeof(int));
  // Travail
  free(pointeur);
}
```

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


```
// 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
  xTaskCreate(
              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


  // Task to run forever
  xTaskCreate(
              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

  // 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


```
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);
  }
}

// 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);
  }
}
```

Exemple de lancement de tâches avec des priorités différentes


```
// setup correspond à sa propre tâche de priorité 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("priorite :");
  Serial.println(uxTaskPriorityGet(NULL));

  // lancement des taches
  xTaskCreate(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

  xTaskCreate(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


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

Obtenir la priorité de la tâche

Exemple de lancement de tâches avec des priorités différentes

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

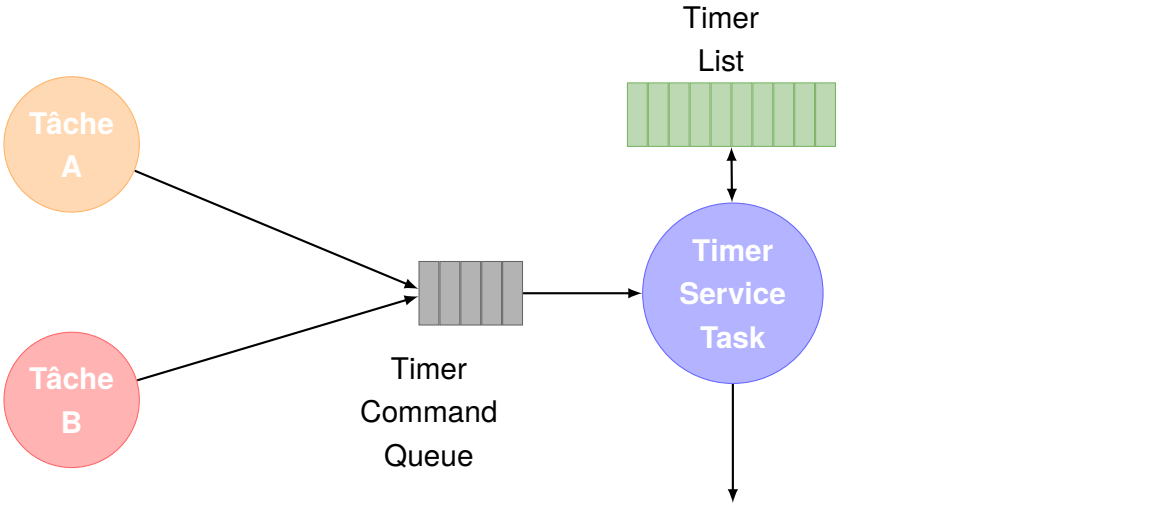
```
void loop() {
  // Suspension de la tache de plus haute priorité de temps en temps
  for(int i=0; i < 6; i++) {
    vTaskSuspend(task2);
    vTaskDelay(2000 / portTICK_PERIOD_MS);
    vTaskResume(task2);
    vTaskDelay(2000 / portTICK_PERIOD_MS);
  }
  // Détruire la tache de plus basse priorité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

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);
}
```

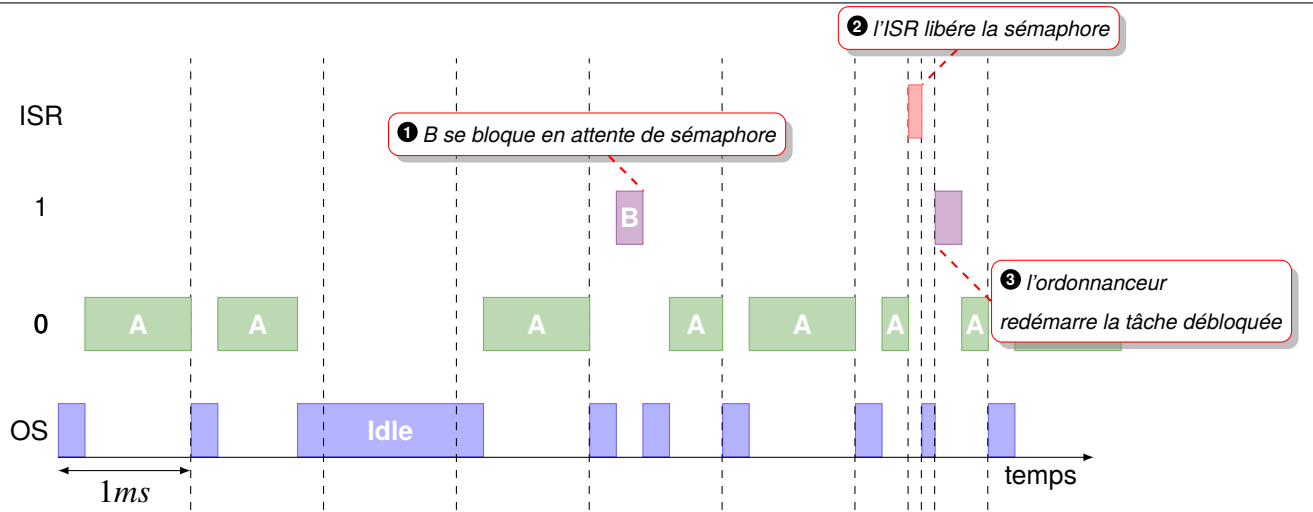
Pour l'utilisation des Files de messages

```
#include <queue.h>

static QueueHandle_t file_messages;

// Lecture du message en mode bloquant
xQueueReceive(file_messages, (void *)&element, portMAX_DELAY);

// 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) { // Une tâche a été réveillée
    portYIELD_FROM_ISR();
  }
}
```

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

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

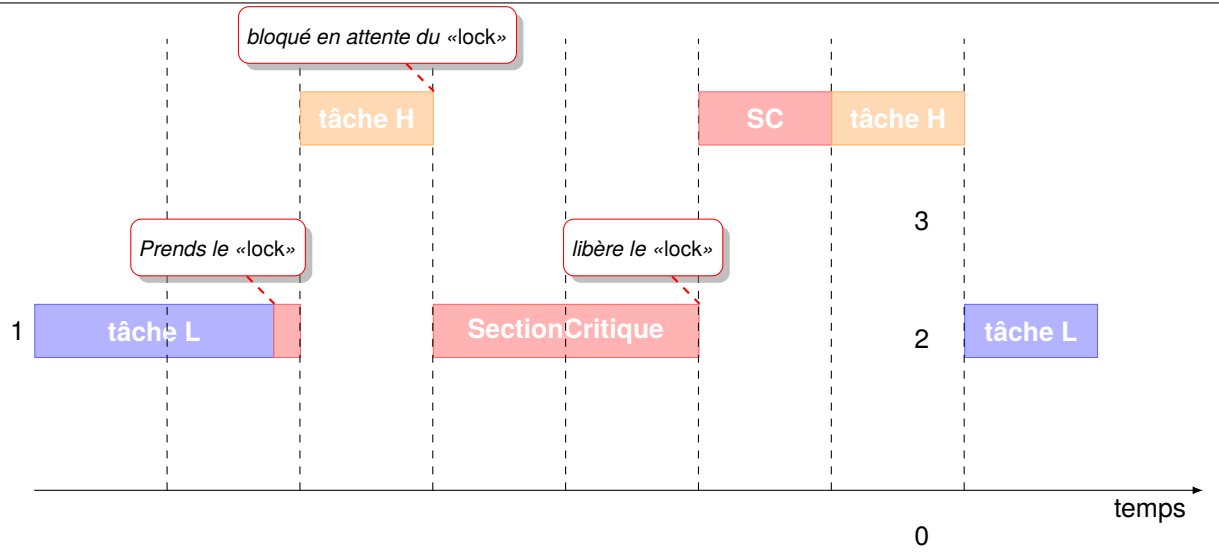
Installation du «*timer*»

```
// Création/démarrage du timer (num, divider, countUp)
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);
```

Déclenchement de la tâche en priorité 2 :

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

Inversion de priorités



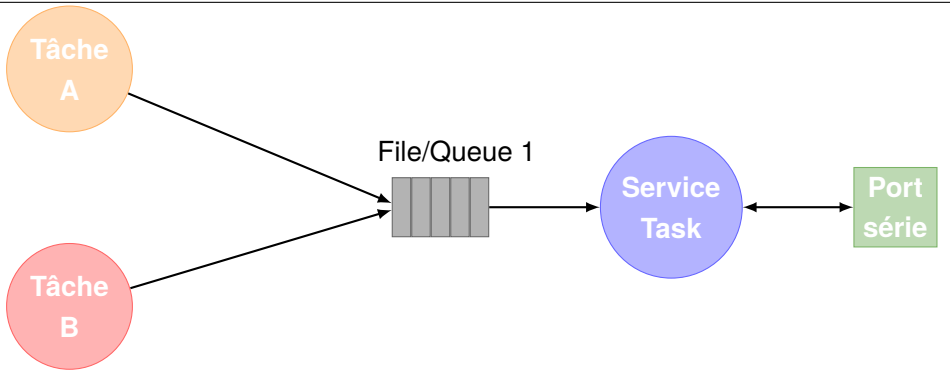
- le «*lock*» : peut être une sémaphore ou un mutex ;
- la tâche L de basse priorité prends le «*lock*» ;
- elle est interrompue par la tâche H de plus haute priorité ;
- à un moment donné la tâche H essaye d'entrer dans la même «*section critique*», mais elle ne peut obtenir le «*lock*» qui est détenu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

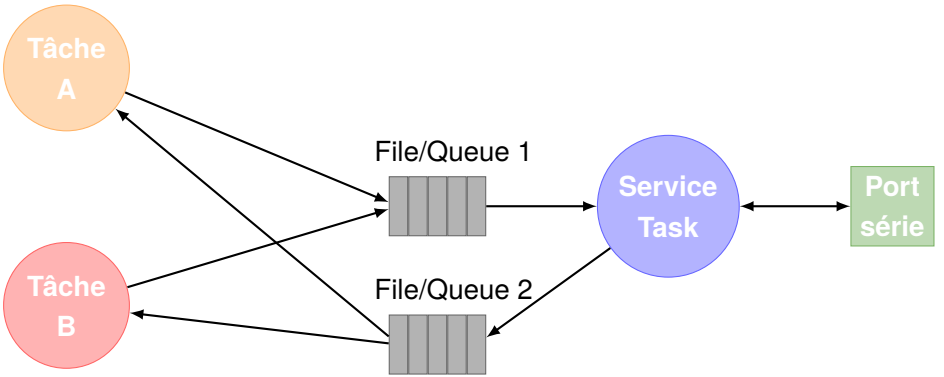
Le temps d'attente de la tâche H est contraint par le temps d'exécution de la tâche L dans la «*section critique*».

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

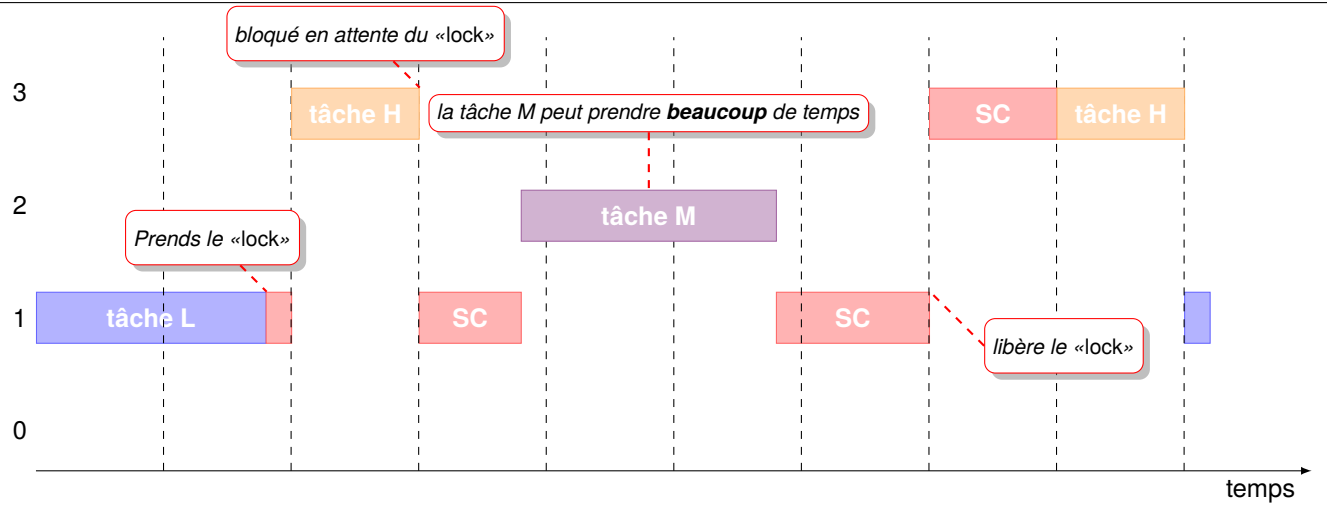
Solution : utiliser une «*tâche de service*»



- 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.

Nouvelle méthode plus rapide et prenant moins de place mémoire qu'une sémaphore binaire.

- `ulTaskNotifyTake()` endort la tâche jusqu'à la réception d'une notification ou jusqu'à l'expiration d'un «*timeout*» ;
- `xTaskNotifyGive()` débloque une tâche ;
- `vTaskNotifyGiveFromISR()` débloque une tâche depuis une ISR.

Identification de la tâche à notifier

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        unsigned short usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        ❶TaskHandle_t *pxCreatedTask );
```

Lors de la création de la tâche :
❶⟹ utilisé pour identifier la tâche à laquelle envoyer la notification.

Exemple d'utilisation

```
void tache(__unused void const* argument)
{
  static uint32_t notification;

  while(1)
  {
    notification = ulTaskNotifyTake(pdTRUE,
                        portMAX_DELAY);

    if(notification)
    {
      associated_work();
    }
  }
}
```

```
void system_power_interrupt_handler(uint32_t time)
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

...

    // Notify the task so it will wake up
    // when the ISR is complete
    vTaskNotifyGiveFromISR(powerTaskHandle,
                           &xHigherPriorityTaskWoken);
    ❷portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

❷⟹ indique au «*scheduler*» si une tâche de plus haute priorité a été réveillée et que le scheduler doit faire un changement de contexte directement vers cette tâche.

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.

Et le «*bare metal*» ?

Qu'est-ce que fait l'OS ?

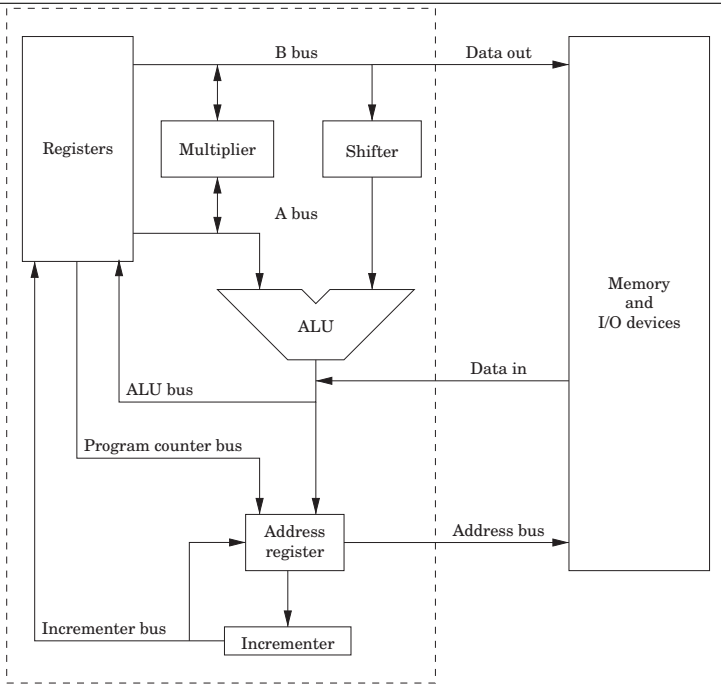
- ☐ initialise la **pile**, «*stack*» ;
- ☐ charge le segment `.text` de **code** ;
- ☐ charge et initialise le segment de `data` pour les **données** ;
- ☐ initialise les **variables statiques** ;
- ☐ fournit une interface **d'entrée** et de **sortie** ;
- ☐ fournit au programmeur une **vue abstraite** de la machine :
  - ◇ lorsque le programmeur accède au **disque dur**, il utilise l'abstraction du «*fichier*» :
    - ⋆ le système sous-jacent ne connait que des «*blocs de données*» ;
    - ⋆ l'OS fournit les structures de données et les opérations qui permettent au programmeur de penser en terme de fichiers et de flux d'octets ;
  - ◇ le programmeur voit la mémoire comme une seule **zone continue** ;
    - ⋆ le programme peut être dispersé en mémoire physique :
    - ⋆ le composant matériel MMU, «*Management Memory Unit*» piloté par l'OS qui donne cette vue simplifiée de la mémoire ;
  - ◇ le programmeur utilise des **appels systèmes** pour accéder à ces abstractions fournies par l'OS.

Qu'est-ce fait le «*bare metal*» ?

En «*bare metal*», il n'y a **pas d'abstractions** à moins que le programmeur ne les crée !

Il existe néanmoins des bibliothèques comme `newlib` qui fournit des services similaires à la bibliothèque standard du C, pour faciliter le portage d'une application en embarqué :

- ▷ soit le matériel est **déjà géré**, et le programmeur n'a rien à faire de plus ;
- ▷ soit le programmeur doit implémenter en ASM/C des **fonctions de bas niveaux** pour porter la bibliothèque.



- ☐ **deux registres** peuvent servir de **source** pour une instruction en passant par les bus A et B ;
- ☐ les données sur le bus B passent par un «*shifter*» : on peut décaler la seconde opérande avant qu'elle atteigne l'ALU ;
- ☐ les bus A et B peuvent fournir des **opérandes** pour le «*multiplier*» et le «*multiplier*» peut fournir des données pour les bs A et B ;
- ☐ les données en **lecture** depuis la mémoire ou des I/Os peuvent aller directement dans l'ALU puis dans un registre.
- ☐ les données en **écriture** vers la mémoire ou dans les I/Os sont prises directement dans le bus B, qui peuvent provenir de regsitres, mais ces données ne peuvent être modifiées sur le chemin.

☐ le **registre d'adresse** est un registre temporaire utilisé à chaque opération de lecture/écriture mémoire/I/Os. Peut être chargé :
- ◇ depuis le «*program counter*» pour **chercher**, «*fetch*», la **prochaine instruction** ;
- ◇ depuis l'ALU pour permettre des **modes d'adressages** où un registre est utilisé comme **adresse de base** et un **décalage** est calculé à la volée.
  Après l'accès, **l'adresse de base** peut être **incrémentée** et cette valeur **stockée** dans un registre ;
  ⟹utilisé pour **incrémenter** le «*program counter*» à chaque instruction ;
  ⟹utilisé pour certains mode d'adressage où un pointeur est **incrémenté** à chaque accès mémoire.

Le processeur dispose de 16 registres

- **R0** à **R12** : 13 registres à usage générique utilisable comme on le veut ;
- **R13** : le registre de pile, «*stack*» ;
- **R14** : le registre de lien, «*link*» ; ❶
- **R15** : le «*program counter*» ou registre ordinal.

❶ *Ces registres sont utilisés dans le cas des appels de fonction :*
Le «link register» *contient l'adresse de retour après exécution de la fonction.*
Le «stack register» *contient l'adresse du sommet de la pile, où on empilera les valeurs courantes des registres, et pour créer les variables locales de la fonction appelée.*

- le **CPSR**, «*Current Program Status Register*» : registre d'état contient des bits d'information sur la dernière instruction utilisée. *Utilisé notamment pour les conditions et branchements.*

Exécution des instructions

Effet Pipeline

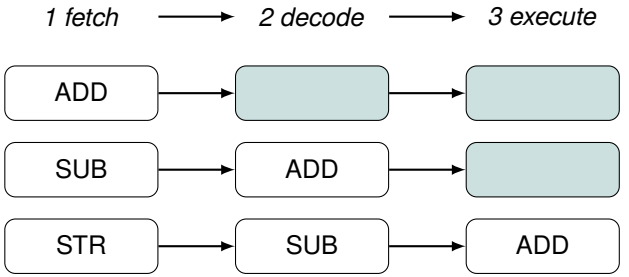
Chaque instruction est exécutée en **trois cycles d'horloge** :

- ▷ un cycle pour le **chargement de l'instruction** depuis la mémoire : *premier étage du pipeline* ;
- ▷ un cycle pour **décoder l'instruction** : *second étage du pipeline* ;
- ▷ un cycle pour **l'exécution** : *troisième étage du pipeline* ;

Lorsqu'une instruction quitte le premier étage du pipeline, une nouvelle instruction peut y entrer :

⟹ une instruction **sort à chaque cycle** du pipeline

⟹ une séquence d'instruction est exécutée **en un cycle chacune** !



En ARMv7 le pipeline a 3 étapes :

- **fetch** : charge une instruction depuis la mémoire ;
- **decode** : identifie l'instruction à exécuter et établit les chemins passant par les bus d'échanges ;
- **execute** : exécute l'instruction et écrit le résultat dans un registre.

Pipeline et «*program counter*»

```
0x8000 LDR pc,[pc,#0]
0x8004 NOP
0x8008 NOP
0x800C DCD JumpAddress
```

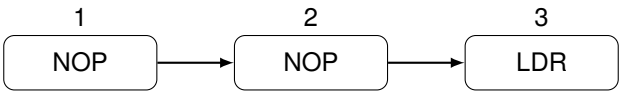
La notation assembleur `LDR pc, [pc,#0]` se traduit par :

- ▷ charge dans PC, «*program counter*» ❶ ;
- ▷ le contenu de PC, «*program counter*», + 0 ❷.

⟹ce qui permet un adressage par décalage.

Mais que vaut PC ?

⟹cela dépend du pipeline d'instruction :



Le PC évolue de la manière suivante :

1. étape 1 du pipeline : `PC = 0x8000` pour l'instruction `LDR` ;
2. étape 2 du pipeline : `PC = 0x8004` pour l'instruction `NOP` ;
3. étape 3 du pipeline : `PC = 0x8008` pour l'instruction `NOP` ;

⟹lorsque l'instruction `LDR` est exécutée dans l'étape 3 du pipeline, `PC` vaut `0x8008` !

31-28	27-25	24-21	20	19-16	15-12	11-0
Condition	Operand type	OpCode	Set Condition Codes	Operand register	Destination register	Immediate Operand

- **Condition** : autorise l'exécution de l'instruction suivant les bits dans le registre CPSR ;
- **Operand type** :spécifie le format de l'opérande des bist 19-0 :
  - par exemple deux registres suivis d'une opérande immédiate ;
- **Opcode** : quelle instruction on veut réaliser comme `Add` ou `MUL` ;
- **Set condition code** : un seul bit indiquant si l'instruction doit mettre à jour le registre CPSR, valeur 0, ou non, valeur 1 ;
- **Operand register** : un registre à utiliser comme entrée ;
- **Destination register** : un registre à utiliser comme sortie ;
- **Immediate operand** : une donnée de petite taille que l'on peut donner directement dans l'instruction.
  - exemple : si on veut ajouter 1 à un registre, on peut mettre la donnée à 1 ce qui évite de mettre 1 dans un autre registre et de faire la somme de ces deux registres.

Architecture 32 bits et 64 bits

- En **32bits** :
  - les **adresses mémoires** sont sur 32 bits ;
  - les **registres du processeur** sont sur 32 bits ;
- En **64bits** :
  - les **adresses mémoires** sont sur 64 bits ;
  - les **registres du processeur** sont sur 64 bits ;
- En **32bits** comme en **64bits**, les **instructions sont sur 32bits** :
  - *Comment peut-on charger une variable depuis la mémoire dasn un regsitre donné avec une instruction sur 32bits ?*
    - l'instruction fait 32bits ;
    - 4bits sont utilisés pour un opcode ;
    - 4bits pour une instruction conditionnelle ;
    - 3bits sont utilisés pour indiquer le type de l'opérande ;
    - 1bit pour indiquer si l'opération affecte le CPSR ;
    - 4bits sont nécessaires pour indiquer le registre ;

    $\Longrightarrow$ **il reste 16bits pour indiquer l'adresse mémoire !**

    $\Longrightarrow$ **il reste 12bits pour indiquer l'adresse mémoire si on a besoin d'indiquer 2 registres !**
- Comment faire ?
  - on peut utiliser un registre pour indiquer l'adresse mémoire : **accés mémoire indirect**
  - Mais comment charger l'adresse mémoire initialement dans le registre en une seule instruction ?
  - charger dans deux registres séparés l'adresse désirée, puis décaler et combiner les deux registres en un seul registre $\Longrightarrow$4 instructions pour arriver au résultat ce qui est excessif !
  - Et alors ? On peut utiliser le PC, «*Program Counter*» pour charger une **mémoire pas trop éloignée**, à 12bits de décalage par rapport au PC, soient **4096 octets accessibles** et beaucoup plus **par décalage de ces bits**.

Définition de contenu

```
label: .byte 74, 0112, 0b00101010, 0x4A, 0X4a, 'J', 'H' + 2
       .word 0x1234ABCD, -1434
       .ascii "Hello World\n"
```

Charger un registre

L'instruction `LDR` peut servir à charger une addresse dans le registe ou la donnée pointée par cette addresse.

Il est également possible d'indexer la mémoire en utilisant la valeur d'un registre comme valeur d'index.

- ▷ addressage relatif au PC, «*Program Counter*» ;
- ▷ charger depuis la mémoire ;
- ▷ indexer à travers la mémoire.

Adressage relatif au PC

Si les données ne sont pas loin de l'instruction utilisant leur adresse, on peut utiliser cet adressage :

```
LDR R1,=helloworld
```

ce qui donne après assemblage :

```
LDR R1, [pc, #20]
```

Ici, la valeur du décalage est de `0x20` *en hexa, soit 32*.

Ce décalage est :

- ▷ déterminé lors de l'assemblage ;
- ▷ sur 12 bits dans l'instruction elle-même :
  - ◇ ce qui donne de 0 à 4095 ;
  - ◇ un bit est utilisé dans l'instruction pour indiquer dans quel sens : ⟹le décalage peut être de $\pm$4095 *words* au maximum.
  - ◇ le décalage peut être par multiple de 1 à 2 octets, comme indiqué dans la table ⟹

`LDR{type} Rt, =label`

Type	Meaning
B	Unsigned byte
SB	signed byte
H	Unsigned halfword (16 bits)
SH	signed halfword (16 bits)
–	omitted for word

Utilisation de la pile

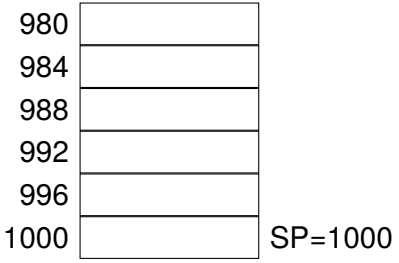
Deux opérations possibles :

- ▷ **push** : ajouter un élément ;
- ▷ **pop** : enlever et retourner l'élément le plus récemment ajouté.

Gestion dans le processeur ARM :

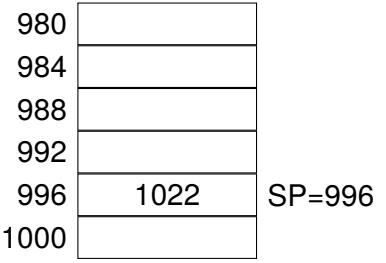
- □ **registre R13** aussi appelé **SP**, «*Stack Pointer*» : il pointe sur l'emplacement de la pile en mémoire ;
- □ deux instructions du jeu d'instruction ARM32 : **LDM**, «*Load Multiple*» et **STM**, «*Store Multiple*» ;
- □ ces instructions sont adaptables :
  - ◇ choix du sens d'augmentation de la pile par incrémentation ou décrémentation des adresses ;
  - ◇ le registre pointe soit sur la fin de la pile, soit sur le prochain emplacement libre ;

  $\Longrightarrow$ *le système est adaptable aux besoins de différents systèmes d'exploitation*.
- □ l'assembleur GNU offre des pseudo-instructions qui s'appuient sur les instructions LDM et STM :
  - ◇ PUSH liste de registre : `PUSH {R0, R5-R12};`
  - ◇ POP liste de registres : `POP {R0-R4, R6, R9-R12};`




```
Push R5
```

R5 = 1022



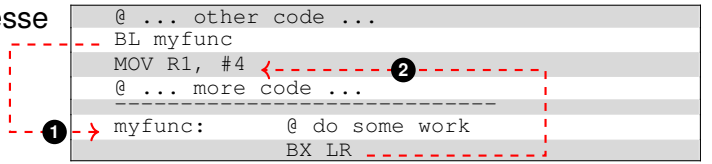
Branchement avec lien de retour

Après l'appel de la fonction, il faut retourner à l'exécution des instructions qui suivent :

- le registre R14, «*link register*» : sert à stocker l'adresse de retour ;
- l'instruction `BL`, «*Branch with Link*» : réalise un branchement aprés avoir stocké l'adresse de la prochaine instruction dans LR ;
- l'instruction `BX` : réalise le retour de fonction en sautant à l'adresse présente dans le registre LR ;

▷ `BL` saute à l'adresse de ❶ et sauvegarde l'adresse de retour dans `LR` ;

▷ `BX` saute à l'adresse stockée dans `LR` ❷



Et comment cela se passe si la fonction appelle une autre fonction ?

On utilise de nouveau dans la fonction `myfunc`, l'instruction `BL` :

⟹`BL` **copie** l'adresse de la prochaine instruction dans le registre `LR`

⟹*ce qui* ***écrase*** *l'ancienne valeur contenue dans* `LR`

⟹la fonction `myfunc` **ne pourra plus retourner** !

⟹il faut :

▷ **sauvegarder** la valeur du registre `LR` dans la pile❸ avant le `BL`

▷ **restaurer** la valeur du regsitre `LR`❹ avant de retourner avec `BX`

```
    @ ... other code ...
    BL myfunc
    MOV R1, #4
    @ ... more code ...
-----------------------------
myfunc: PUSH {LR}❸
    @ do some work ...
    BL myfunc2
    @ do some more work...
    POP {LR}❹
    BX LR
myfunc2: @ do some work
    BX LR
```

- 1 mode utilisateur,«*usr*», non privilégié ;
- 5 modes pour des exceptions + 1 pour le système, priviégiés :

Mode	Sigle	Description
User	usr	mode normal d'exécution non privilégié
Interrupt	_irq	activé lors d'une exception causée par une interruption matérielle
Fast Interrupt	_fiq	activé lors d'une exception causée par une interruption matérielle à gérer rapidement
Abort	_abt	activé lors d'une exception d'accès à la mémoire
Undefined	_und	activé lorsqu'une instruction indéfinie est exécuté (une valeur sur 32bits qui ne correspond à aucune instruction existante)
Supervisor	_svc	Software Interrupt: activé lors du reset ou lors d'un appel système
System	sys	mode privilégié pour le travail de l'OS.

- le passage d'un mode à l'autre peut être fait **manuellement** en modifiant des bits du registre CPSR ;
- les **modes privilégiés** sont activés pour traiter des **interruptions** ou des **exceptions** ;
- le **mode système**, «*sys*», est un **mode spécial** pour accéder à des **ressources protégées** (comme la MMU par exemple si disponible) ;
- les modes traitant les **interruptions** disposent de **registres dupliqués**, remplaçant les registres qu'ils dupliquent durant l'exécution du traitement, ce qui évitent les collisions et corruptions.
- les modes «*usr*» et «*sys*» **partagent le même ensemble** de registres.

Registres dupliqués :

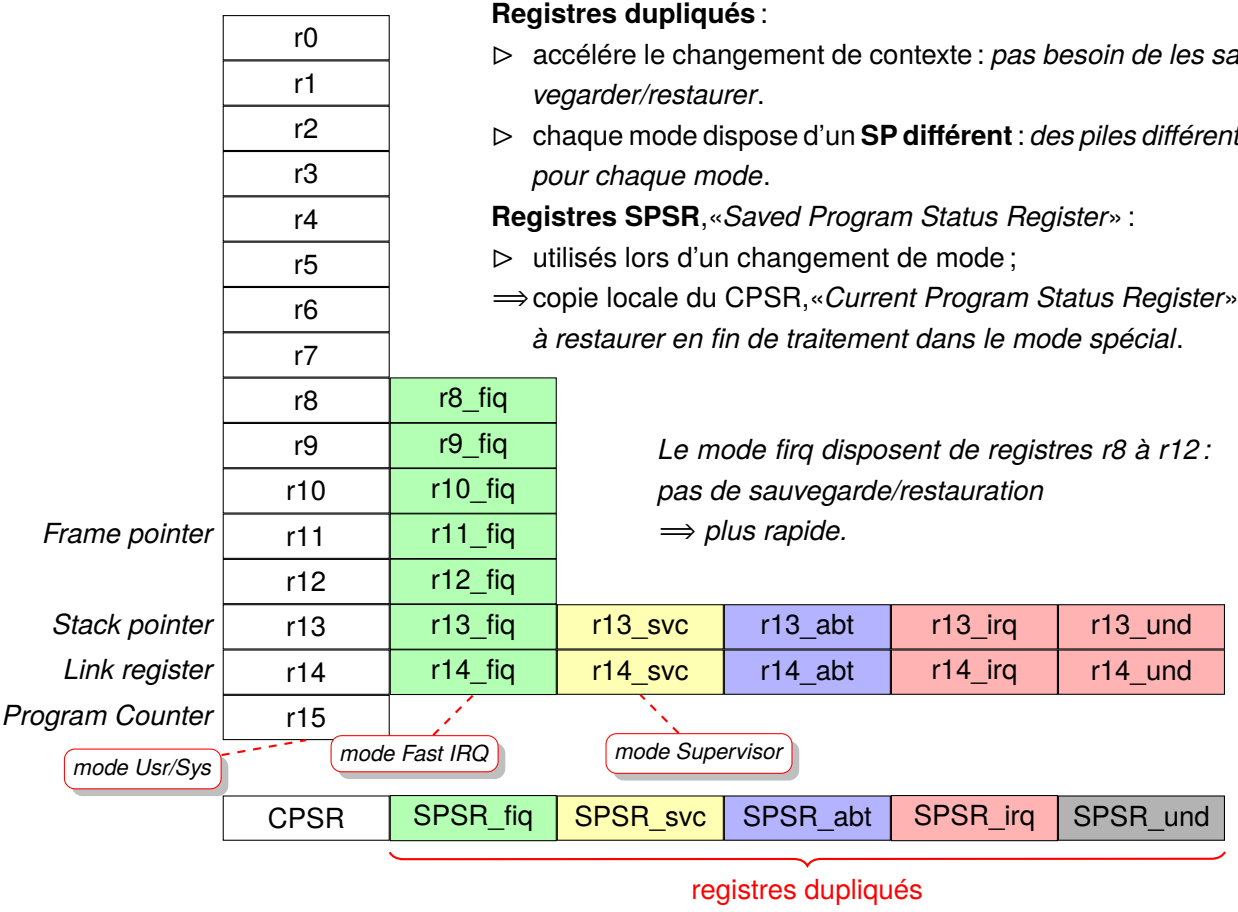
▷ accélére le changement de contexte : *pas besoin de les sauvegarder/restaurer*.

▷ chaque mode dispose d'un **SP différent** : *des piles différentes pour chaque mode*.

Registres SPSR,«*Saved Program Status Register*» :

▷ utilisés lors d'un changement de mode ;

⟹copie locale du CPSR,«*Current Program Status Register*», *à restaurer en fin de traitement dans le mode spécial*.



Le mode firq disposent de registres r8 à r12 : pas de sauvegarde/restauration ⟹ plus rapide.

Mais ça marche comment les interruptions ?

Assigner les interruptions

C'est le **designer du SoC** qui décide quel matériel peut produire quelle interruption : il s'agit d'interconnecter des circuits avec le processeur.

- ▷ SWI, «*SoftWare Interrupt*» ou svc, utilisée pour accéder à des fonctions privilégiées de l'OS ;
- ▷ IRQ, assignées à des interruptions génériques comme des timers périodiques ;
- ▷ FIQ, réservée pour une seule source qui nécessite un faible temps de réponse.

Les latences dans le traitement d'une interruption

La **latence** est l'intervalle de temps entre :

- ▷ l'instant où le signal externe change d'état ;
- ▷ la première instruction du traitement est chargé dans le processeur (fetch) ;

Le système essaie d'atteindre deux buts :

- □ gérer **plusieurs interruptions** simultanément ;
- □ **minimiser** la latence.

Deux métodes pour l'atteindre :

- ▷ autoriser la **gestion imbriquée**, «*nested*», des interruptions ;
- ▷ donner des **priorités** aux différentes sources d'interruption.

Activer/désactiver les interruptions

Les instructions dédiées :

MRS	lire le CPSR
MSR	stocker dans le CPSR
BIC	effacer un bit
ORR	opération OR

Activer une IRQ / FIRQ :

```
MRS r1, cpsr
BIC r1, r1, #0x80/0x40
MSR cpsr_c,r1
```

Désactiver une IRQ / FIRQ :

```
MRS r1, cpsr
ORR r1, r1, #0x80 / 0x40
MSR cpsr_c,r1
```

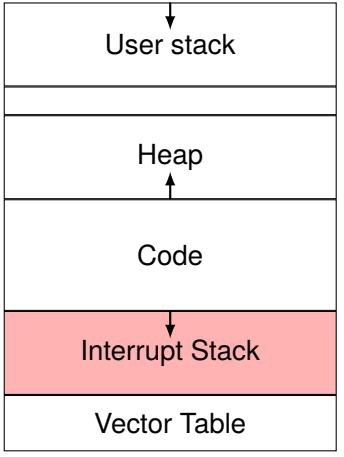
Pourquoi une pile ?

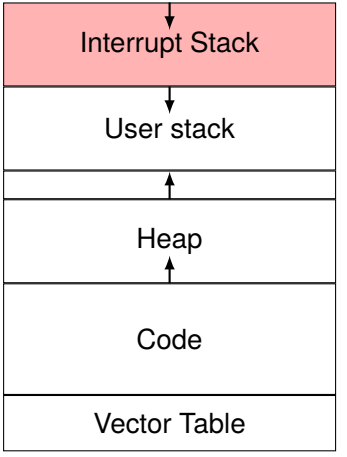
La pile est nécessaire pour le changement de context :

- ▷ empiler les valeurs des registres pour les sauvegarder ;
- ▷ dépiler les valeurs pour restaurer les registres ;
- ▷ variables locales nécessaires pour la gestion des interruptions.

Il faut choisir :

- ☐ la taille de la pile ;
- ☐ sa position dans la mémoire.





Dans ce cas là, la table des vecteurs d'interruption ne peut être écrasée lors d'un dépassement de pile.

Table des vecteurs d'interruption

La **table des vecteurs d'interruption** est une table d'adresses :

- ▷ le processeur ARM **saute** à l'adresse associée à l'exception qui s'est déclenchée ;
- ▷ à cette adresse se trouve une **instruction de saut** vers le code de traitement de cette exception.

Dans le tableau, on trouve une instruction de branchement :

```
ldr pc, [pc,#_IRQ_HANDLER_OFFSET]
```

Où `#_IRQ_HANDLER_OFFSET` est le décalage pour atteindre le code de gestion de l'interruption.

Adresse	Exception	Mode en entrée
0x00000000	Reset	Superviseur
0x00000004	Undefined instruction	Undefined
0x00000008	Software interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	Reserved	Reserved
0x00000018	IRQ	IRQ
0x0000001C	FIRQ	FIRQ

Cette table doit être remplie par l'OS lors du démarrage de la machine.

Priorités

indique si le handler de l'interruption peut être ou non interrompu

définie quelle exception est la plus importante parmi celles déclenchées

Exception	Priorité	bit I	bit F
Reset	1	1	1
Data Abort	2	1	-
FIQ	3	1	1
IRQ	4	1	-
Prefetch Abort	5	1	
SWI	6	1	-
Undefined Instruction	6	1	-

déclenchée lors de l'étape «*execute*» du pipeline

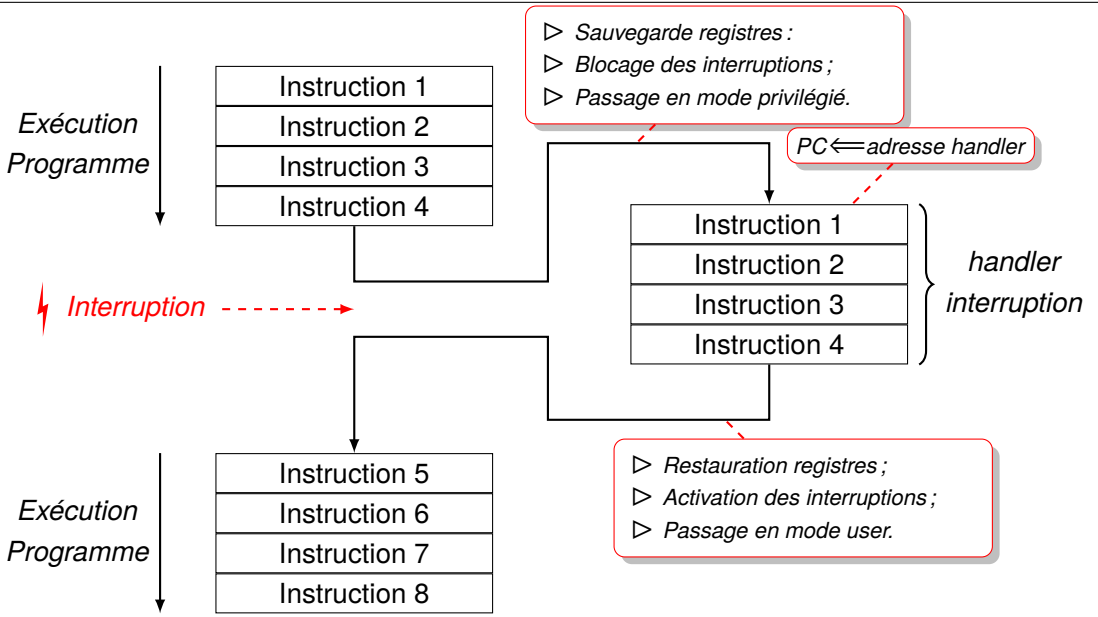
Décalage du registre PC sauvegardé dans le registre LR

Exception	Adresse de retour
Reset	aucune
Data Abort❷	LR - 8
FIQ, IRQ, prefetch Abort❷	LR - 4
SWI, Undefined Instruction	LR

Dans certain cas, le registre PC a avancé au delà de l'instruction qui a causé l'exception :

- ▷ en ❶, il faut revenir en arrière de deux instructions par rapport à la valeur du PC sauvegardée dans le registre LR ;
- ▷ en ❷, il faut revenir en arrière d'une seule instruction.

Ceci est du au pipeline d'exécution du processeur ARM.



Entrée dans le handler d'exception :

❶⇒sauver l'adresse de l'instruction suivante dans le registre LR aproprié ;

❷⇒copier le CPSR dans le SPSR du nouveau mode ;

❸⇒changer le mode en modifiant les bits du CPSR ;

❹⇒aller chercher l'instruction suivante dans la table des vecteurs d'interruption.

Sortie du handler d'exception :

❶⇒charger le registre LR dans le PC (avec le bon décalage) ;

❷⇒restaurer le SPSR dans le CPSR, ce qui remettra les bits de mode comme avant ;

❸⇒effacer le bit de blocage des interruptions s'il avait été positionné.

Et sur un exemple d'interruption logicielle
ça donne quoi ?

On va utiliser l'assembleur `as` pour assembler l'exécutable :

```
@
@ Assembler program to print "Hello World!"
@ to stdout.
@
@ R0-R2 - parameters to linux function services
@ R7 - linux function number
@

.global _start @ Provide program starting
@ address to linker
@ Set up the parameters to print hello world
@ and then call Linux to do it.
_start: mov R0, #1 @ 1 = StdOut
        ldr R1, =helloworld @ string to print
        mov R2, #13
        mov R7, #4
        svc 0 @ Call linux to print

@ Set up the parameters to exit the program
@ and then call Linux to do it.
@ length of our string
@ linux write system call
        mov R0, #0  @ Use 0 return code
        mov R7, #1  @ Service command code 1
                    @ terminates this program
        svc 0       @ Call linux to terminate
.data
helloworld: .ascii "Hello World!\n"
```

l'étiquette `_start` indique au «linker» le point d'entrée du programme

On utilise un appel système pour faire l'affichage

On utilise un appel système pour terminer le programme

`.ascii` indique à l'assembleur de mettre la chaîne dans le segment data, et d'utiliser l'étiquette pour avoir son adresse

Assemblage et exécution :

```
xterm
$ as -o HelloWorld.o HelloWorld.s
$ ld -o HelloWorld HelloWorld.o
$ ./HelloWorld
Hello World!
$
```

- l'instruction `MOV` déplace des données dans un registre.
- le `#4` indique une opérande directe, soit la valeur 4.
- `LDR R1, =helloworld` charge le registre avec l'adresse de la chaîne.
- `svc 0` réalise une interruption logicielle pour donner le contrôle au noyau Linux.

Comment utiliser une fonctionnalité de l'OS ?

```
_start: mov R0, #1 @ 1 = StdOut
        ldr R1, =helloworld @ string to print
        mov R2, #13
        mov R7, #4
        svc 0 @ Call linux to print
```

Les **registres R0 à R4** vont être utilisés pour passer les paramètres à l'appel système.
Lors du retour de l'appel système, la **valeur de retour** sera donnée dans le registre R0.

Ici, on fait appel à l'appel système `print` :

- ▷ on indique dans le registre R0 la destination `stdout` ;
- ▷ l'adresse de la chaîne à afficher est mise dans le registre R1 ;
- ▷ le numéro de l'appel système est donné dans le registre R7, ici c'est la valeur 4 pour indiquer `print`.

On réalise ensuite une **interruption logicielle** avec l'instruction `svc 0` :

- ▷ le contrôle, le CPU, est transmis au code du traitement de cette interruption ;
- ▷ le **numéro** de l'appel système permet de savoir quel code doit être appelé pour réaliser l'appel système ;
- ▷ le code de traitement de l'interruption et de l'appel système est dans le noyau Linux ;
- ▷ grâce au **mécanisme d'interruption** :
  - ◇ le programme **ne sait pas où se trouve** ce code de traitement : *il n'y a même pas accès pour cause de protection d'accès mémoire* ;
  - ◇ le code de traitement est **exécuté dans un mode protégé** du processeur : *il accède à toutes les ressources de la machine, comme l'écran pour y afficher du texte*.
  - ◇ si le code de traitement de l'appel système est **mis à jour**, il n'y a **pas de problème** : *le programme utilise simplement un numéro pour l'identifier*.

Utilisation de la commande `objdump`

On peut obtenir le désassemblage du programme

```
$ objdump -s -d HelloWorld.o

HelloWorld.o:     file format elf32-littlearm

Contents of section .text:
 0000 0100a0e3 14109fe5 0d20a0e3 0470a0e3  ......... ...p..
 0010 000000ef 0000a0e3 0170a0e3 000000ef  .........p......
 0020 00000000                             ....
Contents of section .data:
 0000 48656c6c 6f20576f 726c6421 0a         Hello World!.
Contents of section .ARM.attributes:
 0000 41110000 00616561 62690001 07000000  A....aeabi......
 0010 0801                                 ..

Disassembly of section .text:

00000000 <_start>:
   0:   e3a00001        mov     r0, #1
   4:   e59f1014        ldr     r1, [pc, #20]   ; 20 <_start+0x20>
   8:   e3a0200d        mov     r2, #13
   c:   e3a07004        mov     r7, #4
  10:   ef000000        svc     0x00000000
  14:   e3a00000        mov     r0, #0
  18:   e3a07001        mov     r7, #1
  1c:   ef000000        svc     0x00000000
  20:   00000000        .word   0x00000000
```

instructions en mode LittleEndian

les différents segments du programme

Le segment de code

mnémonique

valeur réelle de l'instruction

L'intérêt du mode LittleEndian ?
pour convertir un entier sur 4 octets en 1 octet,
il suffit de lire que le premier octet.

Ici, on remarque que :

- il n'y a plus d'étiquettes comme dans le source assembleur : elles sont remplacées par des adresses ;
- les **instructions** sont indiquées avec **leur valeur réelle**, indiquée en **hexadécimal** en plus de leur notation sous forme de **mnémonique**, comme `e3a00001` pour `mov r0, #1`.

Décomposition d'une instruction

```
00000000 <_start>:
   0:   e3a00001        mov     r0, #1
   4:   e59f1014        ldr     r1, [pc, #20]   ; 20 <_start+0x20>
   8:   e3a0200d        mov     r2, #13
   c:   e3a07004        mov     r7, #4
  10:   ef000000        svc     0x00000000
```

Soit l'instruction `e3a00001 movr0, #1:`

Hex Digit	e	3	a	0	0	0	0	1
Binary	1110	0011	1100	0000	0000	0000	0000	0001

- Chaque instruction du code commence par un digit hexa à `e` :
  - champs sur 4bits indiquant une exécution conditionnelle suivant les valeurs du registre CSPR ;
  - *ici, la valeur est* `e` *qui indique que l'instruction doit être réalisée de manière inconditionnelle*
- les 3bits suivants `001` indique le **type des opérandes** :
  - *ici, un registre et une valeur immédiate* ;
- les 4bits suivants `1110` est l'**opcode** pour l'instruction `MOV` ;
- le bit suivant `0` indique le **type du paramètre** pour le mode immédiat, qui ici ne sert pas ;
- les 4bits suivants `0000` indiquent le **registre** R0 ;
- les 4bits suivants `0000` indiquent un **autre registre** dans le cas où le `MOV` travaillerait sur deux registres *ce qui n'est pas le cas ici* ;
- les 12bits restants : la **valeur immédiate**, *ici* `1`.

Le source :

```
_start: mov R0, #1 @ 1 = StdOut
        ldr R1, =helloworld @ string to print
        mov R2, #13
        mov R7, #4
        svc 0 @ Call linux to print
```

est devenu :

```
00000000 <_start>:
   0:   e3a00001        mov     r0, #1
   4:   e59f1014        ldr     r1, [pc, #20]   ; 20 <_start+0x20>
   8:   e3a0200d        mov     r2, #13
   c:   e3a07004        mov     r7, #4
  10:   ef000000        svc     0x00000000
```

`ldr R1, =helloworld` est devenu `ldr r1, [pc, #20]; 20 <_start+0x20>`

L'assembleur :

- ▷ l'étiquette `=helloworld` a été traduite en un décalage, «*offset*», depuis le «*program counter*» ;
- ▷ la chaîne de caractères a été placée 20*4 octets plus loin que l'origine du programme indiqué par l'étiquette `_start`, c-à-d :

  ```
  1c:   ef000000        svc     0x00000000
  20:   00000000        .word   0x00000000
  ```

  juste après le code.

- ▷ l'accès à cette mémoire est fait depuis un registre contenant déjà une adresse, car une seule instruction ARM ne permet pas de charger une **adresse entière sur 32bits** directement dans un registre.

```
$ gdb helloword
GNU gdb (Raspbian 10.1-1.7) 10.1.90.20210103-git
Copyright (C) 2021 Free Software Foundation, Inc.
Reading symbols from helloword...
(gdb) list
1
2   @
3   @ Assembler program to print "Hello World!"
4   @ to stdout.
5   @
6   @ R0-R2 - parameters to linux function services
7   @ R7 - linux function number
8   @
9
10   .global _start @ Provide program starting
(gdb) l
11   @ address to linker
12   @ Set up the parameters to print hello world
13   @ and then call Linux to do it.
14   _start: mov R0, #1 @ 1 = StdOut
15      ldr R1, =helloworld @ string to print
16      mov R2, #13
17      mov R7, #4
18      svc 0 @ Call linux to print
19
20   @ Set up the parameters to exit the program
(gdb) l
21   @ and then call Linux to do it.
22   @ length of our string
23   @ linux write system call
24      mov R0, #0  @ Use 0 return code
25      mov R7, #1  @ Service command code 1
26                  @ terminates this program
27      svc 0       @ Call linux to terminate
28   .data
29   helloworld: .ascii "Hello World!\n"
(gdb)
```

GDB	
Command (short form)	*Description*
list (l)	*List the program*
break (b) line	*Set breakpoint at line*
run (r)	*Run the program*
step (s)	*Single-step program*
continue (c)	*Continue running the program*
quit (q or control-d)	*Exit gdb*
control-c	*Interrupt the running program*
info registers (i r)	*Print out the registers*
info break	*Print out the breakpoints*
delete n	*Delete breakpoint n*
x /nuf expression	*Show contents of memory*

Utilisation de gdb : désassemblage et pose de point d'arrêt, «*breakpoints*»

xterm

```
(gdb) disassemble _start
Dump of assembler code for function _start:
   0x00010074 <+0>:    mov    r0, #1
   0x00010078 <+4>:    ldr    r1, [pc, #20] ;
                      0x10094 <_start+32>
   0x0001007c <+8>:    mov    r2, #13
   0x00010080 <+12>:    mov    r7, #4
   0x00010084 <+16>:    svc    0x00000000
   0x00010088 <+20>:    mov    r0, #0
   0x0001008c <+24>:    mov    r7, #1
   0x00010090 <+28>:    svc    0x00000000
   0x00010094 <+32>:    muleq   r2, r8, r0
End of assembler dump.
(gdb)
```

décalage

xterm

```
(gdb) b _start
Breakpoint 1 at 0x10074: file HelloWorld.s, line 14.
(gdb) r
Starting program:
/home/pi/ASMRASPI/hello_world_asm/helloword
Breakpoint 1, _start () at HelloWorld.s:14
14   _start: mov R0, #1 @ 1 = StdOut
(gdb) s
15      ldr R1, =helloworld @ string to print
(gdb) info registers
r0             0x1                1
r1             0x0                0
r2             0x0                0
r3             0x0                0
r4             0x0                0
r5             0x0                0
r6             0x0                0
r7             0x0                0
r8             0x0                0
r9             0x0                0
r10            0x0                0
r11            0x0                0
r12            0x0                0
sp             0x7efff620         0x7efff620
lr             0x0                0
pc             0x10078            0x10078 <_start+4>
cpsr           0x10               16
fpscr          0x0                0
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00010074
HelloWorld.s:14
   breakpoint already hit 1 time
```

décalage

xterm

```
(gdb) r
Starting program: /home/pi/ASMRASPI/hello_world_asm/helloword

Breakpoint 1, _start () at HelloWorld.s:14
14   _start: mov R0, #1 @ 1 = StdOut
(gdb) s
15      ldr R1, =helloworld @ string to print
(gdb) info registers
r0             0x1                 1
r1             0x0                 0
r2             0x0                 0
r3             0x0                 0
r4             0x0                 0
r5             0x0                 0
r6             0x0                 0
r7             0x0                 0
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
sp             0x7efff620          0x7efff620
lr             0x0                 0
pc             0x10078             0x10078 <_start+4>
cpsr           0x10                16
fpscr          0x0                 0
(gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x00010074 HelloWorld.s:14
   breakpoint already hit 1 time
(gdb) x /4ubft _start
0x10074 <_start>:   00000001   00000000   10100000   11100011
(gdb) x /4ubfi _start
   0x10074 <_start>:   mov   r0, #1
=> 0x10078 <_start+4>:   ldr   r1, [pc, #20]   ; 0x10094 <_start+32>
   0x1007c <_start+8>:   mov   r2, #13
   0x10080 <_start+12>:   mov   r7, #4
(gdb) x /4ubfx _start
0x10074 <_start>:   0x01   0x00   0xa0   0xe3
(gdb) x /4ubfd _start
0x10074 <_start>:   1   0   -96   -29
```

31	30	29	28	27	-	24	-	19 - 16	-	9	8	7	6	5	4 - 0
N	Z	C	V	Q		J		GE		E	A	I	F	T	M

- **N**egative : N vaut 1 si la valeur est négative, et 0 si la valeur est positive ;
- **Z**ero : Z vaut 1 si le résultat est zéro (par exemple lors d'une comparaison), et 1 si le résultat n'est pas zéro ;
- **C**arry : indique pour
  - une opération d'addition s'il y a un bit de retenu comme résultat du calcul, c-à-d un dépassement, «*overflow*», de capacité ;
  - une opération de soustraction s'il y a un bit de retenu, c-à-d un dépassement, «*underfow*», de capacité ;
  - une opération de décalage, le bit qui a été décalé vers l'extérieur ;
- o**V**erflow : pour l'addition et la soustraction, indique un «*overflow*».
  Pour certaines autres instructions, ce bit peut être utilisé pour indiquer une erreur.
- les bits liés aux **interruptions** :
  - **I** : lorsqu'il est à 1 : désactive les IRQ, «*interrupt request*» ;
  - **F** : lorsqu'il est à 1 : désactive les FIQ, «*Fast interrupt request*» (par exemple : traiter les paquets réseau, les mouvements de la souris) ;
  - **A** : lorsqu'il est à 1 : désactive les abandons ;
- les bits liés aux **instructions** :
  - **Thumb** : lorsqu'il est à 1 : indique des instructions compactes sur 16 bits ;
  - **Jazelle** : lorsqu'il est à 1 : mode obsolète pour l'exécution directe de bytecode Java ;
- les **autres** bits :
  - **Q** : lorsqu'il est à 1 : indique un «*underflow*» ;
  - **GE** : contrôle le «*Greater than Equal*» dans le traitement des données SIMD ;
  - **E** : contrôle l'«*endianness*» pour le traitement des données.
- **M** : indique si le processeur est en mode «***user***» ou «***supervisor***».

Branchement sur condition

```
B{condition} label
```

{condition}	Flags	Meaning
EQ	Z set	equal
NE	Z clear	not equal
CS or HS	C set	higher or same (unsigned >=)
CC or LO	C clear	lower (unsigned <)
MI	n set	negative
PL	n clear	positive or zero
VS	V set	overflow
VC	V clear	no overflow
HI	C set and Z clear	higher (unsigned >)
LS	C clear and Z set	lower or same (unsigned <=)
GE	n and V the same	Signed >=
LT	n and V differ	Signed <
GT	Z clear, n and V the same	Signed >
LE	Z set, n and V differ	Signed <=
AL	any	always (same as no suffix)

Mais le rôle fondamental de l'OS n'est-il pas
d'organiser l'accès au matériel
pour le logiciel de l'utilisateur ?

Comment protéger le code de l'OS des programmes de l'utilisateur ?
⟹disposer de **différents niveaux de privilèges**, au moins deux :

- **Mode utilisateur** : mode normal utilisé pour l'exécution des programmes de l'utilisateur s'exécutant dans l'OS ;
- **Mode privilégié** : réservé à l'exécution de l'OS : il existent des opérations qui peuvent être exécutées en mode privilégié et qui ne peuvent pas être exécutées en mode utilisateur.

Processeur ARM

- dispose de 6 modes **privilégiés** et 1 **non privilégié** ;
- 5 de ces modes disposent de leur propre registre de pile, «*stack register*» R13 et registre de lien, «*link register*» R14 :
  - quand on change vers un **mode privilégié**, les regsitres de pile R13 et de lien R14 associés à ce mode **remplacent** les registres R13 et R14 de l'utilisateur ;
- depuis chaque **mode privilégié**, les registres de pile R13 et de lien R14 **des autres modes** sont **accessibles** ;
- depuis le mode **utilisateur**, les registres de pile R13 et de lien R14 **des modes privilégiés** sont **inacessibles**.

Comment avoir accès :

à la mémoire

à un périphérique

si on n'a pas d'OS ?

C'est quoi ?

C'est le **lien** entre le **SoC** et le **firmware/logiciel** qui s'exécute dessus.

La «carte mémoire» ou «*memory map*» définie comment le logiciel peut :

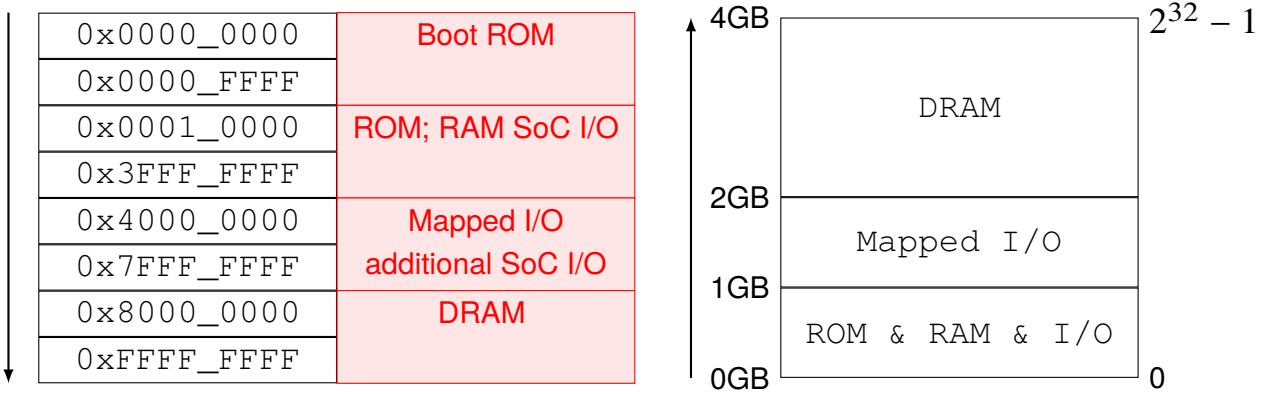
- ▷ accéder au matériel ;
- ▷ déclencher les opérations élémentaires, «*primitives*» pour contrôler et observer les fonctions du SoC.

Comment est-elle construite ?

En deux étapes :

1. les **blocs fonctionnels**, comme une UART, viennent avec une carte, «*map*», de leur **registres internes** qui représente leur structure d'adresses locales :
   - ◇ une **liste d'adresses** définies à partir d'une adresse de référence à zéro ;
   - ◇ chacune de ces adresses indique :
     - ⋆ où l'on peut trouver les CSRs, «*Control and Status Registers*» ;
     - ⋆ les détails et le but des différents **champs de bits**, «*bitfields*» ;
     - ⋆ les **règles** pour lire/écrire et effacer ces champs de bits ;
     - ⋆ comment gérer les **interruptions** et la **gestion d'énergie** associés (allumer/éteindre le composant).
2. les blocs fonctionnels sont **intégrés** dans le SoC au travers d'un circuit d'interconnexion :
   - ◇ chaque fonction est associé à un **adresse mémoire** qui la rend accessible de manière distincte, par exemple :
     - ⋆ un registre d'un bloc fonctionnel a une adresse locale en `0xA3` ;
     - ⋆ le bloc fonctionnel est associé à l'adresse `0x2000` ;
     - ⋆ $\Longrightarrow$ **le registre est accessible à `0x20A3` par le logiciel s'exécutant sur le processeur embarqué**.

- Boot ROM : région de la mémoire où démarre le travail du CPU ;
- ROM, RAM SoC I/O : internal ROM, SRAM, mémoire statique, registres des périphériques du SoC (divisée en pages de 64KB, pour alignement sur des pages de 64KB avec la MMU des ARMv8) ;
- Mapped I/O and Additional SoC I/O : région des E/S mappées mémoire comme le PCIe (alignée également par page de 64KB) ;
- DRAM : l'espace d'adresse recommandé pour la DRAM qui doit être contigue et doit être placé vers l'adresse haute de la «*memory map*».



L'utilisation du LPAE, «Large Physical Address Extension» permet un espace d'adressage sur 36 ou 40bits en ARMv7 32bits et sur 48bits en ARMv8 64bits :
MMU : adresses 32bits $\Longrightarrow$ 40bits IPA *Intermediate Physical Address* $\Longrightarrow$ 40bits PA, *Physical Address*.

Memory Map : exemple avec Versatile/PB, plateforme simulée par Qemu


```
Memory map for Versatile/PB:

0x10000000 System registers
0x10001000 PCI controller config registers
0x10002000 Serial bus interface
0x10003000 Secondary interrupt controller
0x10004000 AACI (audio)
0x10005000 MMCI0
0x10006000 KMI0 (keyboard)
0x10007000 KMI1 (mouse)
0x10008000 Character LCD Interface
0x10009000 UART3
0x1000a000 Smart card 1
0x1000b000 MMCI1
0x10010000 Ethernet
0x10020000 USB
0x10100000 SSMC
0x10110000 MPMC
0x10120000 CLCD Controller
0x10130000 DMA Controller
0x10140000 Vectored interrupt controller
0x101d0000 AHB Monitor Interface
0x101e0000 System Controller
0x101e1000 Watchdog Interface
0x101e2000 Timer 0/1
0x101e3000 Timer 2/3
0x101e4000 GPIO port 0
0x101e5000 GPIO port 1
0x101e6000 GPIO port 2
0x101e7000 GPIO port 3
0x101e8000 RTC
0x101f0000 Smart card 0
0x101f1000 UART0
0x101f2000 UART1
0x101f3000 UART2
0x101f4000 SSPI
0x34000000 NOR Flash
```

```
/* Hardware Registers */
#define UART0_BASE     0x101F1000
#define DR       0x0
#define RSRECR   0x4
#define FR       0x18
#define ILPR     0x20
#define IBRD     0x24
#define FBRD     0x28
#define LCRH     0x2C
#define CR       0x30
```

l'adresse mémoire de l'UART0

l'adresse locale du CSR

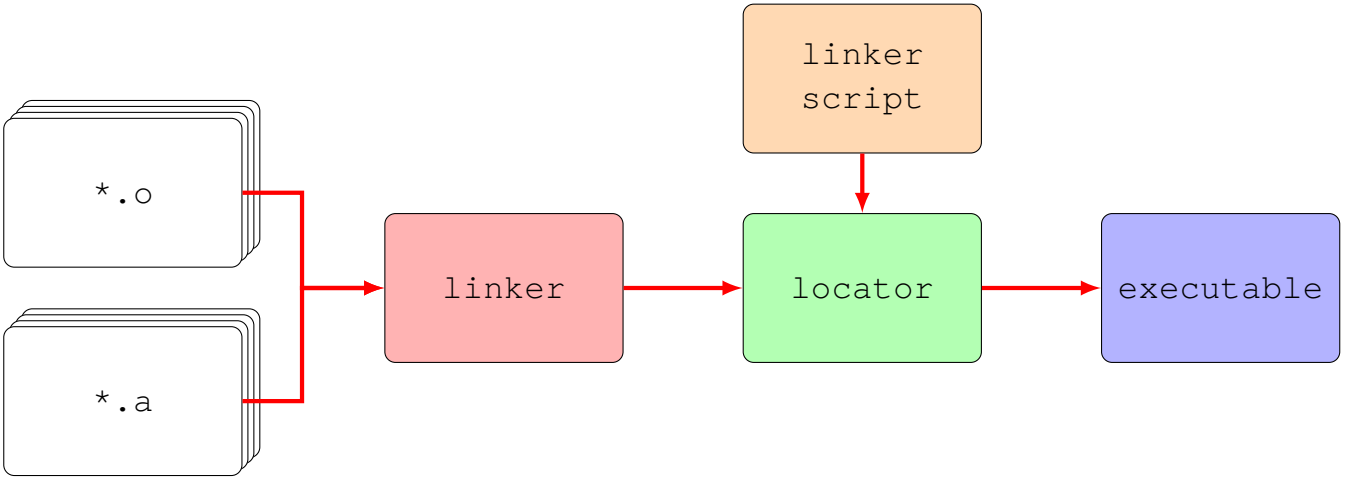
```
/* Set configuration registers based the configuration
struct */
int vpb_uart_configure(struct vpb_uart_dev* dev,
             struct vpb_uart_config* config)
{
   uint32_t reg_data = 0;
   int ret = 0;

   /* Acquire lock for hardware*/
   mutex_lock(&dev->hw_mutex);

   /* Disable UART */
   reg_data = ioread8(dev->iomem + CR) & ~CR_UARTEN;
   iowrite8((uint8_t)reg_data, dev->iomem + CR);

   /* Flush Fifo */
   while(ioread8(dev->iomem + FR) & FR_BUSY);
   reg_data = ioread8(dev->iomem + LCRH) & ~LCRH_FEN;
   iowrite8((uint8_t)reg_data, dev->iomem + LCRH);

   /* Set baudrate */
   //Precalculated for baudrate 9600
   iowrite16((uint16_t)0x9c, dev->iomem + IBRD);
   iowrite8((uint8_t)0x10, dev->iomem + FBRD);
```

adresse de base + adresse locale

Ok Pour la «*memory map*»...

Mais comment l'intégrer dans un programme ?

Rôle du «*linker script*» dans le développement pour l'embarqué

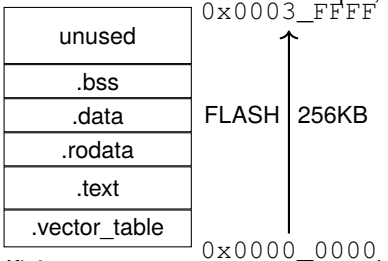


Lors du processus de compilation, le «*locator*» est chargé de :

- ▷ **lire** le «*linker script*» ;
- ▷ **rassembler** les **objets** et les **bibliothèques** utilisées pour produire l'exécutable, comme dans le cas du développement C classique ;
- ▷ assigner des **adresses** à chaque **section** de «*code*» et de «*data*» en accord avec la «*memory map*» du matériel embarqué ciblé ;
- ▷ **l'exécutable** est alors **installable** dans le matériel embarqué.

Linker & Locator : le «*linker script*» pour se conformer à la «*memory map*»

- ▷ défini la **taille** des différentes **sections** mémoire : code, data, taille de la pile, *etc.* ;
- ▷ assigne des positions en **adresses absolues** pour les différentes sections dans la mémoire flash et RAM du matériel embarqué ;
- ▷ est écrit en «*GNU linker script language*» avec une extension `.ld`.



Les 6 commandes principales :

- ☐ ENTRY : définie le point d'entrée dans le fichier ELF : en général, le «*Reset_handler*».

```
ENTRY(Reset_handler)
```

- ☐ MEMORY : définie les différentes régions memoire de chaque section.

```
MEMORY
{
   FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 256K
    RAM (rwx) : ORIGIN = 0x20000000, LENGTH = 32K
}
```

Deux régions sont définies : `FLASH` et `RAM` em accord avec la «*memory map*» du SoC ciblé, avec des atributs d'accès où on indique par exemple que la FLASH n'est pas inscriptible.

- ☐ SECTIONS : crée les sections compilées du programme dans le fichier ELF final : .text, .data, .bss, stack *etc*

```
SECTIONS
{
   PROVIDE( _stack_ptr = ORIGIN(RAM) + LENGTH(RAM));
    .text :
    {
        _text = .;
        KEEP(*(.vector_table))
        *(.text*)
        *(.rodata*)
        _etext = .;
    } > FLASH
    .data :
    {
        _data = .;
        *(.data*)
        _edata = .;
    } > RAM AT >FLASH
```

place la pile au sommet de la mémoire RAM

le . exprime l'adresse courante

copie les données initialisées de la flash vers la ram

```
    .bss :
    {
        _bss = .;
        *(.bss*)
        *(COMMON)
        _ebss = .;
    } > RAM
}
```

celles-ci sont non initialisées

Dans la commande SECTIONS, il y a la définition de chaque section avec le code, les données initialisées ou non et leur placement dans les régions.

- ☐ KEEP, ALIGN (pour aligner par multiple de 4 octets par exemple) et AT> pour placer.

Utilisons QEMU
pour disposer
d'une plateforme matérielle simulée

Exemple avec la plateforme «*versatilepb*» de Qemu : comment fait Linux ?

Description de la plateforme :

- PCI/PCIe devices
- Flash memory
- One PL011 UART $\Longrightarrow$ *le port série*
- An RTC
- A PL061 GPIO controller
- An optional SMMUv3 IOMMU
- hotpluggable DIMMs
- hotpluggable NVDIMMs
- 32 virtio-mmio transport devices
- large amounts of RAM (at least 255GB, and more if using highmem)
- many CPUs (up to 512 if using a GICv3 and highmem)
- Secure-World-only devices if the CPU has TrustZone:
- A second PL011 UART
- A second PL061 GPIO controller, with GPIO lines for triggering a system reset or system poweroff
- A secure flash memory
- 16MB of secure RAM

On récupère le «*.dtb*», «*device tree binary*» depuis Qemu :

xterm

```
$ qemu-system-aarch64 -machine virt -cpu
cortex-a15 -machine dumpdtb=virt.dtb
$ dtc -I dtb -O dts -o - virt.dtb
```

Rôle du «*device tree*»

Le «*device tree*» permet d'adapter le noyau Linux à la «*memory map*» du SoC.

Le périphérique est connu du Linux, il peut nécessiter un pilot et donne lieu à une entrée dans «`/dev`»

Le contenu du «*dtb*» :

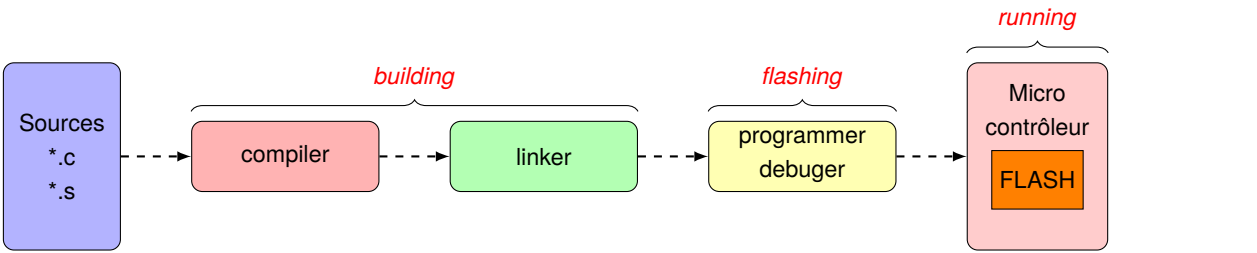
```
/dts-v1/;
/ {
    interrupt-parent = <0x8002>;
    #size-cells = <0x02>;
    #address-cells = <0x02>;
    compatible = "linux,dummy-virt";
    ...
    memory@40000000 {
        reg = <0x00 0x40000000 0x00 0x8000000>;
        device_type = "memory";
    };
    ...
    pl011@9000000 {
        clock-names = "uartclk\0apb_pclk";
        clocks = <0x8000 0x8000>;
        interrupts = <0x00 0x01 0x04>;
        reg = <0x00 0x9000000 0x00 0x1000>;
        compatible = "arm,pl011\0arm,primecell";
    };
    ...
    flash@0 {
        bank-width = <0x04>;
        reg = <0x00 0x00 0x00 0x4000000 0x00
0x4000000 0x00 0x4000000>;
        compatible = "cfi-flash";
    };
    ...
        cpu@0 {
            phandle = <0x8001>;
            reg = <0x00>;
            compatible = "arm,cortex-a15";
            device_type = "cpu";
        };
    };
    ...
};
```

l'adresse d'accès à l'UART pour laquelle il faut un pilote

Exemple avec la plateforme «*versatilepb*» de Qemu : cadre de développement

Pour développer une application «*bare metal*», il faut :

- un **compilateur/assembleur adapté à l'architecture de la machine cible** disponible pour son propre environnement de développement.
  Exemple : un compilateur ARM pour Linux basé x86 (on parle alors de «cross-compiler»*) ;*
- la «***memory map***» de la machine cible ;
  Elle décrit les périphériques disponibles et leurs adresses d'accès.
- un **fichier de démarrage** du processeur et **d'installation des sections** en mémoires :
  - écrit en **assembleur** pour contrôler précisèment les instructions et leur placement ;
  - installation de la **table des vecteurs d'interruption**, avec par exemple :
    - interruption «*reset*» qui va débuter notre programme ;
    - interruption «*svc*» pour passer du mode non privilégié à privilégié ;
  - installation des **fonctions de traitement de ces interruptions**, «*handler*», en rapport avec la table des vecteurs d'interruption (pour obtenir l'adresse de la fonction) ;
    C'est le «linker& Locator» *qui va définir les adresses et intégrer ces adresses dans le code*.
  - configuration des **différents registres de pile** pour les **différents modes d'opération** du processeur ;
    Allocation des différentes piles en mémoire, «stack»*, nécessaires au différents traitement, code de gestion, associé à leur mode d'opération (mode privilégié, non privilégié, IRQ, FIRQ, sys* etc.*).*
- les **fichiers sources** pour le code du **logiciel utilisateur**, les **pilotes** de périphériques *etc.*



Exemple avec la plateforme «*versatilepb*» de Qemu : exemple de programme

But du programme : afficher «*Hello world*» sur le port série d'une plateforme simulée dans Qemu.

Il faut les **outils** suivants :

- ☐ suite de développement pour processeur ARM 32bits : compilateur, linker/locator, assembleur, débogueur ;
  On parle également de «toolchain».
- ☐ émulateur Qemu.

Il faut les **fichiers** suivants :

- ▷ `bootstrap.s` : en assembleur, pour le démarrage du processeur ;
- ▷ `linker.ld` : destiné au linker/locator, pour adapter le code à la «*memory map*» du matériel cible ;
- ▷ `svc_handler.c` : bibliothèque en C de fonctions «*lire/écrire sur le port série*» :
  - ◇ appelée au travers d'un «*appel système*» depuis le code de l'utilisateur ;
  - ◇ initialise et configure le port série suivant l'interface d'accès au matériel (adresse de base + adresses locales des CSR pour le composant UART) ;
  - ◇ analyse la chaîne et envoie un à un ses caractères sur le port série ;
- ▷ `main.s` : le programme principal, utilisateur, qui va afficher «*Hello World*».
  Ici, il est écrit en assembleur : il configure des registres et réalise l'appel système vers la fonction de bibliothèque décrite plus hait).

Exemple avec la plateforme «*versatilepb*» de Qemu : exemple de programme

Le contenu du fichier «*bootstrap.s*» :

```
.global _start
_start:
        LDR PC, reset_addr
        LDR PC, hang_addr
        LDR PC, svc_top_handler_addr
        LDR PC, hang_addr
reset_addr:      .word reset
hang_addr:       .word hang
svc_top_handler_addr: .word svc_top_handler

.align 4

/** reset: args(void) returns(void)
 *  Resets registers, sets up the stack pointer,
 *  and then branches to the program entry point.
 */
.func
reset:
        MOV R0, #0x10000
        MOV R1, #0x00000
        LDMIA R0!, {R2-R5}
        STMIA R1!, {R2-R5}
        LDMIA R0!, {R2-R5}
        STMIA R1!, {R2-R5}

        MSR CPSR_c, 0x13  /* Supervisor mode */
        MOV SP, #0x10000
        MSR CPSR_c, 0x10  /* User mode */
        MOV SP, #0x9000

        BL main     /* Branch to program entry */
.endfunc
```

❶ ⟹ ISR Vector Table, table des vecteurs d'interruption

la suite du fichier «*bootstrap.s*» :

```
/** hang: args(void) returns(void)
 */
.func
hang:
        B hang
.endfunc
/** svc_top_handler: args(void) returns(void)
 *  Called with SVC/SWI instruction,
 *  hands control to C SVC handler (e.g
 *  for printing characters).
 *  In the case of SVC 1, svc_handler() is stored
 *  in a local variable memory address
 *  to be retrieved after all the
 *  registers are restored.
 */
.func
svc_top_handler:
  STMFD SP!, {R0-R11,LR}

  LDR R0, [LR, #-4] /* Get SVC instruction */
  BIC R0, #0xFF000000 /* Mask off SVC number */
  MOV R4, R0
  MOV R1, SP
  BL svc_handler
  MOV R5, R0

 LDMFD SP!,{R0}/* R4:SVC code,R5:svc_handler() */
  CMP R4, #1
  MOVEQ R0, R5

  LDMFD SP!, {R1-R11,PC}^
.endfunc
```

En cas de SWI, appel de la fonction svc_handler()

Exemple avec la plateforme «*versatilepb*» de Qemu : exemple de programme

La fin du fichier «*bootstrap.s*»

```
/** halt: args(void) returns(void)
 *  Generates reset signal, halting execution and
exiting QEMU.
 */
SYS_LOCK:        .word 0x10000020
SYS_RESETCTL:    .word 0x10000040
unlock_signal:   .word 0x0000A05F
reset_signal:    .word 0b100000110
.global halt
.func
halt:
        LDR R0, SYS_LOCK
        LDR R1, unlock_signal
        STR R1, [R0]
        LDR R0, SYS_RESETCTL
        LDR R1, reset_signal
        STR R1, [R0]
.endfunc
```

Le contenu du fichier «*linker.ld*» :

```
MEMORY
{
    ram :    ORIGIN = 0x00000000,
             LENGTH = 0x07FFFFFF
}
SECTIONS
{
    . = 0x10000;
    .text : { *(.text*) }
    .data : { *(.text*) }
    .bss  : { *(.text*) }
}
```

Le fichier «*main.s*» :

```
.global main /* Must declare main globally for linker */

hello:  .string "Hello World!"

.align 4

/** main: args(void) returns(void)
 *  Main assembly procedure
 */
.func
main:
        ADR R0, hello
        SVC 3 /* Print string at address in R0 */
        MOV R0, #'\n'
        SVC 0
        SVC 2
.endfunc
```

l'adresse de la chaîne

Une fois compilé, on obtient le fichier ELF :

```
xterm
$ readelf -A main.elf
Attribute Section: aeabi
File Attributes
  Tag_CPU_name: "ARM926EJ-S"
  Tag_CPU_arch: v5TEJ
  Tag_ARM_ISA_use: Yes
  Tag_THUMB_ISA_use: Thumb-1
  Tag_ABI_PCS_wchar_t: 4
  Tag_ABI_FP_denormal: Needed
  Tag_ABI_FP_exceptions: Needed
  Tag_ABI_FP_number_model: IEEE 754
  Tag_ABI_align_needed: 8-byte
  Tag_ABI_enum_size: small
```

Exemple avec la plateforme «*versatilepb*» de Qemu : exemple de programme

Le code du gestionnaire de l'interruption «*svc*» (se comporte comme un appel système) :

```
void halt ( void ); /* prototype of the function hang() written in asm */
...
/** svc_handler:
 *  Executes SVC (formerly SWI) routine specified by target
 *  0) Output the single character in R0 to UART0
 *  1) Read in, to R0, a single character typed in UART0
 *  2) Halt execution
 *  3) Print a string, whose start address is in R0, to UART0
 *  4) Print out, to UART0, in decimal, the (signed) integer stored in R0
 */
char svc_handler ( uint32_t target, uint32_t *reg )
{
    char * str;

    switch (target)
    {
        case 0:
            uart_send(UART0, reg[0]);
            break;
        case 1:
            return (int32_t) uart_read(UART0);
        case 2:
            halt();
            break;
        case 3:
            uart_send_string(UART0, (char*) reg[0]);
            break;
        case 4:
            str = int_to_string(reg[0]);
            uart_send_string(UART0, str);
            break;
    }

    return target;
}
```

La gestion de l'appel système :

- ▷ écrit en C ;
- ▷ utilise le paramètre du `svc` pour déterminer dans le `case` quel traitement est demandé ;
- ▷ exemple `svc 3` envoi sur l'UART le contenu de la chaîne dont l'adresse est dans le registre `R0`.

Exemple avec la plateforme «*versatilepb*» de Qemu : exemple de programme

Un extrait du code du «*pilote*» de l'UART :

```
enum {
    RXFE = 0x10,
    TXFF = 0x20,
};

pl011_T * const UART0 = (pl011_T *) 0x101F1000;

static void uart_send ( pl011_T *pl011, uint32_t c );
static void uart_send_string ( pl011_T *pl011, char * s );
static char uart_read ( pl011_T *pl011 );
static char *int_to_string(int32_t val_s);
char svc_handler ( uint32_t target, uint32_t *reg );

/** uart_send_string:
 *  Sends a single character to the serial port.
 */
static void uart_send ( pl011_T *pl011, uint32_t c )
{
    while ((pl011->FR & TXFF) != 0) {};
    pl011->DR = c;
}

/** uart_send_string:
 *  Sends a null terminated string to the serial port.
 */
static void uart_send_string ( pl011_T *pl011, char * s )
{
    while(*s != '\0')
    {
        while ((pl011->FR & TXFF) != 0) {};
        pl011->DR = *s;
        s++;
    }
}
```

l'adresse mappée en mémoire du périphérique UART

Et l'exécution finale en mode déboguage :

xterm

```
$ qemu-system-arm -M versatilepb -m 128M -nographic -no-reboot -audiodev none,id=none -kernel
examples/main.bin -S -s
Hello World!
```

Et on se connecte avec gdb :

xterm

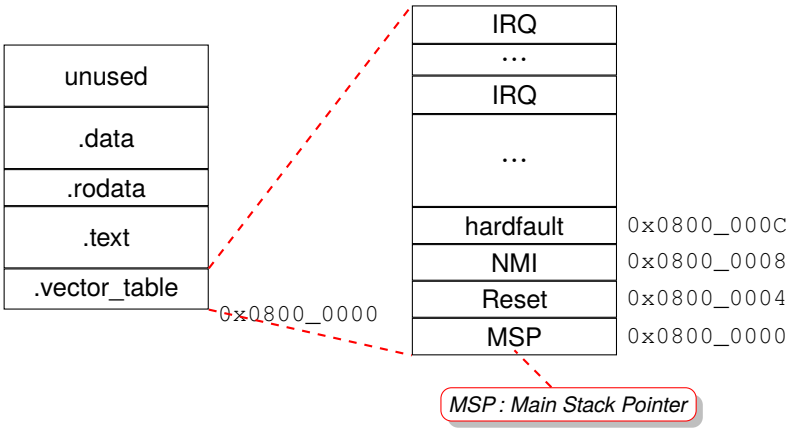
```
$ gdb-multiarch -nx -ex 'target remote localhost:1234'
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.

For help, type "help".
Type "apropos word" to search for commands related to "word".
Remote debugging using localhost:1234

0x00000000 in ?? ()
(gdb) list
No symbol table is loaded.  Use the "file" command.
(gdb) add-symbol-file main.elf
add symbol table from file "main.elf"
(y or n) y
Reading symbols from main.elf...
(gdb) list
1   .global _start
2   _start:
3           LDR PC, reset_addr
4           LDR PC, hang_addr
5           LDR PC, svc_top_handler_addr
6           LDR PC, hang_addr
7   reset_addr:     .word reset
8   hang_addr:      .word hang
9   svc_top_handler_addr: .word svc_top_handler
10
(gdb)
```

Que fait le programme «*startup.s*» ?

Le rôle du fichier `startup`

- écrit en général en **assembleur** ;
- responsable pour **configurer l'environnement d'exécution** nécessaire au programme de l'utilisateur ;
- exécuter **avant** le «*main()*» de l'utilisateur ⟹ c'est lui qui est responsable de l'appel à la fonction «*main()*» ;
- **dépend de l'architecture** du processeur cible ;
- installe la **table des vecteurs d'interruption** ⟹ dépend du micro-contrôleur pour son adresse ;
- initialise la **pile** dépend de la mémoire disponible et de sa taille ;
- initialise les sections `.data` et `.bss` dans la mémoire principale⟹ copie de la FLASH dans la SRAM.

Exemple sur un processeur ARM



L'adresse `0x08000000` est définie par l'architecture du processeur.

Qu’est-ce que le format ELF ?

Qu'est-ce qu'un fichier ELF ?

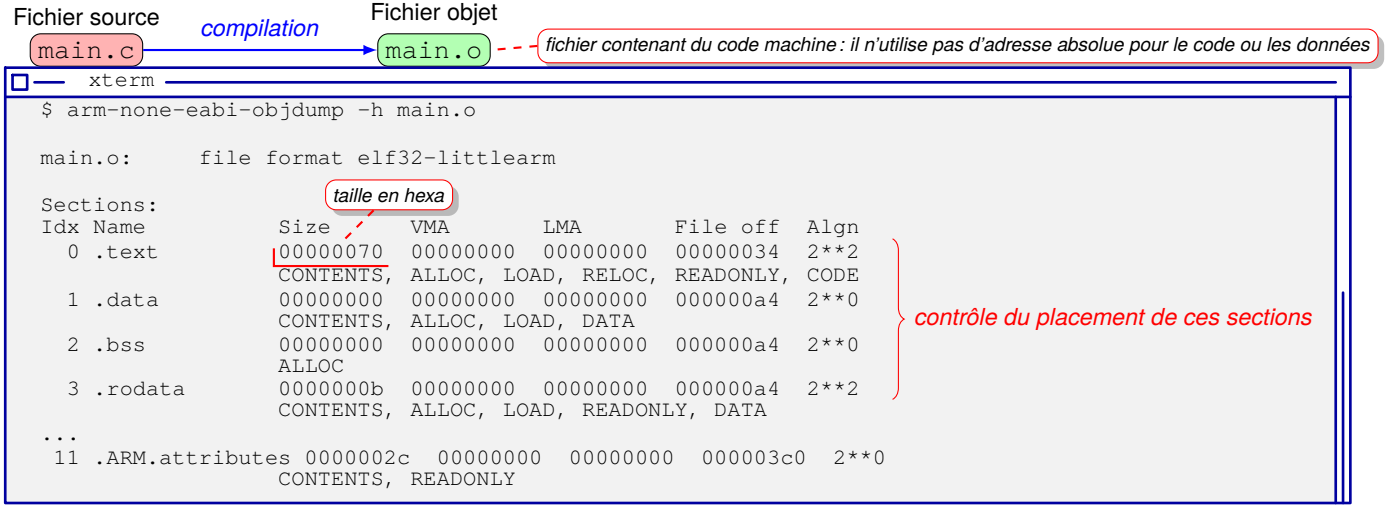
- signifie «*Executable and Linkable Format*» ;
- format standard utilisé par `gcc` pour les fichiers :
  - objets «*.o*», «*relocatable object files*» ;
  - exécutables ;

  ⟹description des éléments d'un programme : données, données en lecture seule, code, données non initialisées, *etc.*
  - organisation de ces éléments dans différentes **sections**.

    Par exemple, chaque fichier «objet» *peut contenir des données et du code qu'il faudra rassembler dans une seule sction code et une seule section données dans le fichier exécutable final.*

    Il existe d'autres formats comme celui AIF, «ARM Image Format»*.*

Qu'est-ce qu'un «*relocatable object file*»



La commande `objdump` *permet de lister les différents éléments/sections d'un objet ou d'un exécutable.*

L'outil au cœur du «*bare metal*» ?

le linker

Comment le contrôler ?

le «*linker script*»

Les différentes commandes :

- `ENTRY` : le **point d'entrée** dans l'application, c-à-d le code **exécuté en premier** après le reset du processeur.
  C'est aussi l'information utilisée par le débogueur GDB pour trouver la première fonction à exécuter.
  ⟹commande non obligatoire, mais requise pour utiliser GDB sur le fichier ELF.

- `MEMORY` : décrire les différents types de mémoire présent sur le SoC avec :
  - leur adresse de début ;
  - leur taille ;

  L'information fournie par cette commande permet au linker de calculer la taille mémoire totale utilisée pour le code, les données, la pile et le tas, ce qui lui permet de remonter une erreur en cas de dépassement de capacité.
  ⟹permet d'utiliser au mieux la mémoire disponible et permet à différentes sections d'occuper différents types de mémoire (le code en flash par exemple).

```
MEMORY
{
    name (attr) : ORIGIN = origine, LENGTH = longueur
}
```

droits — *longueur* — *nom de la région* — *adresse départ*

Exemple :

Microcontroller	Flash size (in KB)	SRAM1 size (in KB)	SRAM2 size (in KB)
STM32F4VGT6	1024	112	16

```
MEMORY
{
    FLASH (rx) : ORIGIN =0x08000000,LENGTH =1024K
    SRAM1 (rwx) : ORIGIN =0x20000000,LENGTH =112K
    SRAM2 (rwx) : ORIGIN =0x20000000+112K-4,LENGTH =16K
}
```

ORIGIN=0x20000000,LENGTH=128k

Attribute	Meaning
r	read-only sections
w	read and write sections
x	sections containing executable code
a	allocated sections
I	initialized sections

- ☐ `SECTIONS` : créer différentes sections en sortie dans l'exécutable final généré.
  - ◇ permet de fusionner différentes sections d'entrée pour produire une section en sortie ;
  - ◇ définie l'ordre des différentes sections en sortie dans le fichier ELF généré ;
  - ◇ définie le placement d'une section dans la mémoire.

    Par exemple, placer la section `.text` dans la région mémoire `FLASH` décrite dans la commande `MEMORY`. Exemple :

```
SECTIONS
{
    .text :
    {
       // merge all .isr_vector sections of all input files
       // merge all .text sections of all input files
    }>(vma) AT>(lma)

    .data :
    {
       // merge all .data sections of all input files
    }>(vma) AT>(lma)
}
```

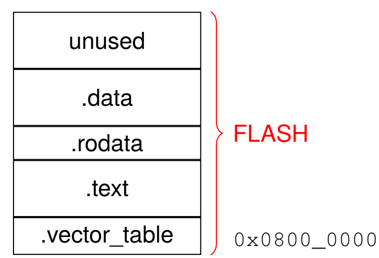
étiquette (`.text :`) — *ordre en mémoire* — *indique au locator où il doit mettre la section* (`>(vma) AT>(lma)`)

lma : «*load memory address*»

vma : «*virtual memory address*»

```
SECTIONS
{
    .text :
    {
       *(.isr_vector)
       // .text of main.o library.o startup.o
       *(.text)
       *(.rodata)
    }> FLASH AT> FLASH
}
```

désigne tous les fichiers (`*`) — ❶ (`}> FLASH AT> FLASH`)



❶⟹ ⋆ le **linker** lit le «*vma*» : il va générer des adresses absolues pour la section `.text` qui sont dans la région mémoire indiquée : `FLASH`, c-à-d à partir de l'adresse `0x08000000` ;

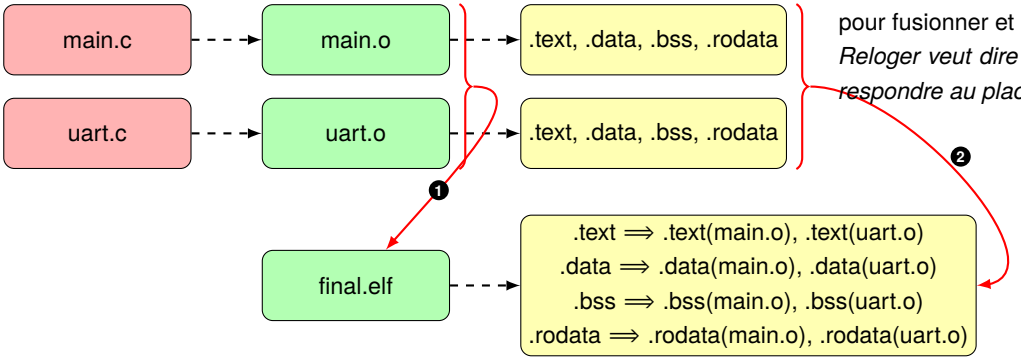
⋆ le **locator** lit le «*lma*» : il va choisir une adresse dans la région `FLASH` pour charger cette section.

Le `AT> FLASH` peut être omis si le «lma» *est identique au* «vma».

❶⇒chaque objet possède différentes sections : ces sections sont fusionnées dans l'exécutable final.

*Le «*linker script*» défini l'ordre, l'emplacement des sections.*

En général, on fusionne les sections de même nature : toutes les sections `.text`, `.data` etc.



❷⇒le «*linker*» et le «*locator*» utilise le «*linker script*» pour fusionner et reloger les différentes sections ;

Reloger veut dire que l'on réécrit les adresses pour correspondre au placement (address relocation).

```
SECTIONS
{
    .text :
    {
        *(.isr_vector)
        // .text of main.o library.o startup.o
        *(.text)
        *(.rodata)
    }> FLASH
    .data :
    {
        *(.data)
    }> SRAM AT> FLASH
    .bss
    {
        *(.bss)
    }> SRAM
}
```

❸ les **adresses absolues** seront dans la région `SRAM`, c-à-d à partir de l'adresse `0x20000000` ;

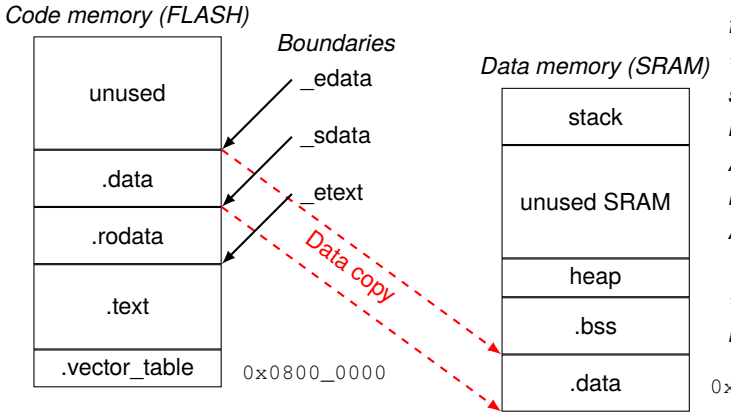
❹ la section `.data` contient des données initialisées, elle est placée dans la région `FLASH` pour être stockée dans le SoC.
⟹**copier au démarrage du micro-contrôleur le contenu de la section `.data` de la `FLASH` vers la `SRAM`.**
C'est le rôle du fichier «startup.s».

❺ la section `.bss` concerne les **données non initialisées**, ces données n'existent pas dans la flash
⟹ elles ont une «*vma*» **mais pas** de «*lma*».

Le «*location counter*»

- ▷ représenté par le symbole «.» correspond à l'adresse courante dans le «*linker script*» ;
- ▷ permet de suivre et de définir des frontières entre les différentes sections ;
- ▷ peut être fixé à une valeur choisie ;
- ▷ n'est utilisable qu'à l'intérieur de la commande `SECTIONS` ;

Le «*location counter*» est automatiquement incrémenté de la taille de section en sortie :



le transfert de la section .data de la flash vers la sram est fait dans le «startup.s»

⟹*Il faut connaître les* «frontières» *de cette section : la section .rodata commence à la fin de la section .text, ici indiqué par le symbole* `_etext` ;

Avec la taille de la section `.rodata`*, on peut déterminer le début* `_sdata` *de la section* `.data` ;

Avec la taille de `.data` *on peut déterminer sa fin* `_edata`.

⟹*les frontières* `_sdata` *et* `_edata` *peuvent être transmises à* «startup.s» *pour faire la copie.*

À chaque sortie d'une section, le linker met à jour le «*linker counter*», ce qui permet de suivre les frontières des sections.

Mais comment passer la frontière au programme ?

Un symbole est le **nom d'une adresse** :
- ▷ d'une variable globale ;
- ▷ d'une fonction ;

Le compilateur construit une **table des symboles** contenant ces adresses/symboles. Lorsque l'on utilise la variable ou la fonction dans un programme, le compilateur les remplace par l'adresse correspondante dans la table des symboles.

Pour échanger des adresses entre le «*linker script*» et le programme, on va utiliser des **symboles** qui seront utilisés par le compilateur lors de la compilation du programme.

Linker Script : un exemple complet avec copie des données d'une région à une autre

```
MEMORY
{
   FLASH (rx) : ORIGIN =0x08000000,LENGTH =1024K
   SRAM (rwx) : ORIGIN =0x20000000,LENGTH =128k
}
__max_heap_size = 0x400;  /* déclaration d'un symbole ajouté dans la table des symboles*/
__max_stack_size = 0x200;
SECTIONS
{
   .text :
   {
      *(.isr_vector)
      *(.text)
      *(.rodata)
      end_of_text = .; /* stocke la valeur courante du location counter */
   }> FLASH
   .data :
   {
      start_of_data = 0x20000000; /* assigner une valeur à un symbole */
      *(.data)
   }> SRAM AT> FLASH
}
```

au départ, . = vma, c-à-d l'emplacement de la FLASH 0x08000000

taille qui incrémente le location counter

le vma

le location counter qui est incrémenté de la taille des sections fusionnées précédentes

⟹Le symbole `end_of_text` peut être exporté vers le compilateur : **le programme peut utiliser cette adresse !**

```
SECTIONS
{
   .text :
   {
      *(.isr_vector)
      *(.text)
      *(.rodata)
      _etext = .; /* vma, c-à-d 0x0800xxxx */
   }> FLASH /* ici, vma = lma */
   .data :
   {
      _sdata = .; /* vma, c-à-d 0x20000000 */
      *(.data)
      _edata = .;
   }> SRAM AT> FLASH
```

Attention

Le «*location counter*» suit les **vma**, «*virtual memory address*» !

Pour copier :

```
// Copy .data section to SRAM
uint32_t size = &_edata - &_sdata;

uint8_t *pDst = (uint8_t*)&_sdata; //sram
uint8_t *pSrc = (uint8_t*)&_etext; //flash

memcpy(pDst, pSrc, size);
```

*il faut la **lma**,* «location memory address»

Essayons sur un nouvel exemple
plus petit
combinant de l'assembleur et du C

Le linker script en assembleur

Version simple :

```
ENTRY(_Reset)

MEMORY
{
  ram : ORIGIN = 0x00000000, LENGTH = 0x07FFFFFF
}

SECTIONS
{
  .text : { *(.text) }
  .rodata : { *(.rodata) }
  .data : { *(.data) }
  .bss  : { *(.bss)  }
  stack_top = 0x80000;
}
```

toutes les sections sont les unes à la suite des autres et la région ram est par défaut en `(rwx)`

À droite :

❶⇒on calcule directement l'emplacement de la pile à la plus haute adresse possible ;

❷⇒on utilise «.» pour enregistrer la «*vma*» de la fin du code ;

❸⇒enregistrement de la «*vma*» de début des données initialisées ;

❹⇒enregistrement de la «*vma*» de fin.

Là encore, la région est sous-entendue `ram`.

Version améliorée :

```
ENTRY(_Reset)

MEMORY
{
  ram : ORIGIN = 0x00000000, LENGTH = 0x07FFFFFF
}

❶stack_top = ORIGIN(ram)+LENGTH(ram);

SECTIONS
{
  .text : {
    *(.text)
    *(.rodata)
❷   _etext = .;
  }
  .data : {
❸   _sdata = .;
    *(.data)
❹   _edata = .;
  }
  .bss  : {
    *(.bss)
  }
}
```

On va détailler comment cette version est obtenue dans les prochains transparents.

Ici, les sections sont placées dans la région `ram` *à partir du début. La pile est placée à l'adresse la plus haute de la mémoire.*

On peut consulter la **table des symboles** de la version simple :

```
$ arm-none-eabi-objdump -t prog_write_serial.elf

prog_write_serial.elf:     file format elf32-littlearm

SYMBOL TABLE:
00000000 l    d  .text              00000000 .text
000000a8 l    d  .rodata            00000000 .rodata
00000000 l    d  .ARM.attributes    00000000 .ARM.attributes
00000000 l    d  .comment           00000000 .comment
00000000 l    d  .debug_line        00000000 .debug_line
00000000 l    d  .debug_info        00000000 .debug_info
00000000 l    d  .debug_abbrev      00000000 .debug_abbrev
00000000 l    d  .debug_aranges     00000000 .debug_aranges
00000000 l    d  .debug_str         00000000 .debug_str
00000000 l    d  .debug_frame       00000000 .debug_frame
00000000 l    df *ABS*              00000000 startup.o
00000000 l    df *ABS*              00000000 write_serial.c
00000038 g     F .text              00000054 print_uart0
00080000 g        *ABS*             00000000 stack_top❶
0000008c g     F .text              0000001c c_entry
000000a8 g     O .rodata            00000004 UART0DR❷
00000000 g        .text             00000000 _Reset
```

Le contenu de «*startup.s*» modifié :

```
.global _Reset

_Reset:
    ldr sp, =stack_top
    mov r0, #0x47
@    ldr r1, =0x101F1000❸
    ldr r1, =UART0DR    } ❹
    ldr r1, [r1]        }

    str r0, [r1]
    mov r0, #0x6F
    str r0, [r1]
    mov r0, #0x21
    str r0, [r1]
    mov r0, #0x0A
    str r0, [r1]
    bl c_entry
    bl .
```

Il existe un symbole pour la définition de l'adresse de la pile «*stack_top*» en ❶.

On voit en ❷ qu'il existe un symbole pour l'adresse de l'UART0DR dans la section `.rodata`, défini dans le fichier «*write_serial.c*» :

```
/* architecture versatilepb */
volatile unsigned int * const UART0DR = (unsigned int *) 0x101F1000;
```

On peut réécrire le code de «*startup.s*», en remplaçant le chargement direct de l'adresse de l'UART en ❸ par le chargement du symbole en ❹.

⟹le code de «*startup.s*» devient adaptable...

Exemple avec la plateforme «*versatilepb*» de QEMU : obtenir la «*memory map*»

Lors de la compilation, on peut demander la création d'un rapport sur la «*memory*»

```
prog_write_serial.elf: startup.o write_serial.o
	$(CROSS_PREFIX)ld -g -T prog_write_serial.ld -Map=memoire.map❶ $^ -o $@
```

L'option `-Map` en ❶ est passée au linker pour générer la «*memory map*» :

```
$ cat memoire.map
Memory Configuration

Name             Origin             Length             Attributes
ram              0x0000000000000000 0x00000000007fffff
*default*        0x0000000000000000 0xffffffffffffffff

Linker script and memory map

.text            ❶0x0000000000000000      ❷0xac
 *(.text)
 .text           0x0000000000000000       0x3c startup.o
                 0x0000000000000000                _Reset
 .text           0x000000000000003c       0x70 write_serial.o
                 0x000000000000003c                print_uart0
                 0x0000000000000090                c_entry
...
.rodata          0x00000000000000ac        0xb
 *(.rodata)
 .rodata         0x00000000000000ac        0xb write_serial.o
                 0x00000000000000ac                ❸UART0DR
...
.data            0x00000000000000b7        0x0
 *(.data)
 .data           0x00000000000000b7        0x0 startup.o
 .data           0x00000000000000b7        0x0 write_serial.o
...
.bss             0x00000000000000b7        0x0
 *(.bss)
 .bss            0x00000000000000b7        0x0 startup.o
 .bss            0x00000000000000b7        0x0 write_serial.o
                 0x0000000000080000                ❹stack_top = 0x80000
...
```

❶⇒L'adresse de la section ;

❷⇒sa taille ;

❸⇒la variable globale de «*write_serial.c*» qui est devenue un symbole ;

C'est pour cela que dans le fichier «*startup.s*» :

```
@ on charge le symbole
@ = l'adresse de la variable
ldr r1, =UART0DR
@ on charge son contenu,
@ c-à-d l'adresse de l'UART
ldr r1, [r1]
```

❹⇒la définition de la pile.

Exemple avec la plateforme «*versatilepb*» de QEMU : obtenir les positions des sections

On peut utiliser le «.» pour connaitre l'adresse de fin de la section de code, `.text` et de données en lecture seule `.rodata`:

```
ENTRY(_Reset)
MEMORY
{
  ram : ORIGIN = 0x00000000, LENGTH = 0x07FFFFFF
}
stack_top = 0x80000;

SECTIONS
{
```

```
.text : {
  *(.text)
  *(.rodata)
  _etext = .;
}
.data : {
  *(.data)
}
.bss  : {
  *(.bss)
}
}
```

```
$ cat memoire.map

Memory Configuration

Name             Origin             Length             Attributes
ram              0x0000000000000000 0x0000000007ffffff
*default*        0x0000000000000000 0xffffffffffffffff

Linker script and memory map

                 0x0000000000080000                stack_top = 0x80000

.text            0x0000000000000000       0xb7
 *(.text)
 .text           0x0000000000000000       0x3c startup.o
                 0x0000000000000000                _Reset
 .text           0x000000000000003c       0x70 write_serial.o
                 0x000000000000003c                print_uart0
                 0x0000000000000090                c_entry
 *(.rodata)
 .rodata         0x00000000000000ac        0xb write_serial.o
                 0x00000000000000ac                UART0DR
                 0x00000000000000b7                _etext = .
...
```

Le symbole «*_etext*» ❶ nous donne l'adresse «*vma*» de fin du code : ici `0xb7`.

Ajoutons des **données initialisées** dans le fichier «*write_serial.c*» :

```
#include <stdint.h>
volatile unsigned int * const UART0DR = (unsigned int *) 0x101F1000;

const uint32_t const_v_1 = 1;

void print_uart0(const char *s) {
    while(*s != '\0') {          /* Loop until end of string */
         *UART0DR = (unsigned int)(*s); /* Transmit char */
          s++;                   /* Next char */
    }
}
void c_entry() {
     print_uart0("Yop !\n");
}
```

❶⟹une variable globale constante
⟹ section «*.rodata*».

```
$ cat memoire.map
Memory Configuration

Name             Origin             Length             Attributes
ram              0x0000000000000000 0x00000000007fffff
*default*        0x0000000000000000 0xffffffffffffffff

Linker script and memory map

                 0x0000000000080000                stack_top = 0x80000
.text            0x0000000000000000        0xbb
 *(.text)
 .text           0x0000000000000000        0x3c startup.o
                 0x0000000000000000                _Reset
 .text           0x000000000000003c        0x70 write_serial.o
                 0x000000000000003c                print_uart0
                 0x0000000000000090                c_entry
 *(.rodata)
 .rodata         0x00000000000000ac         0xf write_serial.o
                 0x00000000000000ac                UART0DR
                 0x00000000000000b0                const_v_1
                 0x00000000000000bb                _etext = .
```

❷⟹notre variable
❸⟹le symbole `_etext` est décalé de 4 octets (un entier sur 32bits).
⟹`_etext = 0xbb`

Exemple avec la plateforme «*versatilepb*» de QEMU : aligner les positions des sections

Si on ajoute plusieurs variables, dont une sur 8bits :

```
volatile unsigned int * const UART0DR = (unsigned int *) 0x101F1000;

const uint32_t const_v_1 = 1;
const uint32_t const_v_2 = 1;
const uint8_t const_v_3 = 1;

void print_uart0(const char *s) {
    while(*s != '\0') {          /* Loop until end of string */
        *UART0DR = (unsigned int)(*s); /* Transmit char */
...
```

❶⟹ici on a un entier sur 8bits

```
*(.rodata)
.rodata         0x00000000000000ac       0x17 write_serial.o
                0x00000000000000ac                UART0DR
                0x00000000000000b0                const_v_1
                0x00000000000000b4                const_v_2
                0x00000000000000b8                const_v_3
                0x00000000000000c3                _etext = .
```

❷⟹la fin de la section `.rodata` est à l'adresse `c3`
⟹`0xc3 = 195` qui n'est pas divisible par 4 !

On peut «*aligner*» les données en mettant à jour le «*location counter*» à un multiple de 4 :

```
SECTIONS
{
  .text : {
    *(.text)
    *(.rodata)
    . = ALIGN(4);
    _etext = .;
  }

  .data : {
```

```
*(.rodata)
.rodata         0x00000000000000ac       0x17 write_serial.o
                0x00000000000000ac                UART0DR
                0x00000000000000b0                const_v_1
                0x00000000000000b4                const_v_2
                0x00000000000000b8                const_v_3
                0x00000000000000c4                . = ALIGN (0x4)
*fill*          0x00000000000000c3       0x1
                0x00000000000000c4                _etext = .
```

Avec l'opération `ALIGN()` en ❸ :

- ▷ des octets de remplissage, «*fill*», ici 1 octet ❹, sont ajoutés ;
- ▷ la fin de la section indiquée par `_etext` passe à `0xc4` qui est un multiple de 4.

*L'alignement sur la frontière d'un mot, «*word*», est important pour des processeurs 32bits.*

Exemple avec la plateforme «*versatilepb*» de QEMU : remplir la section `.data`

Ajoutons une variable initialisée mais **modifiable** :

```
#include <stdint.h>

/* architecture versatilepb */
volatile unsigned int * const UART0DR = (unsigned int *) 0x101F1000;

const uint32_t const_v_1 = 1;
const uint32_t const_v_2 = 1;
const uint8_t  const_v_3 = 1;

char *chaine = "Ceci est une chaine.\n";

void print_uart0(const char *s) {
    while(*s != '\0') {          /* Loop until end of string */
        *UART0DR = (unsigned int)(*s); /* Transmit char */
         s++;                    /* Next char */
    }
}
void c_entry() {
    print_uart0("Yop !\n");
    print_uart0(chaine);
}
```

❶ ⟹ La variable `chaine` est globale mais non constante.
❷ On demande ensuite son affichage.

Ce qui donne dans la «*memory map*» :

```
.data           0x00000000000000f0        0x4
 *(.data)
 .data          0x00000000000000f0        0x0 startup.o
 .data          0x00000000000000f0        0x4 write_serial.o
                0x00000000000000f0                chaine
.bss            0x00000000000000f4        0x0
 *(.bss)
```

Le symbole `chaine` ❸ est situé en `0xf0` et fait une taille de 4 octets ce qui est normal pour un **pointeur**.

```
xterm
$ qemu-system-arm -M versatilepb -nographic -kernel prog_write_serial.elf -serial mon:stdio
Go!
Yop !
Ceci est une chaine.
```

Exemple avec la plateforme «*versatilepb*» de QEMU : definir deux régions mémoire

On va diviser la mémoire en **deux régions** pour simuler une **région de flash** et une **région de ram** :

```
ENTRY(_Reset)
MEMORY
{
   ram : ORIGIN = 0x00000000, LENGTH = 64K
   flash : ORIGIN = 0x00020000, LENGTH = 64K
}
stack_top = ORIGIN(ram)+LENGTH(ram);
SECTIONS
{
   .text : {
     *(.text)
     *(.rodata)
     . = ALIGN(4);
     _etext = .;
   } > ram
   .data : {
     *(.data)
   } >flash AT> ram
   .bss  : {
     *(.bss)
   } >ram
}
```

❶⟹définition de la région `flash`

La section `.data` se trouve en «*vma*» ❷ dans la région `flash` et «*lma*» ❸ dans région `ram`.

❹⟹la section `.data` se trouve maintenant dans la région `flash` située à partir de l'adresse `0x20000`

```
*(.rodata)
.rodata         0x00000000000000c0        0x2f write_serial.o
...
                0x00000000000000cc                const_v_3
                0x00000000000000f0                . = ALIGN (0x4)
*fill*          0x00000000000000ef         0x1
                0x00000000000000f0                _etext = .
.data           0x0000000000020000         0x4 load address 0x00000000000000f0
*(.data)
.data           0x0000000000020000         0x0 startup.o
.data           0x0000000000020000         0x4 write_serial.o
                0x0000000000020000                chaine
.bss            0x00000000000000f4         0x0
*(.bss)
.bss            0x00000000000000f4         0x0 startup.o
.bss            0x00000000000000f4         0x0 write_serial.o
```

❺⟹la section `.bss` se trouve après la section `.data` dans la région `ram`, soit en `0xf4`, qui correspond bien à l'adresse de fin donnée par le symbole `_etext` + les 4octets nécessaires à l'adresse du symbole `chaine`.

Exemple avec la plateforme «*versatilepb*» de QEMU : deux régions mémoire et copie des données

Et si on exécute ?

xterm

```
$ qemu-system-arm -M versatilepb -nographic -kernel prog_write_serial.elf -serial mon:stdio
Go!
Yop !
,G
```

ça affiche n'importe quoi !

Pourquoi ?

Parce que les données sont dans la région `flash` et pas dans `ram` ! ⟹**Il faut copier les données de la flash dans la ram !**

```
ENTRY(_Reset)
MEMORY
{
 ram : ORIGIN =0x00000000, LENGTH =64K
 flash : ORIGIN =0x00020000, LENGTH =64K
}
stack_top = ORIGIN(ram)+LENGTH(ram);
SECTIONS
{
  .text : {
    *(.text)
    *(.rodata)
    . = ALIGN(4);
    _etext = .;
  } > ram
  .data : {
    _sdata = .;
    *(.data)
    _edata = .;
  } >flash AT> ram
  .bss  : {
    *(.bss)
  } >ram
}
```

mémorisation adresse

mémorisation adresse

mémorisation adresse

xterm

```
Go!
Yop !
Ceci est une chaine.
```

ça marche !

```
#include <stdint.h>

volatile unsigned int * const UART0DR =
                          (unsigned int *) 0x101F1000;

const uint32_t const_v_1 = 1;
const uint32_t const_v_2 = 1;
const uint8_t const_v_3 = 1;
char *chaine = "Ceci est une chaine.\n";

extern uint32_t _edata;
extern uint32_t _sdata;
extern uint32_t _etext;

void print_uart0(const char *s) {
    while(*s != '\0') { *UART0DR = (unsigned int)(*s);
                        s++;  }
}
void c_entry() {
  uint32_t size = (uint32_t)&_edata - (uint32_t)&_sdata;
  uint8_t *pDst = (uint8_t*)&_sdata; //ram
  uint8_t *pSrc = (uint8_t*)&_etext; //flash

  for(int i = 0; i<size; i++)
     *pDst++ = *pSrc++;
  print_uart0("Yop !\n");
  print_uart0(chaine);
}
```

on accède aux symboles

on copie de la flash dans la ram !

Mais où se trouvent les chaînes prédéfinies ?

xterm

```
$ arm-none-eabi-objdump -s prog_write_serial.elf

prog_write_serial.elf:     file format elf32-littlearm

Contents of section .text:
 0000 2cd09fe5 4700a0e3 28109fe5 001091e5  ,...G...(.......
 0010 000081e5 6f00a0e3 000081e5 2100a0e3  ....o.......!...
 0020 000081e5 0a00a0e3 000081e5 170000eb  ................
 0030 fefffffb 00000100 3c010000 04b02de5  ........<.....-.
 0040 00b08de2 0cd04de2 08000be5 060000ea  ......M.........
 0050 08301be5 0020d3e5 2c309fe5 002083e5  .0... ..,0... ..
 0060 08301be5 013083e2 08300be5 08301be5  .0...0...0...0..
 0070 0030d3e5 000053e3 f4ffff1a 0000a0e1  .0....S.........
 0080 00d08be2 04b09de4 1eff2fe1 00101f10  ........../.....
 0090 00482de9 04b08de2 10d04de2 84209fe5  .H-.......M.. ..
 00a0 84309fe5 033042e0 14300be5 78309fe5  .0...0B..0..x0..
 00b0 08300be5 74309fe5 0c300be5 0030a0e3  .0..t0...0...0..
 00c0 10300be5 0a0000ea 0c201be5 013082e2  .0....... ...0..
 00d0 0c300be5 08301be5 011083e2 08100be5  .0...0..........
 00e0 0020d2e5 0020c3e5 10301be5 013083e2  . ... ...0...0..
 00f0 10300be5 10301be5 14201be5 030052e1  .0...0... ....R.
 0100 f0ffff8a 28009fe5 cbffffeb 24309fe5  ....(.......$0..
 0110 003093e5 0300a0e1 c7ffffeb 0000a0e1  .0..............
 0120 04d04be2 0088bde8 04000200 00000200  ..K.............
 0130 6c010000 64010000 00000200 00101f10  l...d...........
 0140 01000000 01000000 01000000 43656369  ............Ceci
 0150 20657374 20756e65 20636861 696e652e   est une chaine.
 0160 0a000000 596f7020 210a0000           ....Yop !...
```

Elles sont à la fin de la section `.code`.

La **variable globale** `chaine` ne contient que le pointeur sur l'adresse de la chaîne.

⟹ *la copie de la flash dans la ram ne travaille que sur 4 octets !*

Et le lien entre Assembleur

&

ABI, «*Application Binary Interface*» ?

Suivi d'exécution avec GDB d'un code C compilé vers ARM

Soit le code suivant contenu dans le fichier «ma_fonction.c»:

```
int ma_fonction(int a,int b)
{
  a=54+b;
  return b;
}
```

```
xterm
$ arm-none-eabi-gcc -c ma_fonction.c
$ arm-none-eabi-nm ma_fonction.o
00000000 T ma_fonction
```

Si on le désassemble :

```
xterm
$ arm-none-eabi-objdump -d ma_fonction.o
ma_fonction.o:     file format elf32-littlearm

Disassembly of section .text:
00000000 <ma_fonction>:
   0:   e52db004        push    {fp}            ; (str fp, [sp, #-4]!)
   4:   e28db000        add     fp, sp, #0
   8:   e24dd00c        sub     sp, sp, #12
   c:   e50b0008        str     r0, [fp, #-8]
  10:   e50b100c        str     r1, [fp, #-12]
  14:   e51b300c        ldr     r3, [fp, #-12]
  18:   e2833036        add     r3, r3, #54     ; 0x36
  1c:   e50b3008        str     r3, [fp, #-8]
  20:   e51b300c        ldr     r3, [fp, #-12]
  24:   e1a00003        mov     r0, r3
  28:   e28bd000        add     sp, fp, #0
  2c:   e49db004        pop     {fp}            ; (ldr fp, [sp], #4)
  30:   e12fff1e        bx      lr
```

on sauve le «stack pointer» *dans le* «frame pointer»

sauvegarde de la «frame»

*on fait de la place pour 3 words (3*4octets)*

on sauvegarde r0 et r1 sur la pile

on mets dans r3, la valeur de r1 qui est arg1, c-à-d «b»

on augmente r3 de 54

on stocke r3 dans la pile où on a mis r0, c-à-d arg0 ou «a»

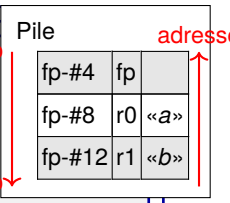
on mets dans r3 la valeur de r1, c-à-d arg1 ou «b»

restaure fp

restaure sp

retourne

«b» *en valeur de retour*



Si on active **l'optimisation** :
La fonction ne fait rien d'utile, son contenu est effacé.

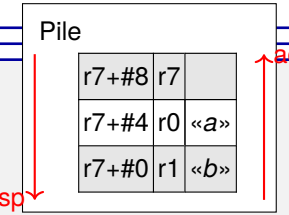
```
xterm
$ arm-none-eabi-gcc -O3 -c ma_fonction.c
$ arm-none-eabi-objdump -d ma_fonction.o
ma_fonction.o:     file format elf32-littlearm

Disassembly of section .text:
00000000 <ma_fonction>:
   0:   e1a00001        mov     r0, r1
   4:   e12fff1e        bx      lr
```

mettre b en valeur de retour

Suivi d'exécution avec GDB : utilisation du mode «*thumb*»

Les instructions du mode «*Thumb*» sont sur 16bits :

```
xterm
$ arm-none-eabi-gcc -mcpu=cortex m4 -mthumb -c ma_fonction.c
$ arm-none-eabi-objdump -d ma_fonction.o
ma_fonction.o:     file format elf32-littlearm

Disassembly of section .text:

00000000 <ma_fonction>:
   0:   b480            push    {r7}
   2:   b083            sub     sp, #12
   4:   af00            add     r7, sp, #0
   6:   6078            str     r0, [r7, #4]
   8:   6039            str     r1, [r7, #0]
   a:   683b            ldr     r3, [r7, #0]
   c:   3336            adds    r3, #54 ; 0x36
   e:   607b            str     r3, [r7, #4]
  10:   683b            ldr     r3, [r7, #0]
  12:   4618            mov     r0, r3
  14:   370c            adds    r7, #12
  16:   46bd            mov     sp, r7
  18:   bc80            pop     {r7}
  1a:   4770            bx      lr
```



sauvegarde de r7 qui va servir de «frame pointer»

réservation de 3 words dans la pile

on mets sp dans r7

on sauve r0, «a», dans r7+4, en position 2 de la pile

on sauve r1, «b», dans r7, en position 1

r7+54

on charge «b» *dans r7*

r7=>a

on charge «b» *dans r3*

on r3 dans r0 comme valeur de retour

on restaure fp

on restaure sp

on retourne

Si on **optimise** la compilation, on obtient un code similaire mais en instructions 16bits :

```
xterm
$ arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -O3 -c ma_fonction.c
$ arm-none-eabi-objdump -d ma_fonction.o
ma_fonction.o:     file format elf32-littlearm

Disassembly of section .text:

00000000 <ma_fonction>:
   0:   4608            mov     r0, r1
   2:   4770            bx      lr
```

mettre b en valeur de retour

Utilisation du simulateur unicorn avec gdb au travers de udbserver

xterm

```
$ arm-none-eabi-gcc -g -c ma_fonction.c
$ arm-none-eabi-objcopy -O binary ma_fonction.o rom.bin
$ cargo run --example server rom.bin
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/examples/server rom.bin`
Waiting for a GDB connection on "0.0.0.0:1234"...
Debugger connected from 127.0.0.1:43486
```

serveur

- ▷ On compile le code précédent et on extrait les octets de code ARM.
- ▷ On lance `udbserver` qui combine **unicorn**, un émulateur basé sur Qemu, avec un «*gdb server*» en rust.

On se connecte ensuite avec `gdb-multiarch`, pour faire une pseudo-exécution :

xterm

```
$ gdb-multiarch -q -ex 'target extended-remote :1234'
Remote debugging using :1234
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x00001000 in ?? ()
 Assembly
 0x00001000  ? push   {r11}      @ (str r11, [sp, #-4]!)
 0x00001004  ? add    r11, sp, #0
 0x00001008  ? sub    sp, sp, #12
 0x0000100c  ? str    r0, [r11, #-8]
 0x00001010  ? str    r1, [r11, #-12]
 0x00001014  ? ldr    r3, [r11, #-8]
 0x00001018  ? add    r3, r3, #54   @ 0x36
 0x0000101c  ? str    r3, [r11, #-12]
 0x00001020  ? ldr    r3, [r11, #-8]
 0x00001024  ? mov    r0, r3

 Registers
     r0 0x00000000    r1 0x00000000    r2 0x00000000    r3 0x00000000     r4 0x00000000   r5 0x00000000
     r6 0x00000000    r7 0x00000000    r8 0x00000000    r9 0x00000000    r10 0x00000000  r11 0x00000000
    r12 0x00000000    sp 0x00003000    lr 0x00001000    pc 0x00001000   cpsr 0x400001d3


[1] id 1 from 0x00001000
>>> set $r0=0x10
>>> set $r1=0x20
>>> set $r11=$sp
>>> x/4wx $sp-12
```

connexion

udbserver est un frontal gdb-server pour unicorn.
⟹ on peut s'y connecter avec gdb pour contrôler l'exécution dans unicorn.

charger des valeurs dans les registres

afficher 4 words (32bits chacun) en hexadécimal

L'affichage de 32bits (4 octets) :
- ▷ «w» *est en* «big endian» *;*
- ▷ «4b» *est en* «little endian».

Si on poursuit l'exécution d'une instruction qui sauvegarde le registre `r11` sur la pile :

```
0x00001000  ? push   {r11}      @ (str r11, [sp, #-4]!)
```

On a alors :

xterm

```
 Output/messages
0x00001004 in ?? ()
 Assembly
 0x00001004  ? add   r11, sp, #0
 0x00001008  ? sub   sp, sp, #12
 0x0000100c  ? str   r0, [r11, #-8]
 0x00001010  ? str   r1, [r11, #-12]
 0x00001014  ? ldr   r3, [r11, #-8]
 0x00001018  ? add   r3, r3, #54   @ 0x36
 0x0000101c  ? str   r3, [r11, #-12]
 0x00001020  ? ldr   r3, [r11, #-8]
 0x00001024  ? mov   r0, r3
 0x00001028  ? add   sp, r11, #0

 Registers
     r0 0x00000010   r1 0x00000020    r2 0x00000000    r3 0x00000000    r4 0x00000000   r5 0x00000000
     r6 0x00000000   r7 0x00000000    r8 0x00000000    r9 0x00000000   r10 0x00000000  r11 0x00003000
     r12 0x00000000  sp 0x00002ffc    lr 0x00001000    pc 0x00001004  cpsr 0x400001d3

[0] from 0x00001004
 Threads
[1] id 1 from 0x00001004
>>> x/wx $sp
0x2ffc:   0x00003000
>>> x/4b $sp
0x2ffc:   0x00   0x30   0x00   0x00
>>> bt
#0  0x00001004 in ?? ()
>>> info frame
Stack level 0, frame at 0x2ffc:
 pc = 0x1004; saved pc = <not saved>
 Outermost frame: outermost
 Arglist at 0x2ffc, args:
 Locals at 0x2ffc, Previous frame's sp is 0x2ffc
```

on est passé à l'adresse de l'instruction suivante

on est passé à l'adresse de l'instruction suivante

affichage du contenu de la pile en «big-endian»

affichage du contenu de la pile en «little-endian»

le débogueur a du mal à retrouver les infos de la «frame»
⟹*on est en simulation d'une fonction sans son appelant*