Microcontrôleur

```
wspr_beacon.cpp  Makefile  ad9850.cpp  ad9850.h  cw.cpp  cw.h
Source  History
53      delay(5);
54      digitalWrite(LOAD, LOW);
55      // Chip is RESET now
56  }
57
58  void AD9850_set_frequency(unsigned long frequency)
59  {
60      unsigned long tuning_word = (frequency * pow(2, 32)) / DDS_CLOCK;
61      digitalWrite (LOAD, LOW);
62
63      shiftOut(DATA, CLOCK, LSBFIRST, tuning_word);
64      shiftOut(DATA, CLOCK, LSBFIRST, tuning_word >> 8);
65      shiftOut(DATA, CLOCK, LSBFIRST, tuning_word >> 16);
66      shiftOut(DATA, CLOCK, LSBFIRST, tuning_word >> 24);
67      shiftOut(DATA, CLOCK, LSBFIRST, 0x0);
68      digitalWrite (LOAD, HIGH);
69  }
70
```

CoursMicroCNAM.v1.0.odp

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Différents types de mémoire

- "famille **RAM**" (random access memory) : **DRAM**, **SRAM**

  Mémoire volatile : alimentation nécessaire pour la rétention des données stockées.

  Accès rapide en lecture et en écriture : (<10 ns)

- "famille **ROM**" (read only memory) : **Flash, EEPROM**

  Mémoire non volatile (permanente) : rétention des données après la coupure de l'alimentation

  Accès : **rapide en lecture** (< 10 ns)
  impossible ou lent en écriture (~ ms)

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

La mémoire

Adresses (ici 16bits)	Données (ici 8bits)
0xFFFF	10011001
0xFFFE	00001000
0xFFFD	01111111
0xFFFC	01111001
0x0003	01001111
0x0002	00001011
0x0001	11001011
0x0000	01001001

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Microprocesseur

On caractérise un microprocesseur par :

son architecture (organisation interne),

sa fréquence d'horloge : en MHz ou Ghz

sa puissance de calcul :
c'est le nombre d'instructions par secondes
qu'il est capable d'exécuter : Mips (10^6 instructions/s)
Mflops (10^6 ins sur float /s)

la taille de son bus de données : en bits

Il existe des microprocesseurs 8, 16, 32 ou 64 bits.

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

L'horloge

L'horloge cadence toutes les activités du microprocesseur.
Sa période fixe la vitesse d'exécution des instructions.
Pour une architecture donnée :
Fréquence horloge élevée = Grande puissance de calcul

Mais

La ***consommation*** augmente avec la fréquence de l'horloge.
Le ***prix*** d'un µP est très lié à la fréquence d'horloge.

Dans des applications alimentées par batteries
ou à faible coût, il n'est pas toujours souhaitable d'utiliser une fréquence d'horloge élevée.
On peut faire varier la fréquence d'horloge en fonction des phases de fonctionnement de l'application.

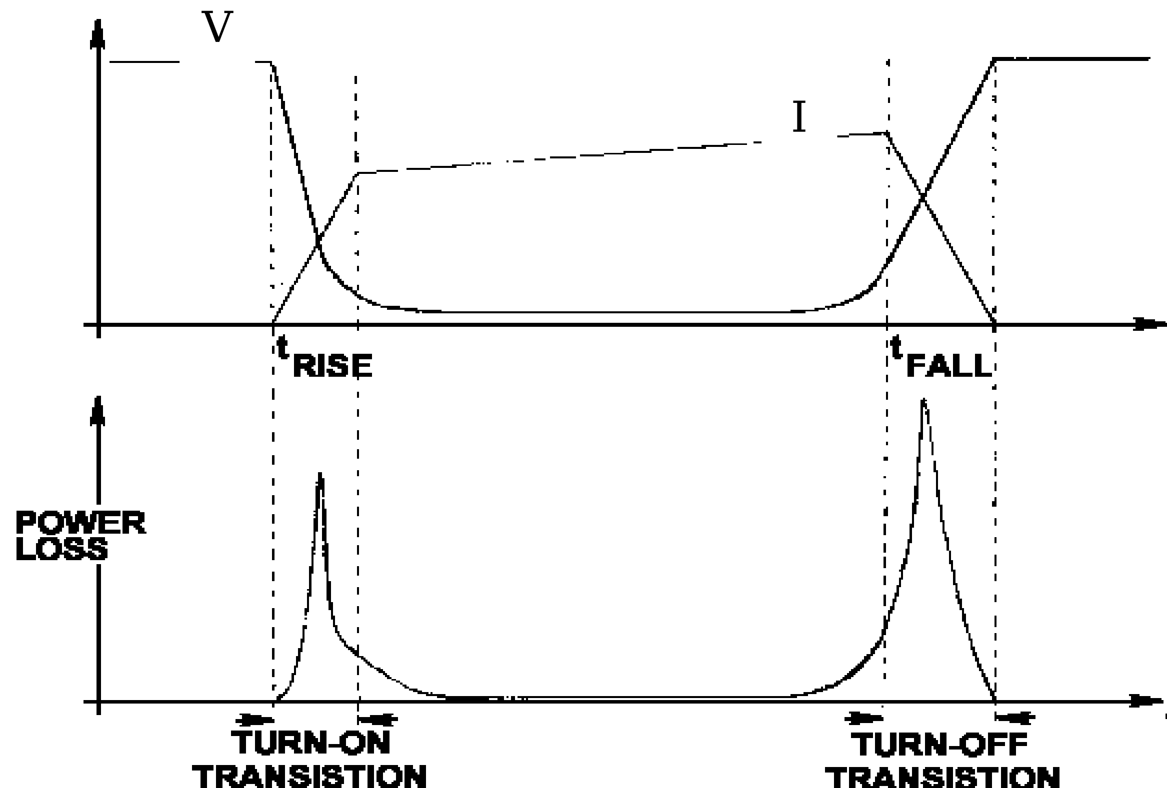
© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Le problème de l'échauffement

La puissance consommée par le processeur dépend de sa tension de fonctionnement et sa fréquence d'horloge :

$$\mathbf{P \approx k \cdot V^2 \cdot F}$$

=> diminuer la tension de fonctionnement du coeur
=> utiliser la fréquence horloge minimale permettant d'obtenir les performances souhaitées.



© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Constitution d'un microprocesseur

- Un microprocesseur (CPU) se compose :
- d'une *unité de commande* qui :
  - lit l'instruction en mémoire
  - la décode (prépare les opérations suivantes)
  - prépare la lecture de l'instruction suivante
- d'une *unité de traitement* qui :
  - exécute l'instruction (ALU)
  - met à jour les registres internes en fonction du résultat
  - sauve éventuellement les résultats en mémoire

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Le compteur programme

PC : program counter (compteur ordinal)

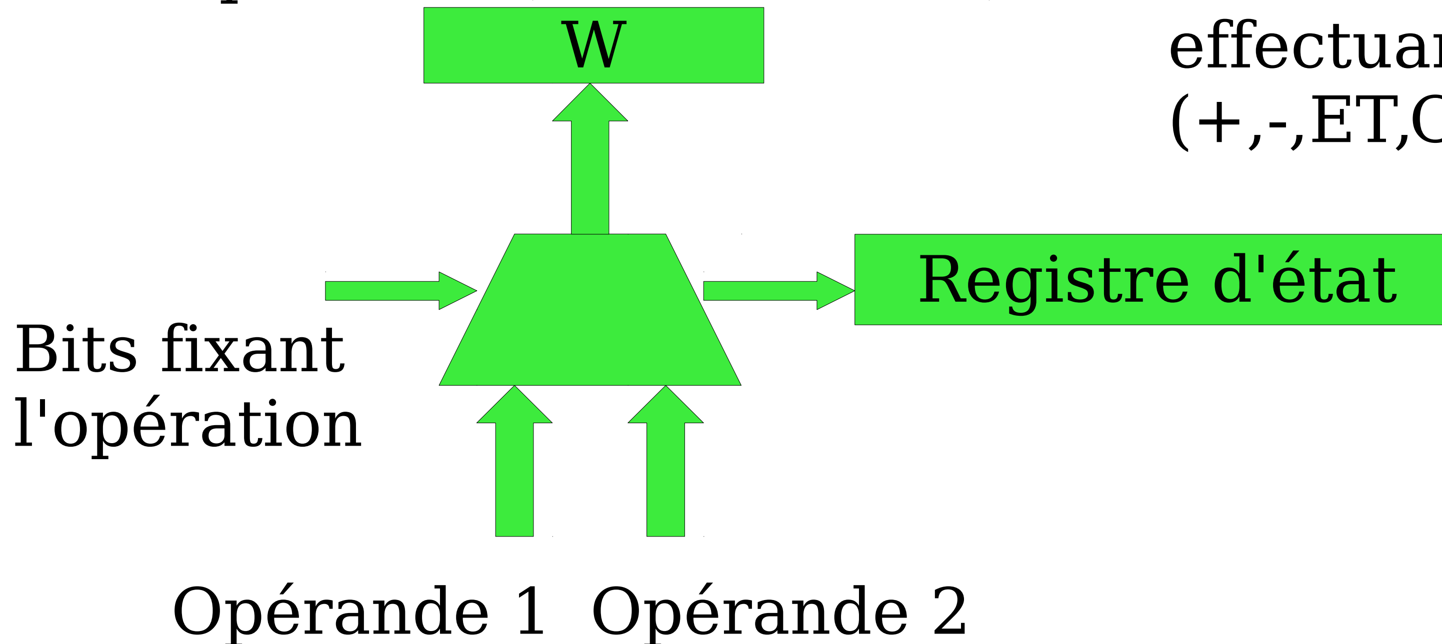
C'est un registre du µP qui contient l'adresse en mémoire
de l'instruction à exécuter.

Au départ (au RESET), il prend une valeur fixe.
(généralement la première ou la dernière adresse de l'espace mémoire)

A chaque instruction , il est incrémenté pour contenir l'adresse de l'instruction suivante.
En cas de branchement (if/else, boucle, appel de fonction),
il change de valeur pour pointer l'instruction suivante qui n'est pas nécessairement la suivante en mémoire.

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Unité arithmétique et logique : ALU



L'ALU est la partie du µP effectuant les opérations (+,-,ET,OU,XOR,....)

Le type d'opération est déterminé par des bits issus du code l'instruction
Les opérandes peuvent être issus du code de l'instruction ou bien lus en mémoire

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Registre d'état

STATUS REGISTER

U-0	U-0	U-0	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
—	—	—	N	OV	Z	DC	C
bit 7							bit 0

bit N : indique si le résultat est <0
bit OV : dépassement de capacité
bit Z : indique si le résultat est 0
bit DC : retenue sur le 4ème bit
bit C : retenue sur le MSB

Ces bits sont utilisés par les instructions conditionnelles pour déterminer les prochaines instructions exécutées.

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Instructions

Une instruction réalise une **opération élémentaire**.
(addition, soustraction, test d'un bit, écriture mémoire,etc...)

Elle est codée par un ou plusieurs mots en mémoire.

Son format est spécifique au type de microprocesseur.

Les instructions élémentaires ne sont pas compatibles entres des processeurs différents =>
Un exécutable est liée à un type de machine (et à un OS).

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Instructions courantes

•Un microprocesseur possède généralement des instructions :

- de **lecture et d'écriture dans la mémoire** (move)
  entre mémoire et registre interne
  (parfois entre deux cases mémoires)

- de **calculs arithmétiques et logiques**
  addition, soustraction, ET, OU, décalage de bit,....
  (parfois multiplication, division,...)

- de **branchement et de saut (conditionnel ou non)**
ces instructions permettent la réalisation d'opération conditionnelle (if/else, boucles)
et les appels de fonctions (saut à une adresse et retour)

- spéciales (liées aux périphériques), mise en veille, etc...

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Instructions (exemple)

ADDLW — **ADD literal to W**

Syntax:	[*label*] ADDLW k
Operands:	$0 \leq k \leq 255$
Operation:	$(W) + k \rightarrow W$
Status Affected:	N, OV, C, DC, Z
Encoding:	`0000` `1111` `kkkk` `kkkk`
Description:	The contents of W are added to the 8-bit literal 'k' and the result is placed in W.
Words:	1
Cycles:	1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process Data	Write to W

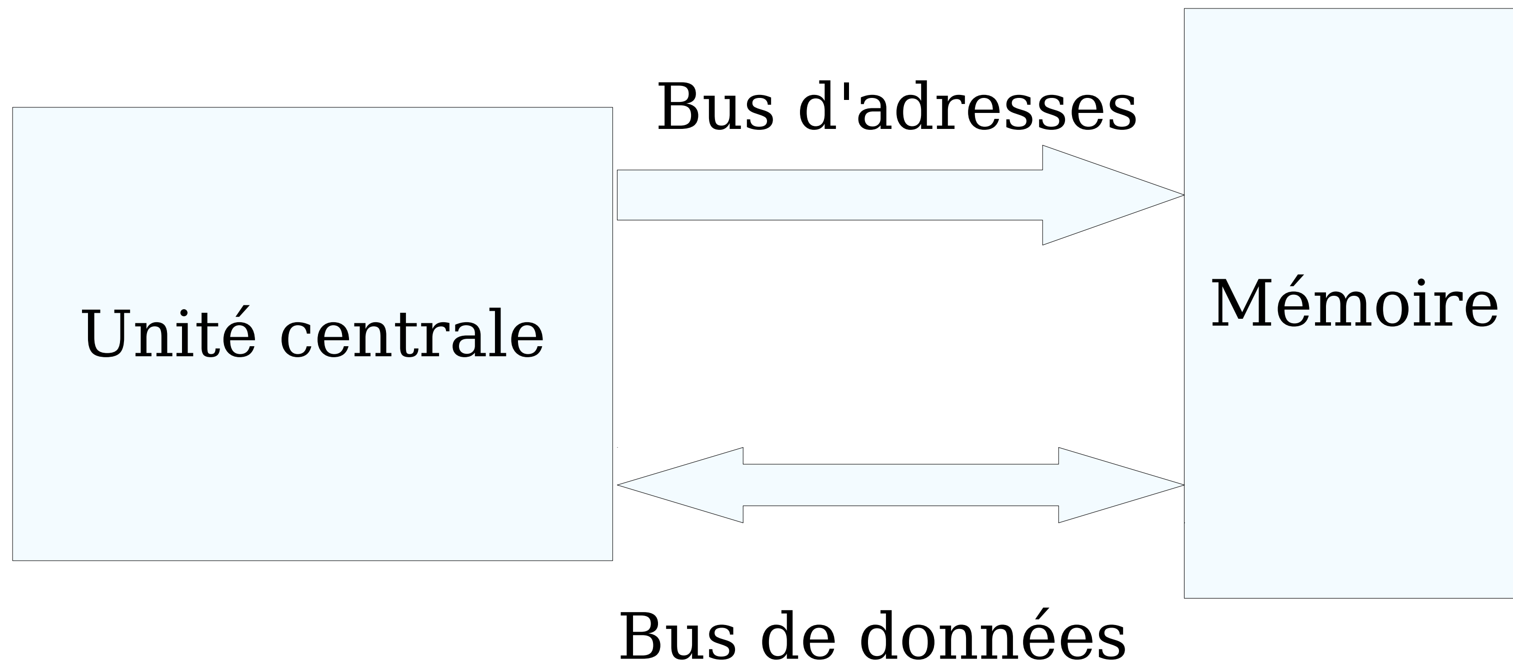
Example: `ADDLW 0x15`

Before Instruction
W = 0x10
After Instruction
W = 0x25

Le code binaire de cette instruction est 0x0F15

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

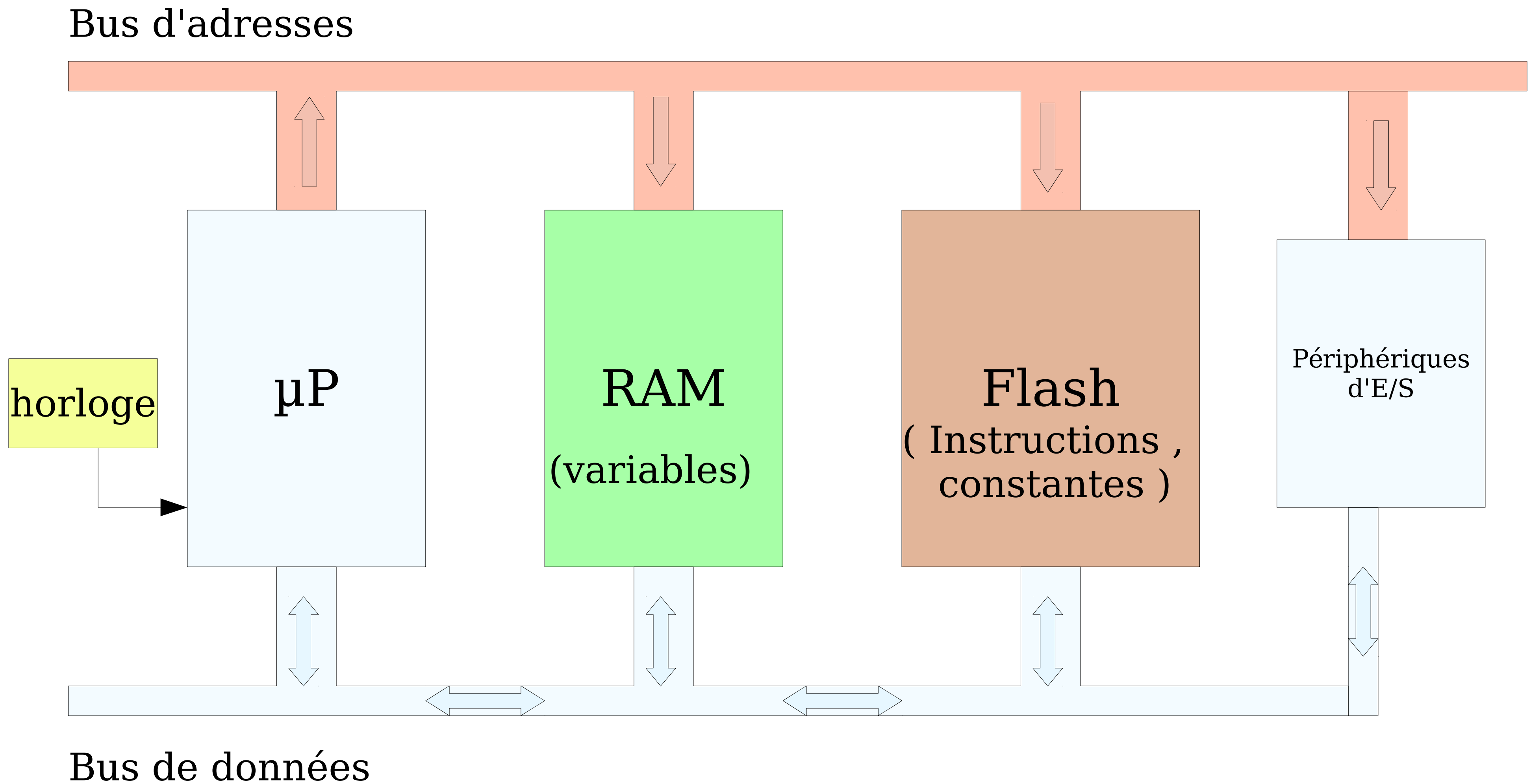
Architecture de Von Neuman



Le même espace mémoire est utilisée pour stocker des données et des instructions

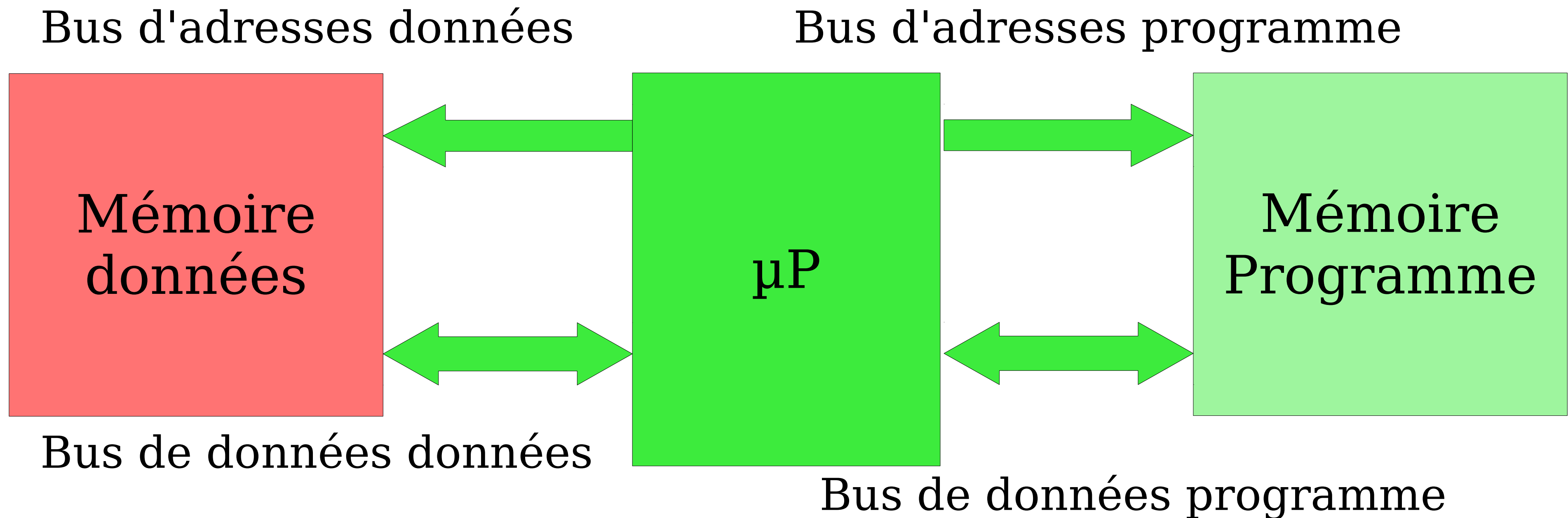
© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Système minimum (Von Neuman)



© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Architecture de Harvard

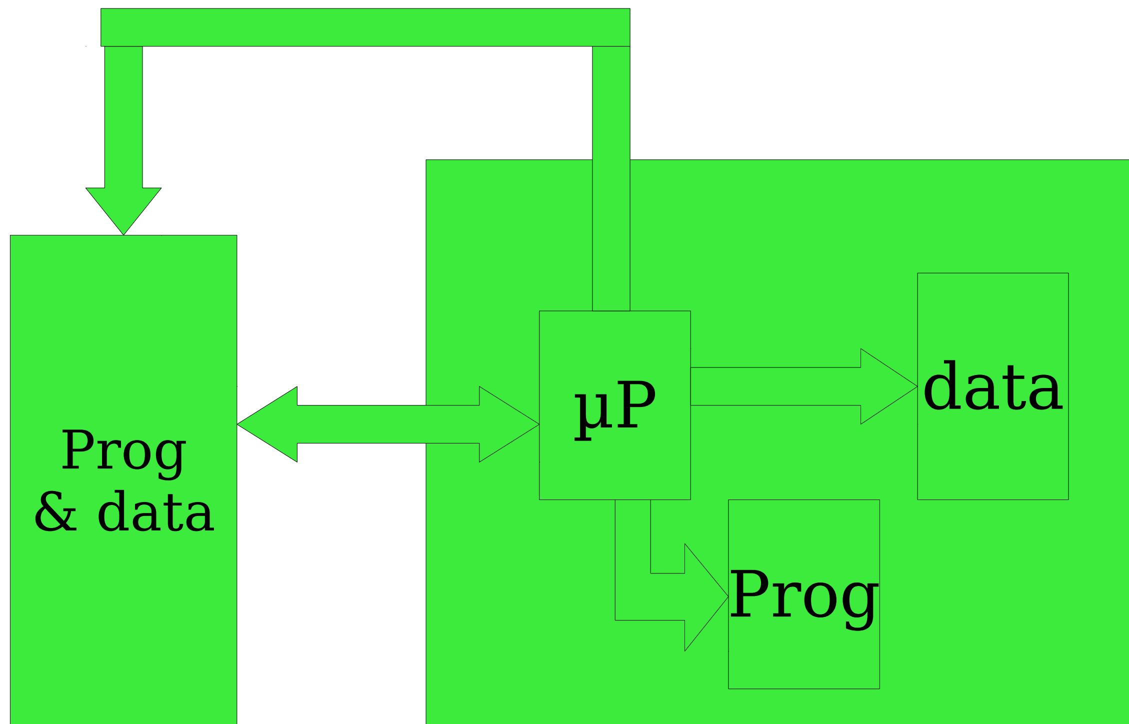


Deux espaces mémoires indépendants pour stocker les données (variables) et les instructions (programme).

Plus de broches mais accès simultané données/instructions

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

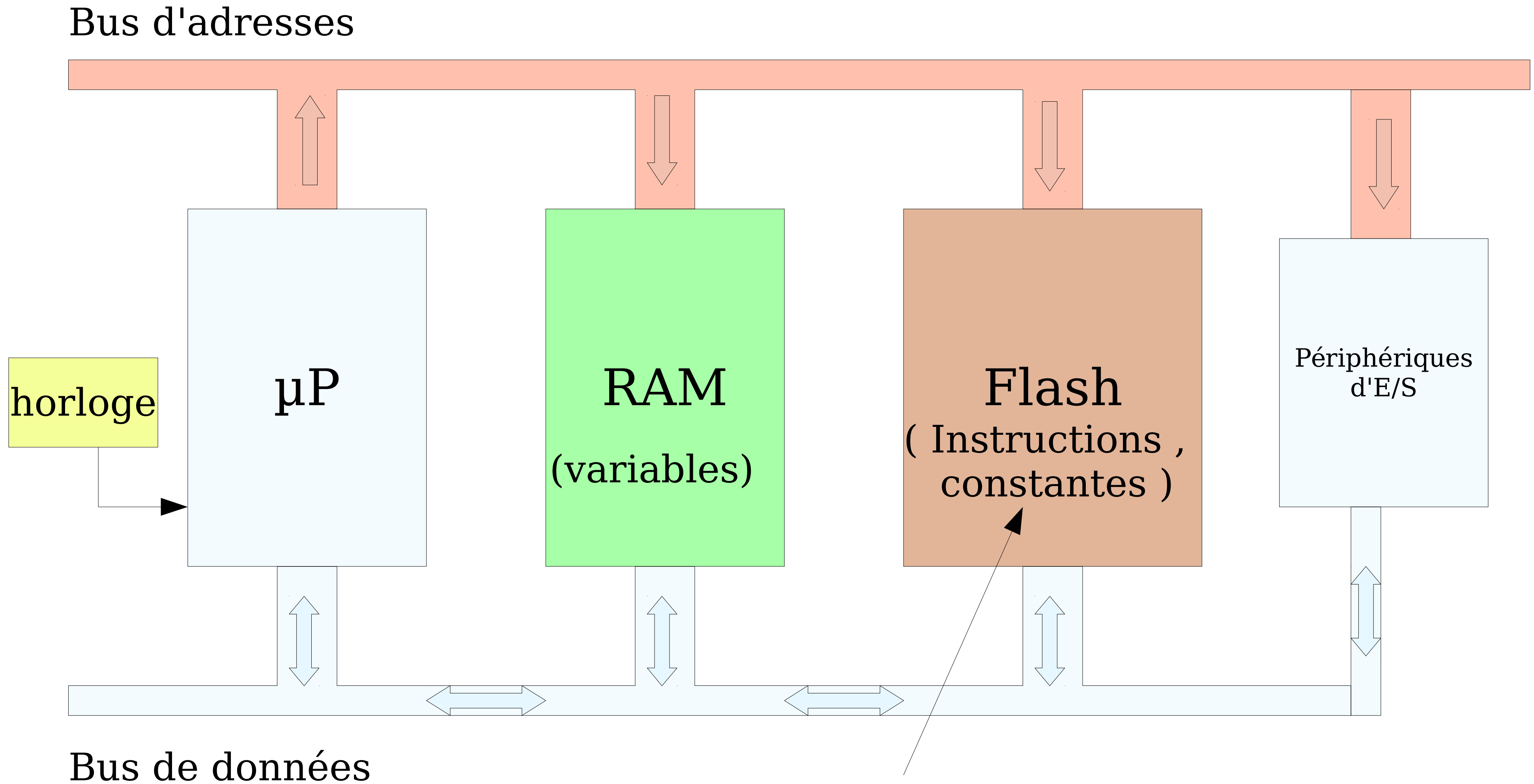
Architecture mixte



Harvard interne / Von Neuman externe

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Système dédié (Von Neuman)



Un programme **unique**
(mise à jour exceptionnelle)

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Système généraliste

L'utilisateur d'un ordinateur généraliste peut charger et exécuter les programmes qu'il souhaite.
Ces programmes peuvent même être inconnus lors de la conception de la machine.

La sécurité et la sûreté sont donc plus difficiles à garantir.

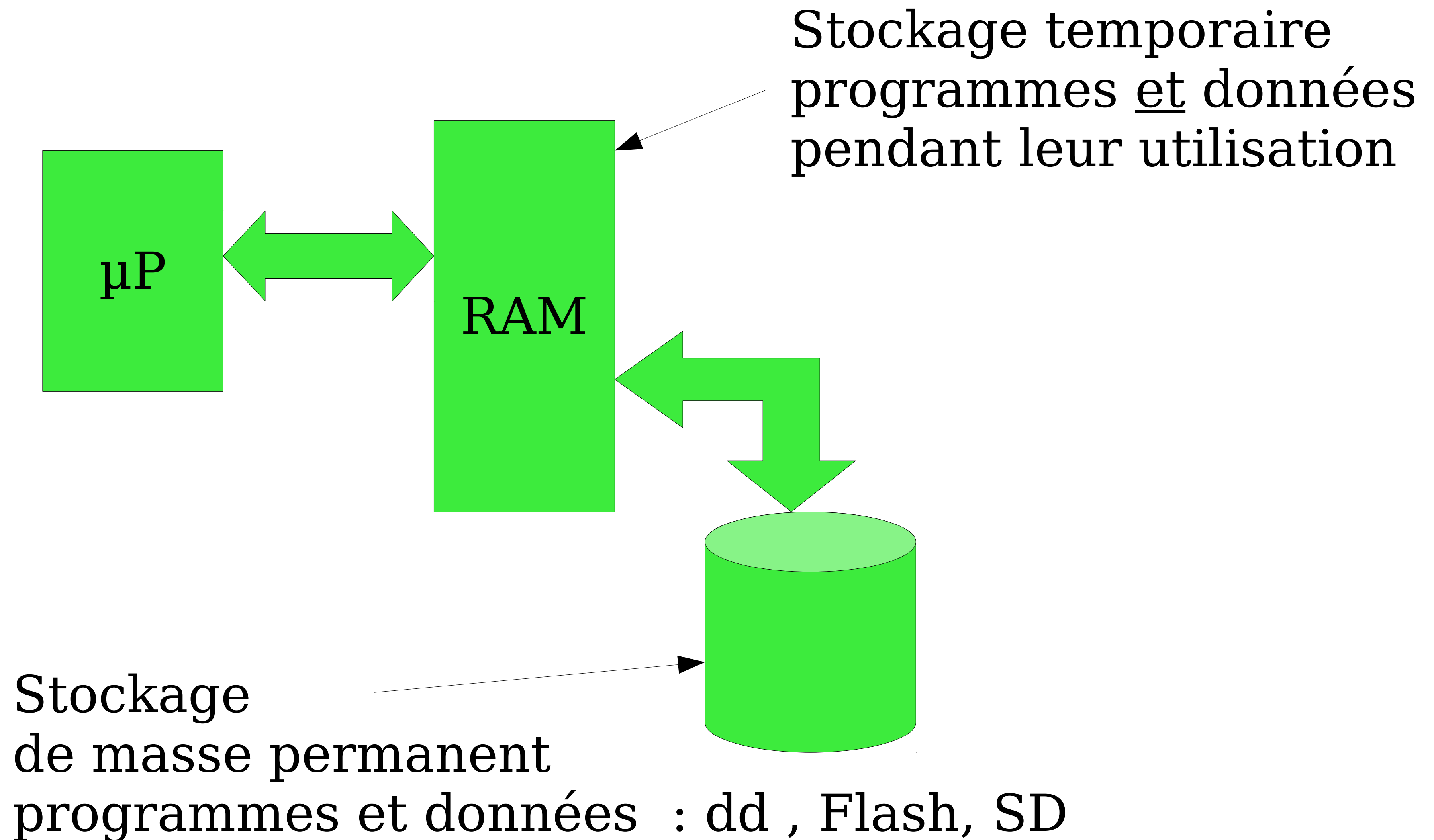
On utilise un périphérique de stockage de masse pour stocker les différents programmes (disque dur, compact flash).
Comme ces périphériques sont généralement lent,
on charge le programme en RAM avant de l'exécuter.

Ordinateur généraliste : Programmes <u>en exécution</u> ET variables en RAM

Stockage programme/data : Flash,SD, disque dur...

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Ordinateur généraliste



© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Microcontrôleur

Un microcontrôleur est un circuit *unique* qui constitue un système minimum. Il permet de réaliser une application complexe avec très peu de composants annexes.
Il comprend :

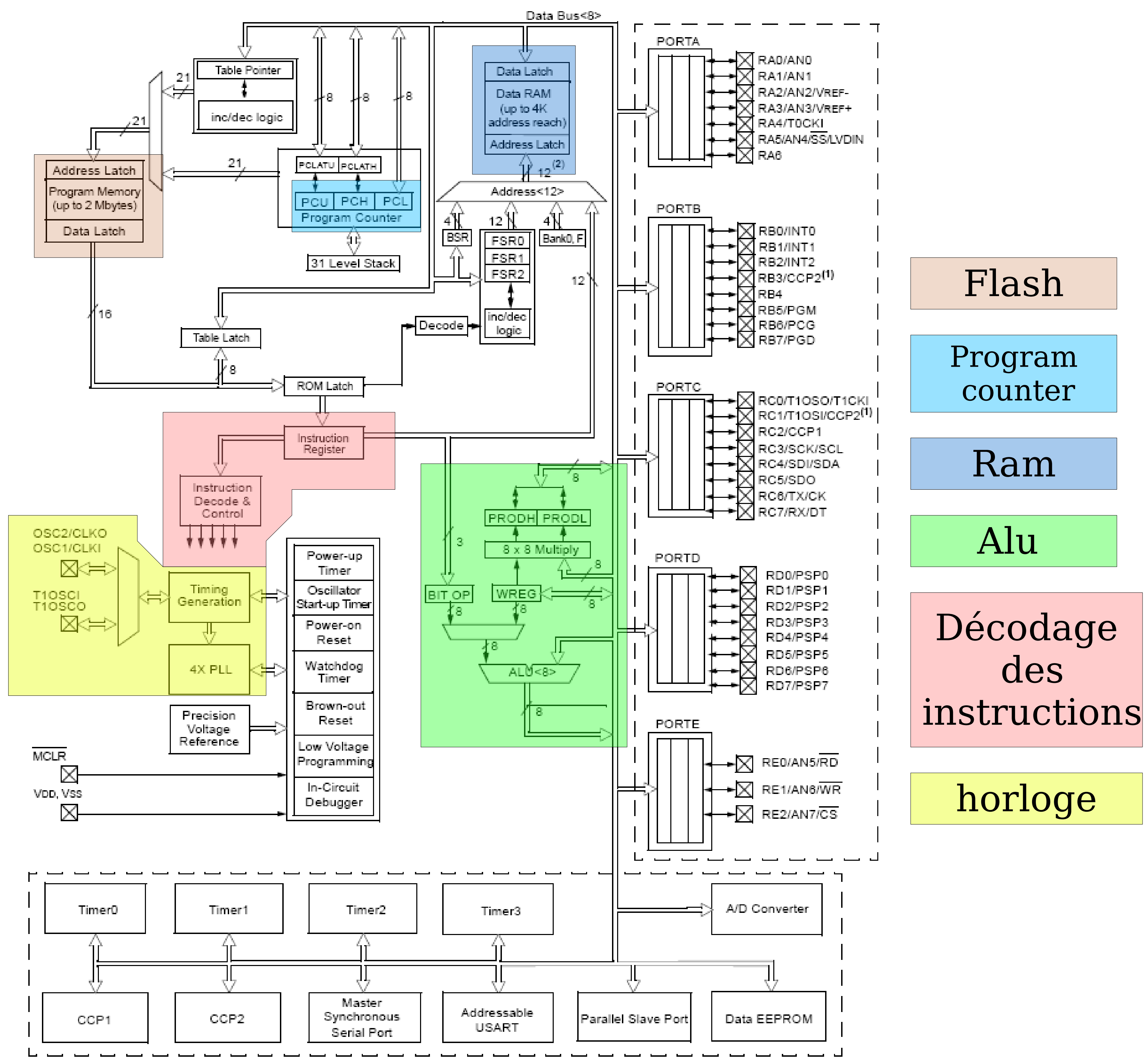
un microprocesseur,
de la mémoire RAM (pour stocker les variables),
de la mémoire Flash ou OTP (pour stocker le programme),

et de nombreux périphériques :
- Ports d'entrées/sorties logiques
- Convertisseurs analogique/numérique
- Périphériques de communications (série,I2C,CAN,USB...)
- Timers (compteurs utilisés pour la gestion du temps)
- Générateur de signaux PWM

etc....

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

PIC18F4520



Microcontrôleur exemples

ATMega 328P : 8bits Horloge 20MHz
32ko Flash , 2ko RAM , 1ko EEPROM
3 Timers/PWM, 1UART, 1 I2C, 1 SPI
1 ADC 10bits, ...

AMD21 ARM Cortex M0+ : 32 bits, 48MHz, 12 DMA
256ko Flash, 32 ko RAM, Interface USB
6 "SERCOM" : UART,I2C,SPI, 8 Timers,
1 ADC 12bits, 1 DAC 10bits, RTCC,....

ESP8266 SoC : 32 bits, 80 MHz, Wifi intégré
96Ko RAM (user 36ko) , 4 Mo Flash
2 UART, 1 I2C, 1 SPI, ADC 10 bits
I2S, DMA

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Système temps réel

Un **système temps réel** est un système qui doit répondre en un temps borné à une sollicitation.

temps réel dur (hard real time) :
la non réponse en un temps minimum conduit à un grave dysfonctionnement du système.

temps réel mou (soft real time) :
les tâches doivent être effectuée le plus rapidement possible mais quelques dépassements sont tolérés.

Les systèmes temps réels doivent également être très fiables

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license
Created with OpenOffice.org 2.0

Système multitâche

Un programme tournant sur un processeur doit assurer plusieurs fonctionnalités "en même temps" :

On découpe le problème en tâches qui sont des parties de code qui tournent indépendamment les unes des autres en partageant le temps CPU. On a l'impression que plusieurs programmes tournent simultanément.

Le découpage en tâches facilite l'écriture, la mise au point et la maintenance de gros programmes

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Système multitâche

Les tâches peuvent être complètement indépendantes.
  Dans ce cas leur seul point commun est de partager le même processeur.

Les tâches peuvent communiquer entre elles.

Les tâches peuvent partager des ressources communes.

Les tâches peuvent se synchroniser.

Une tâche peut lancer ou arrêter d'autres tâches.

Les tâches n'ont pas toutes la même importance :
  notion de priorité (fixe ou dynamique)

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Système multitâche

Ici une tâche (task) peut être appelée indifféremment
process,
thread,

Suivant le contexte ou le système d'exploitation
il peut y avoir des distinctions entre ces termes.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Système d'exploitation multitâche

Un système d'exploitation multitâche classique
 (linux, windows,...) est fait pour optimiser la
performance moyenne.
Toutes les tâches ont (à peu près) la même importance.
Le but est de partager équitablement le temps CPU entre les différentes tâches.

Exemple :
On peut se satisfaire qu'un document OpenOffice soit ouvert en moins de 10 secondes en moyenne.
On tolère que si la charge est importante, ce temps soit exceptionnellement doublé ou triplé.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Système multitâche temps reél

Toutes les tâches n'ont pas la même importance.

Il y a une notion de priorité entre tâches.

Le système d'exploitation doit garantir qu'une tâche de priorité élevée pourra disposer du processeur *avant* une tâche de faible priorité.

Le système doit garantir un temps de réponse d'une tâche maximal dans le ***pire des cas***.

Comme les systèmes multitâches temps réel sont souvent destinés à commander des processus physiques, une grande *fiabilité* est exigée (asservissement, airbag, avionique,....).

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Programmation sans OS Temps réel

Le programmeur est entièrement responsable de la maîtrise du temps et de l'enchaînement des tâches de l'application.

Le modèle de programmation utilisé est :

boucle infinie+interruptions

Très utilisé pour des applications "simples" tournant sur de petits microcontrôleurs où la taille mémoire et la puissance de calcul est limitée.

Mémoire et temps µP réservés entièrement à l'application

Pas de licence, espace mémoire, temps processeur supplémentaires pris par un OS.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Programmation avec un OS temps réel

Un logiciel, le noyau temps réel, prend en charge l'enchaînement des tâches de l'application.

Les tâches sont vues comme des programmes indépendants qui "pensent" qu'ils disposent du processeur pour eux seuls.

Chaque tâche possède une priorité. Le noyau garanti l'exécution de la tâche la plus prioritaire à un moment donné.

Permet de réaliser des applications complexes mais nécessite plus de ressources mémoires et de capacité de calculs. (empreinte de l'OS qq10ko à qq100ko)

S'il n'est pas libre, le coût de l'OS est prendre en compte.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Programmation sans OS temps réel

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Modèle boucle infinie/ interruptions
(superloop background/foreground)

Un programme embarqué a la particularité de démarrer à la mise sous tension du système et de ne s'arrêter qu'à la coupure de l'alimentation. C'est donc un programme qui après une séquence d'initialisation tourne en *une boucle infinie*.

Cette boucle peut être interrompue par des *interruptions*

Les *tâches* de l'application sont exécutés en séquences les unes après les autres dans un ordre prédéfinit.

Les tâches sont ici des séquences d'instruction qui répondent aux différentes fonctionnalités de l'application. Ce ne sont pas des boucles infinies.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Mécanisme d'interruption

Une interruption est un évènement matériel* qui déclenche *très rapidement* l'exécution d'une partie de programme (appelée programme d'interruption). Après l'exécution du programme d'interruption, le µP reprend l'exécution normale du programme interrompu.

* : sur certains µP, il existe aussi des événements logiciels (division par 0, erreur d'adresse,instruction spéciale,...) qui peuvent déclencher une interruption : on parle alors d'*interruption logicielle* (ou synchrone).

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Modèle boucle infinie/ interruptions

```
void main(void)
{
    séquence d'initialisation ;
    ....;
    autorisation des interruptions ;
    for( ;;)
    {
        Tâche n°1 ;
        Tâche n°2 ;
        ....;
        Tâche n° N ;
    }
}
```

```
interrupt void
isr_timer(void)
{
    ......;
}
```

```
interrupt void
isr_pulse(void)
{
  ........;
}
```

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Modèle arduino

```
int main(void)
{
   init();
   initVariant();
#if defined(USBCON)
   USBDevice.attach();
#endif
   setup();
   for (;;) {
    loop();
    if (serialEventRun)
        serialEventRun();
   }

   return 0;
}
```

L'utilisateur fournit seulement les fonctions setup() et loop(). Le modèle init/boucle infinie est rendu quasi-obligatoire.

Avant setup(), des initialisations sont faites, notamment le timer0 pour la fonction millis()

Il est possible d'écrire son propre main()
 pour éviter init par exemple :
http://arlotto.univ-tln.fr/arduino/article/et-main-alors

```
ISR(TIMER1_OVF_vect)
{
 … … ;
}
```

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Interruptions avec avr-gcc

Il faut autoriser l'interruption dans le setup()
(voir la doc de votre micro) puis écrire une fonction correspondant au vecteur utilisé :

```
#include <avr/interrupt.h>

ISR(ADC_vect) {
// user code here
 }
```

Voir la liste des vecteurs ici :

http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Code de la tâche n°2

```
void Tache2(void)
{
    ..... ;
    // fin de course ?
    if(digitalRead(STOP)==1)
    {
      // arrêt moteur !
      digitalWrite(MOTEUR,0);
    }
    ..... ;
}
```

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Timing boucle infinie sans interruption

Exécution

Tboucle

Init | T1 | T2 | T3 | T1 | T2 | T3 | T1

temps

Événement pour T2

Temps de réponse de T2

Le temps de réponse de la tâche <= Tboucle

(Tboucle n'est pas constant)

Pire cas Temps de réponse=Tboucle Maxi

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Pire cas pour la tâche n°2

Exécution

Tboucle

Init	T1	T2	T3	T1	T2	T3	T1

temps

L'entrée change "juste après" l'exécution du test

Temps de réponse de T2 (maxi)

Pire cas Temps de réponse=Tboucle Maxi

(une interruption peut également augmenter Tboucle)

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Encore Pire ! La tâche peut manquer l'événement

Exécution

Tboucle

Init	T1	T2	T3	T1	T2	T3	T1

temps

L'entrée revient à 0
avant l'exécution de T2

La sortie ne passe jamais à 0 !

On peut manquer un événement dont la durée est inférieure au temps de boucle

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license
http://arlotto.univ-tln.fr

Timing boucle infinie avec interruptions

Background

Init T1 T1 T2 T2 T3 T3 T3 T1 T1 T2 T2

temps

Interruption 1

évènement n°1

évènement n°2

Interruption 2 (priorité 2 > à 1)

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Boucle infinie + interruptions

Les interruptions augmentent les temps de réponse des tâches et rendent les temps de réponse encore moins prévisibles.
En général on s'arrange pour avoir des routines d'interruptions les plus courtes possibles.

Par contre les interruptions peuvent être utilisées pour diminuer le temps de réponse à certains événements (qu'on ne risque pas de manquer). Le temps de latence est de l'ordre de quelques centaines de nanosecondes à quelques dizaines de microsecondes. (dépend du type de micro et de la fréquence horloge)

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Programmation entièrement en interruption

Background

Init | idle | idle | idle | idle

temps

Interruption

T1 T2 T3 | T1 T2 T3 | T1 T2 T3

A l'état idle le processeur peut être mis
en *mode basse consommation*.
L'interruption peut être une interruption périodique.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Une interruption déclenche la boucle

Background

Init	idle		T1	T2	T3	T1	T2	T3	idle		T1	T2	T3	idle

temps

Interruption

Un clavier peut fonctionner ainsi : un appui sur une touche provoque une interruption qui réveille la boucle pendant un certain temps et ensuite le système repasse en veille à l'attente d'un nouvel appui.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Les interruptions

Traitement immédiat

Le traitement associé à l'interruption est effectué directement dans la routine d'interruption.

Utilisé pour des événements urgents dont le traitement est court.

Ex : arrêt d'urgence, mesure de temps, temporisations....

Traitement différé

La routine d'interruption positionne simplement une variable (globale) qui sera utilisée par une tâche pour effectuer le traitement associé. Le temps de réponse est alors le temps de réponse de la tâche mais l'événement ne peut plus être manqué.

Utilisé pour des événements à ne pas manquer mais dont le traitement est plus long.

Traitement mixte (une partie immédiate, une partie différée)

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Interruptions : Traitement différé

```
volatile uint8_t it;
void main(void) {
    ......;
    for( ;;) {
        ...... ;
        if ( it )
        {
          traitement ;
          it=0;
        }
       ...... ;
          }
}
```

```
interrupt void isr(void)
{
    it = 1 ;
}
```

Rq : Si it vaut 1 lorsque, on arrive
dans la routine c'est qu'on a manqué
le traitement d'au moins une interruption.
On peut tester it et signaler une erreur.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license
http://arlotto.univ-tln.fr

Programmation boucle infinie/interruptions

Les actions "urgentes" sont traitées par interruption.

Les routines d'interruptions sont les plus courtes possibles.

Les traitements dans la boucle infinie (tâches) sont les plus courts possibles car une tâche "longue" retarde toutes les autres.

Lorsqu'une tâche se bloque, elle bloque toutes les autres tâches : Il faut un mécanisme logiciel pour attendre des événements sans boucle while (ni for) : *machine d'états*.

Si une tâche boucle accidentellement, le système se fige. Un *watch dog* matériel permet de resetter le système.

L'initialisation doit également être très courte en cas de reset par un watch dog.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Attendre sans boucler

Dans beaucoup de systèmes on doit *attendre* un événement avant de passer à la suite des opérations.

Souvent les spécifications s'énoncent ainsi :

Mettre le four en marche tant que la température n'a pas atteint 180°C

Attendre l'arrêt complet (vitesse nulle) avant d'ouvrir la porte

Mettre le ventilateur en marche
Attendre deux minutes
Arrêter le ventilateur

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Attendre sans boucler

Attendre l'arrêt complet (vitesse nulle) avant d'ouvrir la porte

```
while(vitesse()!=0);
digitalWrite(PORTE,1);    // ouvrir porte
```

Mettre le four en marche tant que la température n'a pas atteint 180°C

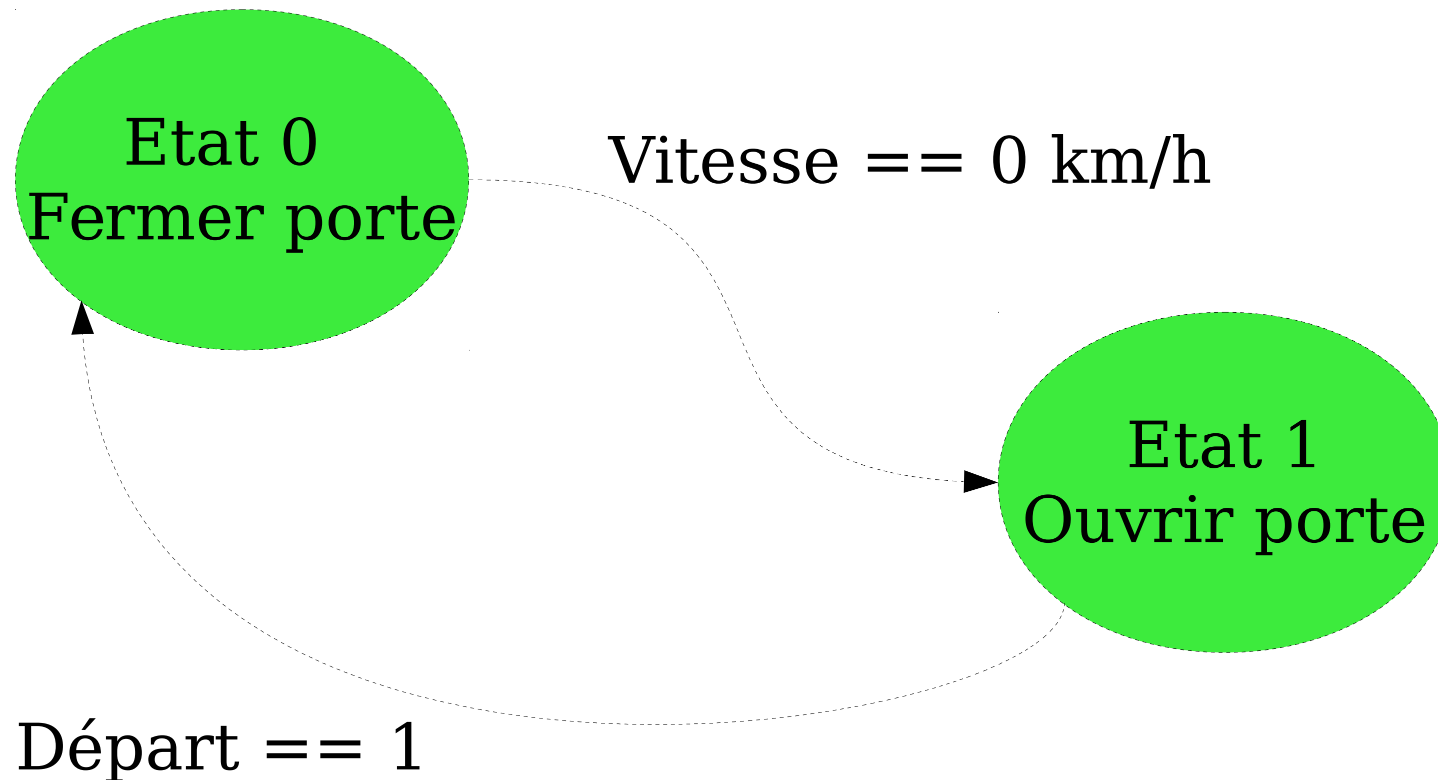
```
digitalWrite(FOUR,1); // four en marche
while(temperature()<180);
digitalWrite(FOUR,0); // arrêt four
```

Avec ces lignes de programme les autres tâches seraient bloquées !

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Attendre sans boucler

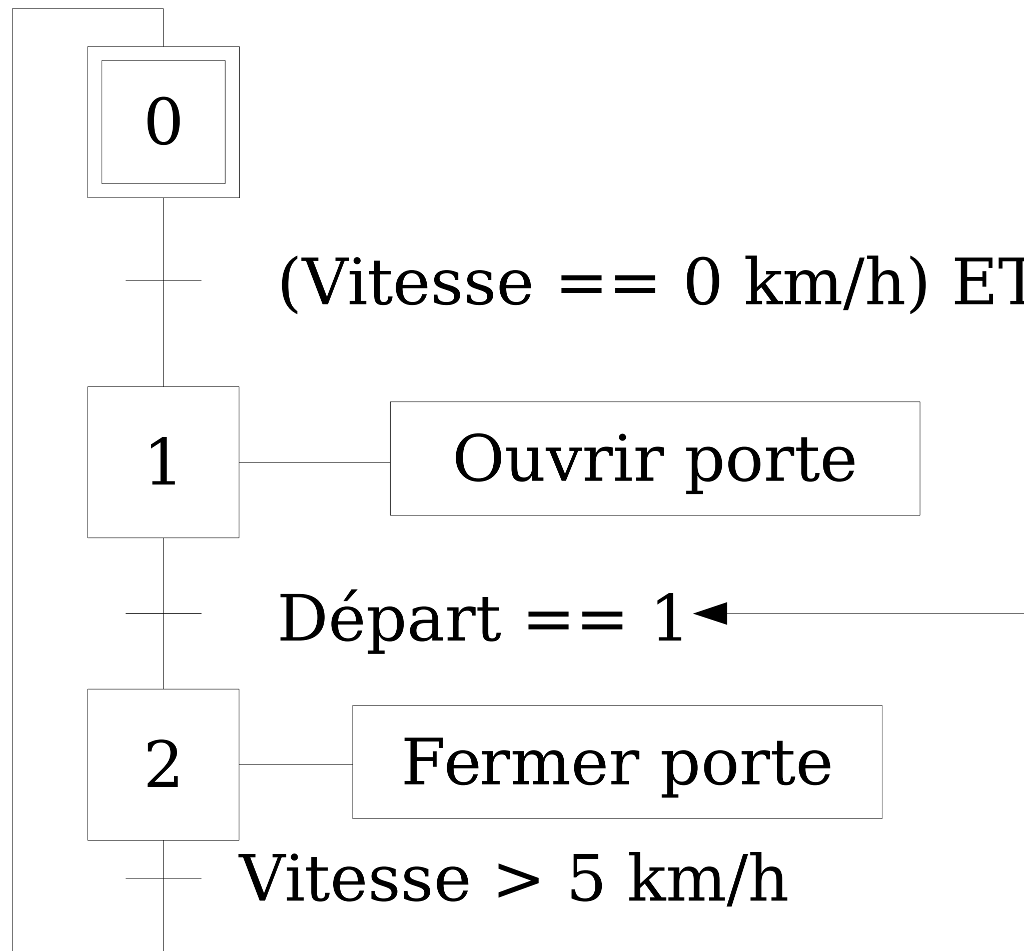
Une solution est de créer une *machine d'état* que ne passe à l'état suivant que lorsque la condition est réalisée



© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Attendre sans boucler

En France, on a coutume d'utiliser la représentation GRAFCET.

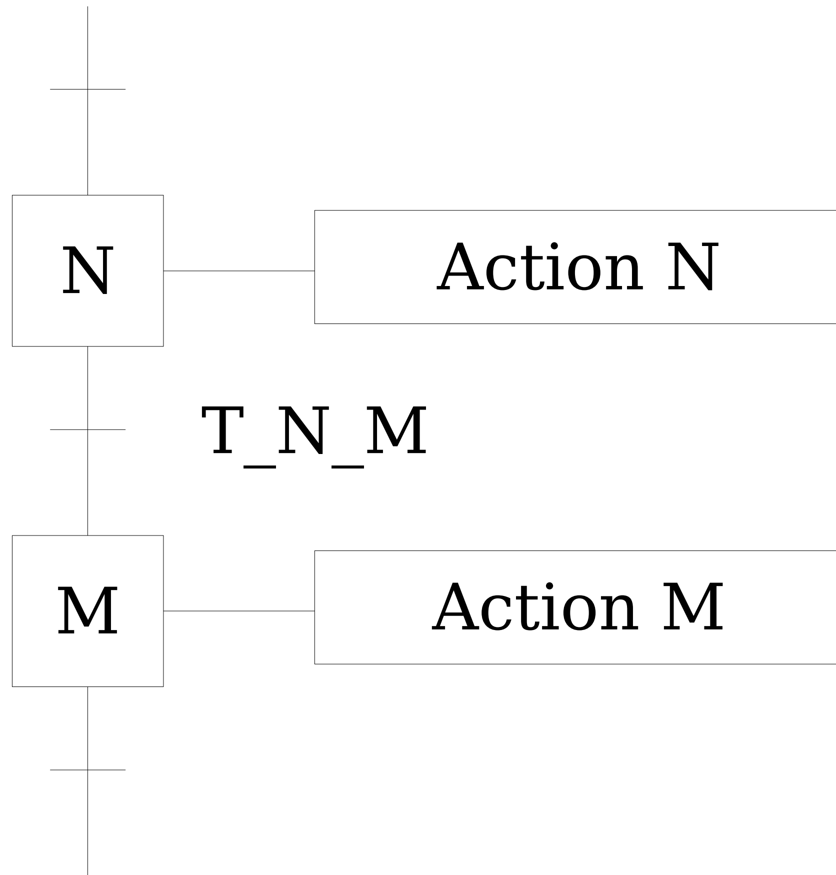


Un état est appelé ici étape

Une transition est une équation logique placée entre deux étapes

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Règles d'évolution d'un grafcet



Lorsque l'étape N est active et que la transition T_N_M est vraie :
L'étape N est désactivée et l'étape M devient active

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Règles d'évolution d'un grafcet

Lorsqu'une étape possède plusieurs étapes suivantes,
on s'imposera la condition qu'une seule transition
ne peut être vraie à la fois :

produit des transitions = 0

Ainsi on est forcé à examiner tous les cas lors de la conception du grafcet.

Lors de la programmation, l'ordre d'évaluation des transitions n'aura quasiment pas d'importance.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Divergence en OU (exclusif)

T_N_M . T_N_P = 0

Après l'étape N une seule des deux étapes M ou P peut être active.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Codage d'un grafcet en C

Une variable (entière) contient le numéro de l'étape courante (en général initialisée à 0)

Le grafcet est codé dans une structure switch...case

Le traitement d'une étape est un cas de la structure switch..case

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Codage d'un grafcet en C

```
int etape = 0 ; //initialisation
....;
for( ; ; ) {
....;
switch (etape) {
  case 0 :
         // traiter étape 0 ;
          break ;
  case 1 :
         // traiter étape 1 ;
          break ;
  case N :
          .......;
          break ;
  default : // erreur !!
          .......;
}
....;
}
```

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

clause default

Normalement on ne peut pas arriver dans la clause default.
Si on y arrive c'est que la variable etape est corrompue !
C'est alors est une défaillance critique du logiciel.

Il faut :
mettre le système en sécurité, prévenir,
redémarrer si nécessaire.
Ce cas doit être prévu dès la conception.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Codage d'un grafcet en C

Traitement d'une étape :

```
case N :
     action_N ;
     if ( T_N_M ) {
       etape = M ; }
     else {
       if ( T_N_P ) {
          etape = P ; }
     }
     break ;
```

On exécute l'action de l'étape courante et ensuite on évalue les transitions vers les étapes suivantes.

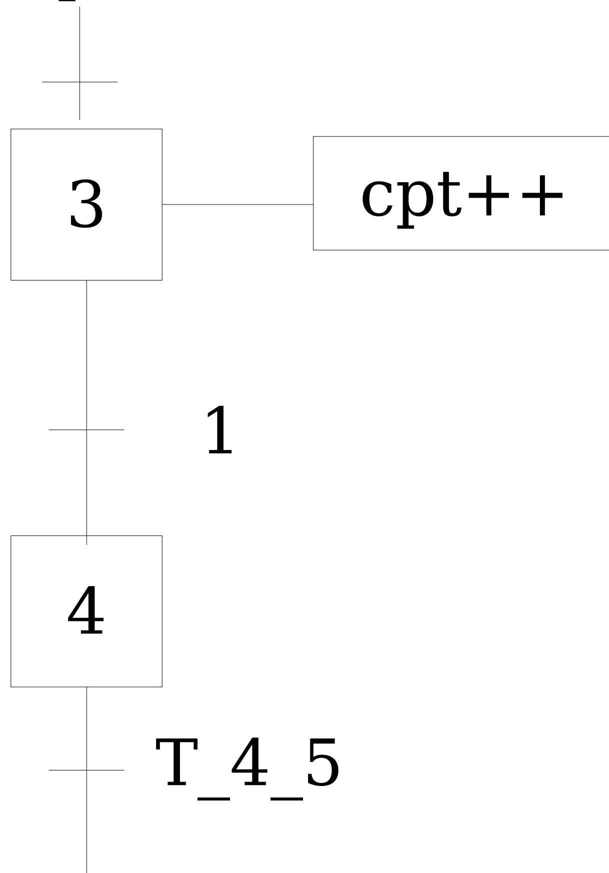
Si une transition est vraie on change uniquement la valeur de la variable qui contient le numéro de l'étape courante.

Lors du prochain passage dans le switch, on va se brancher sur l'étape suivante.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Codage d'un grafcet en C

Lorsqu'une action ne doit pas être répétée, (incrémentation d'un compteur, génération d'une impulsion,...), il est préférable d'ajouter une étape supplémentaire pour dévalider l'étape courante :




```
case 3 :
            cpt++ ;
            etat = 4 ;
            break ;
case 4 :
            if(T_4_5) {
               etat = 5 ;
               }
            break ;
```

Ainsi, l'action de l'étape 3 n'est Effectuée qu'une seule fois.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Temporisations

Temporisations basées sur la durée des instructions

Temporisations basées sur la durée de la boucle générale

Temporisations basées sur une interruption périodique

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Temporisations basées sur la durée des instructions

Ces temporisations consistent à répéter des instructions jusqu'à obtenir la durée voulue. On peut les réaliser à partir d'une boucle for (ou plusieurs imbriqués pour des durées plus longues) :

```
for (i=0; i <3000; i++) ;
```

C'est la cas des fonctions arduino **delay()** (millisecondes) et **delayMicrosenconds()** (µs)

Les durées obtenues sont précises et stables (basées sur l'horloge du µP : quartz).

L'inconvénient majeur est de bloquer la suite du programme pendant la durée de la temporisation.
A réserver à des durées très faibles devant le temps de réponse souhaité.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Exemple d'utilisation de delay() : bit banging

Transformation parallèle/série : (software serial)

```
#define TB 104  // 1/9600 = 104us
void SerialOut( uint8_t data ) {
digitalWrite(LINE,0) ;
delayMicroseconds(TB) ;
for(int i=0;i<8;i++) {
   if ( data & 1)
        digitalWrite(LINE,1) ;
   else
        digitalWrite(LINE,0) ;
   data >>=1 ;
   delayMicroseconds(TB) ;
}
digitalWrite(LINE,1) ;
delayMicroseconds(TB) ; }
```

A l'oscilloscope avec data=0x55, régler TB pour obtenir précisément la durée souhaitée.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Temporisation basée sur la durée de la boucle générale

Il suffit de déclencher un traitement une fois tous les N tours de la boucle générale. La durée obtenue est N.Tboucle.

```
unsigned int n ;
n=0;
.....;
for(; ;) {
....;
n++;
if(n>=N) {
   Action périodique ;
   n = 0 ;
}
....;
}
```

C'est simple à mettre en oeuvre mais c'est peu précis car Tboucle varie.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Temporisations basées sur une interruption périodique

Principe:

Un timer génère un interruption périodique tous les dT.
Dans la routine d'interruption, on incrémente
une variable.
Quand cette variable a variée de N, c'est qu'il s'est écoulé
la durée N.dT.
Ce principe peut être implémenté de différentes manières
selon que l'on souhaite effectuer : des actions périodiques,
des actions à des dates déterminées, des actions
de durées déterminés, ...

C'est précis et non bloquant.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

La fonction millis() de arduino

Cette fonction retourne le nombre de millisecondes début le RESET. Elle est basée sur l'interruption périodique du timer0. On peut l'utiliser pour déclencher des actions périodiques sans bloquer la boucle générale. Voir l'exemple arduino blinkWithoutDelay.

```
unsigned long previousMillis = 0 ;
unsigned long interval = 1000 ;

loop(){
 unsigned long currentMillis = millis();

  if(currentMillis - previousMillis >= interval) {

    previousMillis = currentMillis;

    // on passe ici tous les interval millisecondes
 }
}
```

Le compteur repasse à 0 au bout de 57 jours (2^{32} ms). Si on s'y prend correctement, comme ici en travaillant par soustraction, ce n'est pas un problème, le programme peut fonctionner indéfiniment

Voir : http://arlotto.univ-tln.fr/arduino/article/utilisez-correctement-la-fonction

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Une bibliothèque de temporisation

Un temporisation est vue comme l'association d'un *compteur* et d'un *état* (on peut utiliser une structure en C ou une classe en C++).

l'état peut prendre les valeurs suivantes :
STOP : la temporisation est arrêtée
START : la temporisation démarre
RUNNING : la temporisation est en cours
DONE : la temporisation est échue

Le compteur est une variable entière qui :
est chargée à sa valeur maximale (durée de la temporisation) lorsque la temporisation est dans l'état START.
est décrémentée à intervalle régulier dans l'état RUNNING
fait passer l'état à DONE lorsqu'elle vaut zéro

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Une bibliothèque de temporisation

La temporisation est régie par le grafcet suivant qui est exécuté à intervalle régulier :

0

Start

1 — Cpt = Max

1

2 — Cpt = Cpt - 1

Cpt<=0

3

0 STOP
1 START
2 RUNNING
3 DONE

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Une bibliothèque de temporisation

Le grafcet est écrit dans une fonction timer_tick()
qui est appelée régulièrement dans une fonction d'IT timer,

La condition start est simplement réalisée par l'appel
d'une fonction qui place le grafcet dans l'état START et charge
la valeur maximale dans le compteur.

Le test de l'échéance de la tempo est simplement le test de
La valeur de l'état (ici 3). C'est donc un simple if non bloquant.

En général on fournit au moins une fonction start() et une
fonction isDone() qui teste l'échance.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Une bibliothèque de temporisation

L'utilisation est simple et non bloquante :

```
for(;;) {
        if ( appui_bouton() ) {
          start(600) ;  // démarre la tempo pour 10min
          digitalWrite(CHAUFFAGE,1);
        }
       if ( isDone() ) {
         digitalWrite(CHAUFFAGE,0); // arrêt au bout de 10min
       }
       // le reste de la boucle n'est pas bloqué !
}
```

Si on travaille en c++, on peut créer une classe. Plusieurs temporisateurs peuvent alors facilement partager le même timer. (En c, on peut travailler avec un tableau de compteur et paramétrer les fonctions par un numéro de tempo)

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Programmation avec un OS temps réel

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Les tâches, le scheduler

Un tâche est ici un programme simple qui est conçu comme s'il était seul à s'exécuter sur le processeur.
Un problème complexe est décomposé en une suite de problèmes simples programmés dans des tâches.
En général une tâche est une simple boucle infinie.

Chaque tâche possède, une priorité, son jeu de registres processeur et sa propre zone de pile.
Cet ensemble de données forme le contexte de la tâche.

Le scheduler (ordonnanceur) est la partie du noyau (kernel) qui se charge de donner le processeur à la tâche la plus prioritaire à un moment donné.
Le scheduler est exécuté par l'IT périodique, IT temps reél, et lors de l'appel à certaines fonctions (primitives) de l'OS.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Notion de ressource

Une ressource est une entité utilisée par une tâche.

Ce peut être:

- un périphérique d'E/S : port série, clavier, écran,...
- une variable, un tableau,....
- une zone mémoire
- une fonction non ré-entrante

Une ressource est dite partagée lorsqu'elle est utilisée par plus d'une tâche. Une tâche doit obtenir l'accès à la ressource avant de l'utiliser.

Souvent la ressource ne peut être utilisée que par une seule tâche à la fois : Le mécanisme d'exclusion mutuelle est utilisé pour garantir l'accès exclusif.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Etats d'une tâche

En éxecution (running) :

C'est l'état de la tâche qui en train de s'exécuter. A chaque instant une seule tâche est dans cet état (un seul µP).

Prêt (ready) :

C'est l'état d'une tâche qui possède toutes les ressources nécessaires à son exécution mais qui n'est pas en exécution car elle n'est pas la plus prioritaire.

En attente (waiting) :

Tâche à laquelle il manque une ressource ou qui attend l'occurrence d'un événement pour être dans l'état prêt.

Dormant : Tâche dont le code réside en mémoire mais qui n'est plus disponible pour l'ordonnanceur

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

L'ordonnanceur (scheduler)

Il est exécuté :
périodiquement (à chaque interruption temps réel)
et
lorsqu'une tâche appelle une primitive susceptible de provoquer un changement de contexte
(demande d'une ressource, attente d'un évènement, temporisation, ...).

Il détermine, parmi l'ensemble des tâches prêtes, celle qui est la plus prioritaire et charge son contexte dans les registres du µP.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Etats d'une tâche

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Noyau Préemptif

Un noyau multitâche est dit préemptif lorsqu'il peut mettre une tâche en exécution dès qu'elle devient la plus prioritaire des tâches prêtes sans attendre que la tâche en cours d'exécution se mette d'elle même en attente.

La tâche qui perd le CPU est dite préemptée.

La plupart des noyaux sont préemptifs.

Ce mécanisme permet les meilleurs temps de réponse.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Timing OS temps réel*

Exécution

Init | T1 | T1 | T2 | T3

Interruption périodique (IT temps réel)

temps

Ordonnanceur
(durée exagérée)

Interruption (Task aware)

Interruption (Non Task aware)

*La charge CPU est ici 100% ce qui n'est pas réaliste

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Changement de contexte d'une tâche

Tache n°1

Tache n°2

Pile	Code	Pile	Code

Tâche n°1 — contexte :

Status
...
pc
sp
...
priorité
...
état
...

Tâche n°2 — contexte :

Status
...
pc
sp
...
priorité
...
état
...

Registres du µP

Status
...
pc
sp

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Notion de charge

La tâche de fond (idle task) est une tâche qui a la plus faible priorité
et qui est toujours prête.
Elle est donc exécutée quand aucune autre tâche n'est prête.

Exécution

Init	T1	T1	T2	T3					T2	T2		T1

Interruption périodique (IT temps réel)

temps

Ordonnanceur

Tâche de fond (idle task)

Charge système = Temps passé dans les tâches autre que idle / temps total

Un système devrait être dimensionné pour ne jamais dépasser 70% de charge.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Attribuer des priorités aux tâches

C'est un problème très complexe. Il n'existe pas de méthode générale. La méthode RMS est un bon point de départ mais des ajustements sont nécessaires car ces hypothèses ne sont en général pas entièrement satisfaites.

Il apparaît évident que les tâches les plus critiques devront avoir une priorité élevée. Les tâches les moins importantes auront une priorité plus basse. La priorité des interruptions est également à prendre en compte.

Il existe des outils d'analyses qui permettent des réglages a posteriori mais rien ne remplace une bonne étude préalable

Le problème se simplifie lorsqu'il y a peu de tâches critiques (une ou deux) et que les tâches les plus nombreuses sont des tâches non critiques (affichage, archivage, etc...)

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Méthode RMS : Rate Monotonic Scheduling

Hypothèses (assez réductrices):

Les tâches : sont périodiques
ne sont pas synchronisées
n'échangent pas de données
ne partagent pas de ressources

Un noyau préemptif est utilisé

Attribution des priorités :

Les plus fortes priorités aux tâches qui s'exécutent le plus souvent. (high rate = high priority)

Le théorème RMS garanti alors que toutes les échéances seront respectées si la charge reste inférieure à 70 %

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license
http://arlotto.univ-tln.fr

Méthode RMS : Rate Monotonic Scheduling

Soit E_i le temps d'exécution maximum de la tâche i

Soit T_i la période d'exécution de la tâche i

E_i / T_i correspond à la fraction de temps CPU pris par la tâche i

Toutes les échéances seront respectées en RMS si:

$$\sum \left(\frac{Ei}{Ti}\right) \leq n.(2^{1/n} - 1)$$

n : nombre de tâches

Lorsque n tend vers l'infini $n.(2^{1/n} - 1) \rightarrow ln2 \approx 0.7$ d'où 70%

Attention dans certains cas la tâche la plus fréquente n'est pas la plus importante. RMS reste un bon point de départ.

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Notion de Réentrance

Problème : soit la fonction :

```
int temp ;
void swap(int x, int y) {
 tmp = x ;  (a)
x = y ;     (b)
y = tmp ; } (c)
```

Supposons que cette fonction soit utilisée par deux tâches T1 et T2. (prio2 > prio1)

Si T1 appelle swap et est préemptée par T2 juste après avoir exécutée (a), T2 va exécuter normalement swap, mais quand T1 va s'exécuter elle va reprendre en (b) et (c) avec une valeur de `temp` qui aura été modifiée par T2 !

La fonction swap n'est pas réentrante
(ici il suffirait que `temp` soit locale pour qu'elle le soit)

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Notion de Réentrance

Une fonction est réentrante si elle peut être utilisée sans risques de corruption de donnée dans plusieurs tâches.

Pour qu'une fonction soit réentrante il faut qu'elle n'utilise que des variables locales (ou qu'elle utilise un moyen de protection pour accéder à des données partagées)

Pour utiliser des fonctions non réentrantes dans plusieurs tâches il faut un mécanisme de protection :
sémaphore, section critique, arrêt de scheduler, etc...

Attention les bugs de réentrances sont difficiles à déceler car souvent il n'apparaissent pas durant les premiers tests.
(il faut un changement de contexte au mauvais moment)

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Quelques OS temps réel

Libres : freeRTOS, Rtlinux, eCOS, Xenomai

Propriétaire : µC-OS2, QNX, VxWorks, VRTX,…

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Caractéristiques de µCOS-II

Portable : tourne sur plus de 100 microprocesseurs de 8 à 64 bits

ROMable : Conçu pour l'embarqué (code en ROM)

Configurable : Pour minimiser l'empreinte de l'OS, vous n'activez que les services dont vous avez réellement besoin par des directives `#define` au moment de la compilation.
Les tailles des piles des tâches sont également modifiables

Mutiltâche préemptif : 56 tâches possibles chacune ayant un niveau de priorité différent.

Gère les interruptions : une interruption peut rendre une tâche prête.

Robuste et fiable : Qualification avionique DO178B

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Services de µCOS-II

Gestion du temps : délais, temporisations

Statistique d'utilisation CPU

Sémaphores

Mutex

Evènements

Boite aux lettres

Queue de message (FIFO, LIFO)

Gestion dynamique de la mémoire

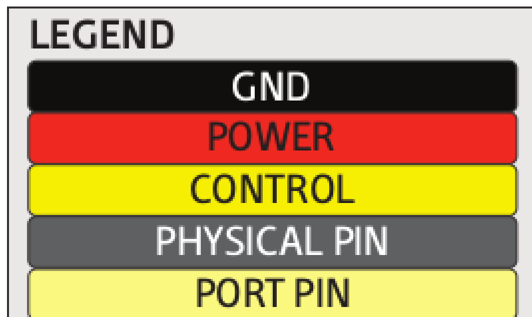
© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

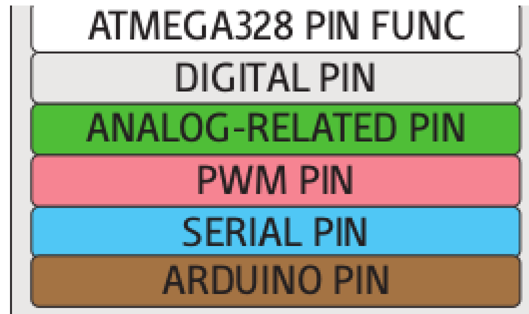
Entrées / sorties logique

ATMEGA328

						Pin
			RESET	PCINT5	PC6	1
		0	RXD	PCINT16	PD0	2
		1	TXD	PCINT17	PD1	3
		2	INT0	PCINT18	PD2	4
OC2B	PWM	3	INT1	PCINT19	PD3	5
	T0	4	XCK	PCINT20	PD4	6
					VCC	7
					GND	8
		OSC1	XTAL1	PCINT6	PB6	9
		OSC2	XTAL2	PCINT7	PB7	10
OC0B	PWM	5	T1	PCINT21	PD5	11
OC0A	PWM	6	AIN0	PCINT22	PD6	12
		7	AIN1	PCINT23	PD7	13
	ICP1	8	CLKO	PCINT0	PB0	14

Pin						
28	PC5	PCINT13	ADC5	A5	SCL	
27	PC4	PCINT12	ADC4	A4	SDA	
26	PC3	PCINT11	ADC3	A3		
25	PC2	PCINT10	ADC2	A2		
24	PC1	PCINT9	ADC1	A1		
23	PC0	PCINT8	ADC0	A0		
22	GND					
21	AREF					
20	VCC					
19	PB5	PCINT5	SCK	13		
18	PB4	PCINT4	MISO	12		
17	PB3	PCINT3	OC2A	11	PWM	MOSI
16	PB2	PCINT2	OC1B	10	PWM	SS
15	PB1	PCINT1	OC1A	9	PWM	





© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Entrées / sorties logique

I/O Pin Equivalent Schematic

Pxn

C_{pin}

R_{pu}

Logic

See Figure "General Digital I/O" for Details

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

E/S logique Arduino

- Configuration en sortie

  pinMode(Numéro,OUTPUT) ;

- Utilisation en sortie

  digitalWrite(Numero,Valeur) ;   Valeur 0 ou 1

- Configuration en entrée

  pinMode(Numéro, INPUT) ; // pas nécessaire au reset

- Utilisation en entrée

  x=digitalRead(Numéro) ;   x = 0 ou 1 selon l'état de l'entrée

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license
http://arlotto.univ-tln.fr

Entrées / sorties logique

Table 14-1. Port Pin Configurations

DDxn	PORTxn	PUD (in MCUCR)	I/O	Pull-up	Comment
0	0	X	Input	No	Tri-state (Hi-Z)
0	1	0	Input	Yes	Pxn will source current if ext. pulled low.
0	1	1	Input	No	Tri-state (Hi-Z)
1	0	X	Output	No	Output Low (Sink)
1	1	X	Output	No	Output High (Source)

Exemple : Pin13 ↔ PORTB bit 5

configuration en sortie :

```
DDRB = 0 ;
DDRB |= _BV(DDB5);
```

Mise à 1 :

```
PORTB |= _BV(PORTB5);
```

Mise à 0 :

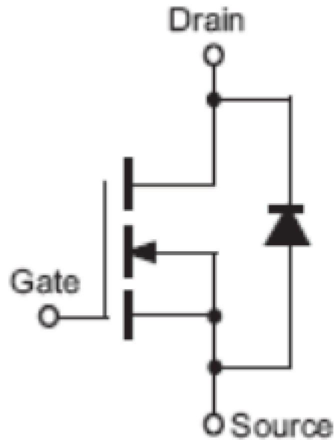
```
PORTB &= ~_BV(PORTB5);
```

Inversion (toggle) : `PINB |= _BV(PINB5);`

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Interfaçage sortie

MOSFET Canal N



BS170

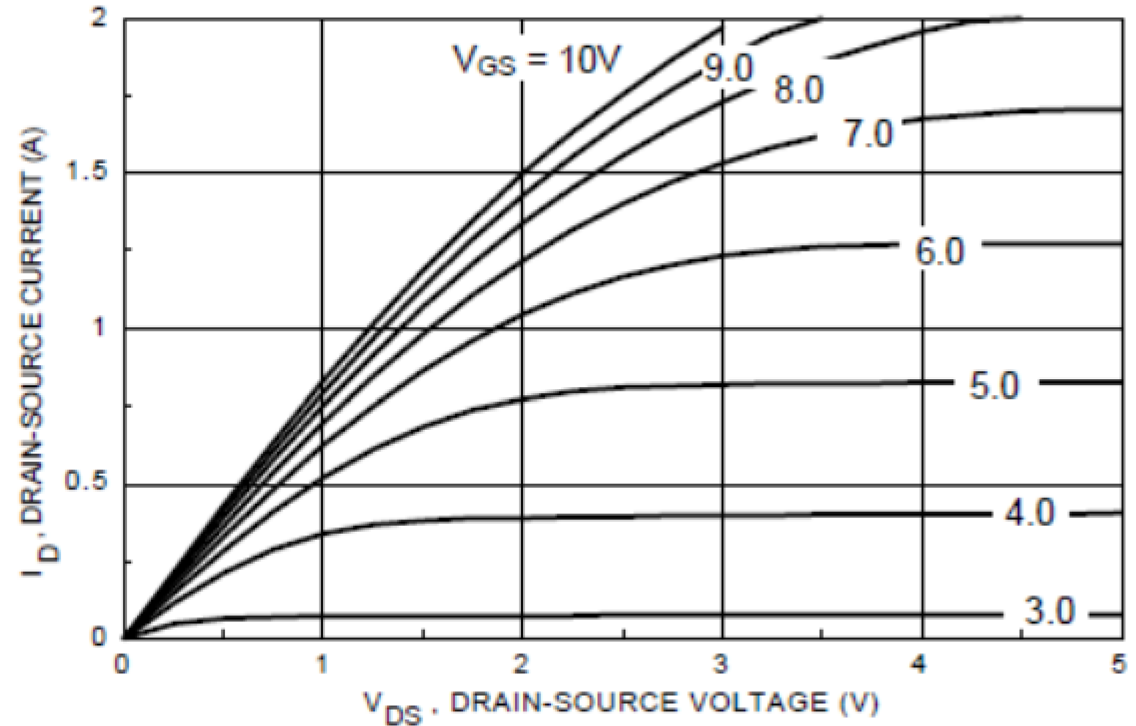


Figure 1. On-Region Characteristics.

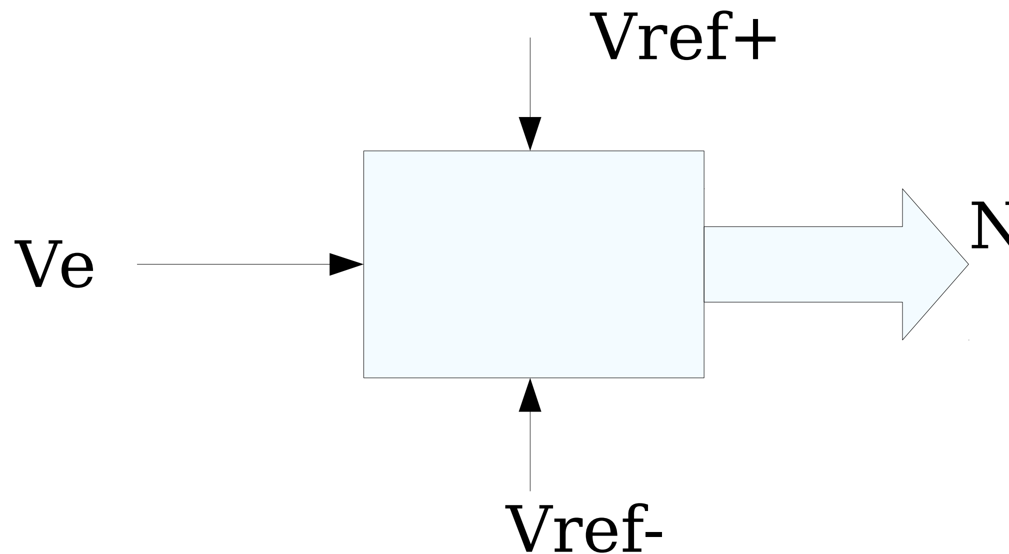
ON CHARACTERISTICS (Notes 1)

$V_{GS(th)}$	Gate Threshold Voltage	$V_{DS} = V_{GS}$, I_D = 1mA	All	0.8	2.1	3	V
$R_{DS(ON)}$	Static Drain-Source On-Resistance	V_{GS} = 10V, I_D = 200mA	All		1.2	5	Ω

© Copyright 2015, Philippe Arlotto Creative Commons Attribution-ShareAlike 2.0 license

Convertisseur analogique/numérique

- Fonction : convertir une tension analogique, comprise entre deux tension de référence Vref- et Vref+, en une valeur numérique N sur n bits.



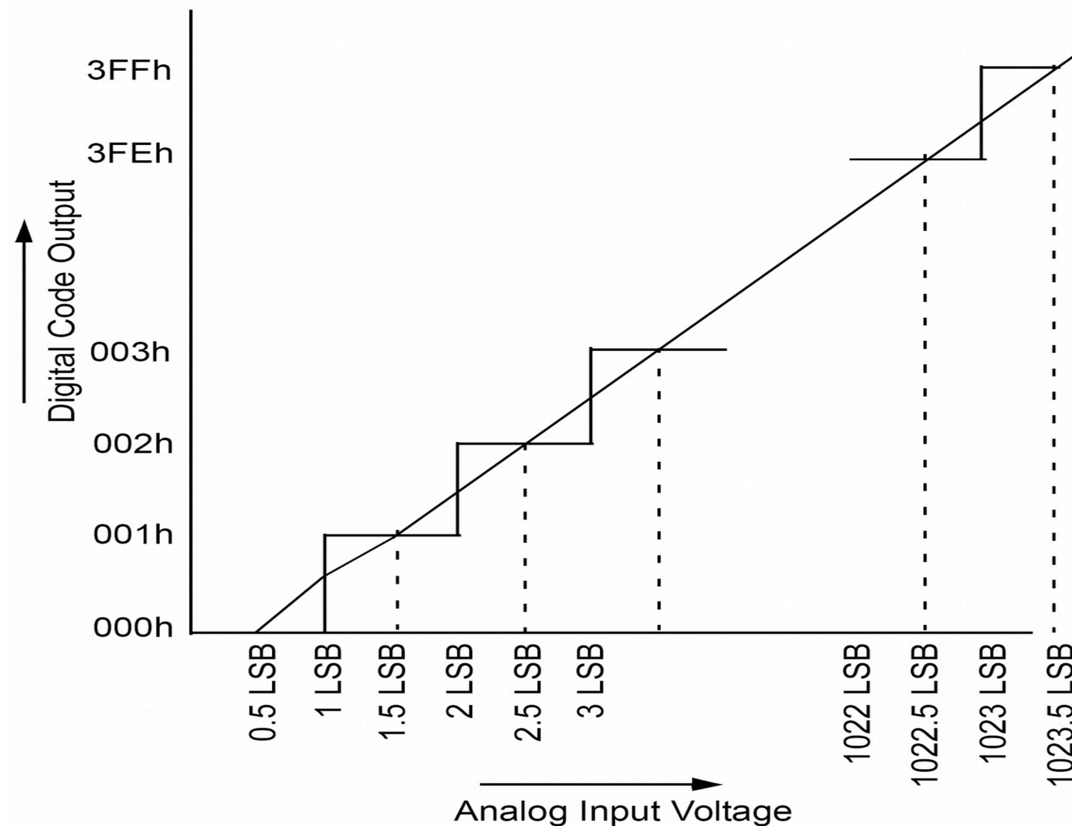
- N = E[2^n x (Ve - Vref-) / (Vref+ - Vref-)] (E[x]:partie entière de x)

(Nmax = 2^n-1 car Ve<Vref+)

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Convertisseur analogique/numérique

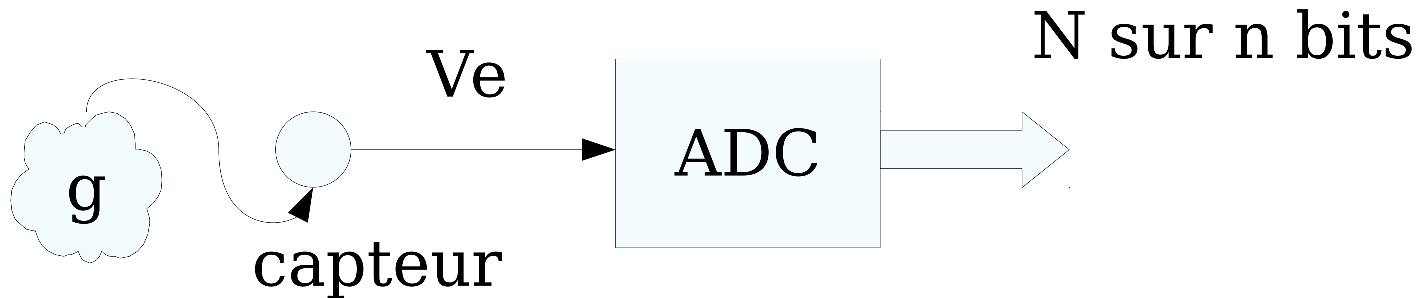
- LSB (ou quantum) : résolution du convertisseur.
  Incrément de tension d'entrée produisant une variation d'une unité du résultat de la conversion $lsb = (Vref+ - Vref-)/2^n$



Fonction de transfert pour n=10bits

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

- <u>Résolution d'une mesure</u> : plus petite variation de la grandeur qui produit une variation du résultat de la conversion.



Ve = f(g) $N = E\,[2^n . (Ve - Vref-) \ / \ (Vref+ - Vref-)]$

g1 => N1 , g2 => N2

(N2-N1)mini = 1 => résolution g2 - g1

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Convertisseur analogique/numérique

- Exemple de calcul de résolution :
  capteur linéaire $Ve = A.g + B$
  $Vref- = 0$ V

  $g1 \Rightarrow Ve1 = A.\ g1 + B \Rightarrow N1 = 2^n.\ (A.g1 + B) / Vref+$
  $g2 \Rightarrow Ve2 = A.\ g2 + B \Rightarrow N2 = 2^n.\ (A.g2 + B) / Vref+$

  $N2 - N1 = 2^n.\ A.\ (g2 - g1) / Vref+$
  $N2 - N1$ mini $= 1$

  => résolution

$$g2 - g1 = Vref+ / 2^n.\ A$$

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

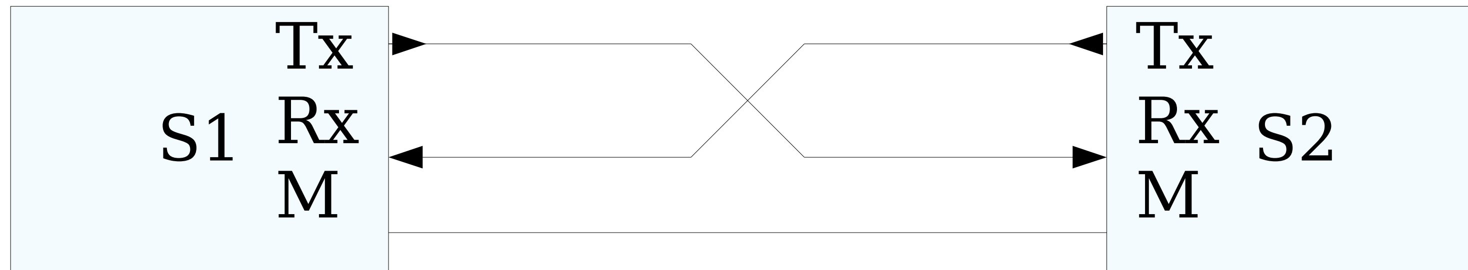
Convertisseur analogique/numérique Arduino

- La fonction **`analogRead()`** retourne le résultat d'une conversion analogique numérique de la tension sur la broche analogique passée en paramètre (A0-A5).
- La fonction **analogReference**() permet de changer la valeur de la référence. Par défaut c'est la tension d'alimentation de la carte.

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Liaison série asynchrone

But : Transmettre des octets entre deux systèmes

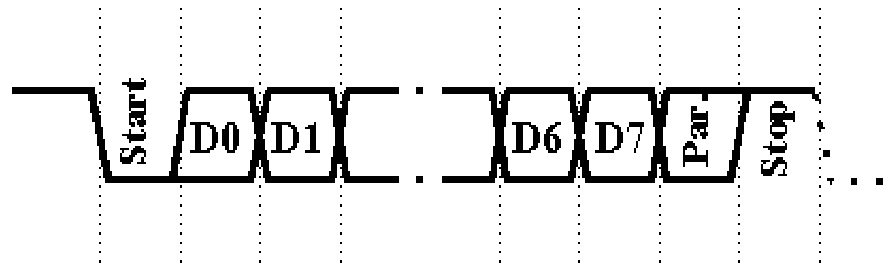


Il y a au moins 3 fils : Rx réception
Tx émission
Masse

Si les systèmes S1 et S2 sont identiques, le câble est forcément *croisé*.

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Format de transmission :



Ligne inactive (idle) : niveau haut
Bit de start : marque le début de la transmission d'un octet.
D0,D1...D7 : l'octet est transmis avec D0 en premier
(lsb fisrt)
Bit de parité : optionnel
Bit de stop : marque la fin de la transmission

(un nouveau caractère peut être transmis dès la fin
du bit de stop. Dans ce cas il n'y a état idle)

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Parité : Mécanisme rudimentaire de détection d'erreur.

A l'émission le bit de parité est positionné pour que le nombre total de bit à 1 soit pair (parité paire)
(ou impair si on travaille en parité impaire).

A la réception, on vérifie que dans l'octet reçu le nombre de bit à 1 est bien pair (ou impair si on travaille en parité impaire).
Si ce n'est pas le cas on considère qu'il y eu erreur de transmission.

Ce mécanisme est mis en défaut si il y a plus d'un bit en erreur.

(Pour détecter (voire corriger) plusieurs erreurs, il faut rajouter plusieurs bits : codes correcteurs d'erreurs.)

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

• Liaison série asynchrone

Norme RS232 :

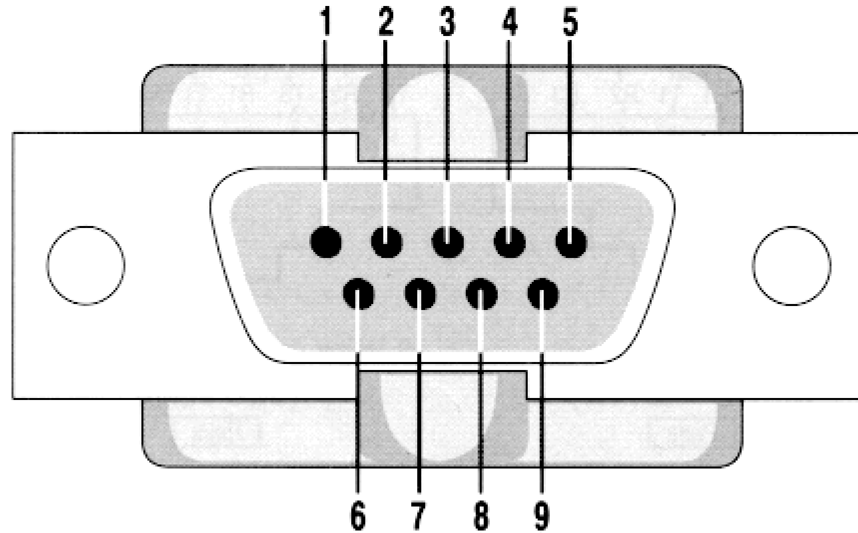
5 V (idle) : - 10 V environ
0V (actif) : + 10 V environ

Norme RS485 :
c'est la différence de tension entre deux fils U+ et U- qui contient la donnée 1:U+>U- 0:U=<U-

Vitesse normalisée (bits/s):
300 , 600 ,1200 , 2400 , 4800, 9600 , 19200 ,38400
 57600,115200
 progression x2 jusqu'à 38400puis 38400 x 1.5 = 57600
 puis 57600 x 2 = 115200

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

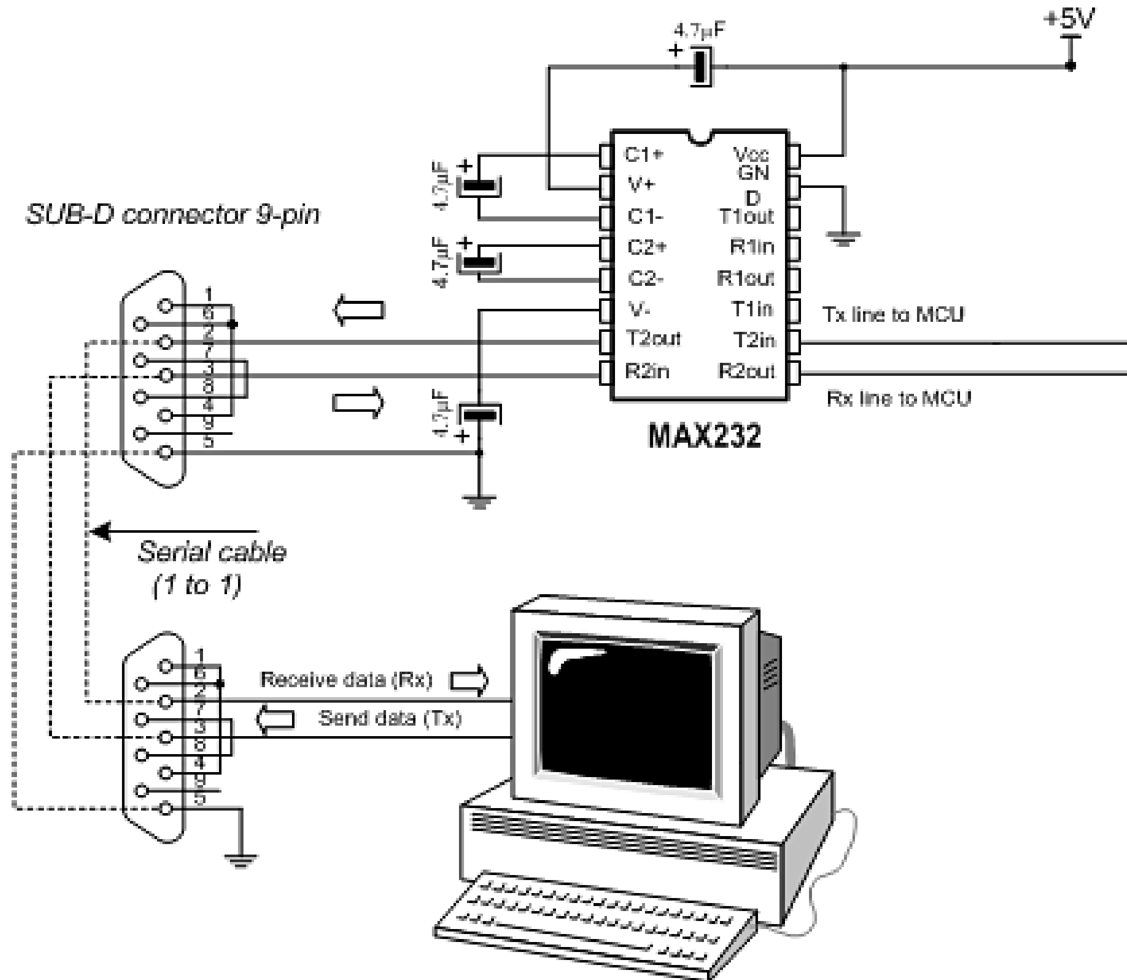
Liaison série asynchrone Brochage DB9



Pin	Signal	Pin	Signal
1	Data Carrier Detect	6	Data Set Ready
2	Received Data	7	Request to Send
3	Transmitted Data	8	Clear to Send
4	Data Terminal Ready	9	Ring Indicator
5	Signal Ground		

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Conversion TTL/RS232



© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Logiciels émulateurs de terminal

Très pratique pour la mise au point un émulateur de terminal est un logiciel qui permet d'envoyer et de recevoir des caractères sur un port série :

Pour Linux : gtkterm , minicom , screen

Pour Windows : Hyperterminal , ...

Nom des ports séries :

Linux : classiques : /dev/ttyS0 /dev/ttyS1, ...

virtuels usb : /dev/ttyUSB0 , /dev/ttyUSB1, ...

Windows :

classiques : COM1 , COM2

virtuels usb : souvent COMx x>3

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

ASCII ou Binaire ?

On peut choisir de transmettre les données en ASCII ou directement en "binaire" :

Pour transmettre la valeur d'une variable :

```
int a = 1000 ;
```

En Ascii :

```
printf("%d",a);
ou Serial.print(a,DEC)  ;
```

Ou

Directement en binaire :

```
Serial.write( a&0xFF) ;    // poids faible
Serial.write( (a&0xFF00)>>8); // poids fort
```

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

ACSII ou binaire ?

En ASCII pour la valeur 1000 , 4 caractères seront
Transmis : '1' '0' '0' '0'

En binaire toujours pour 1000 (0x03E8) seulement
deux octets : 0x03 puis 0xE8 (ou l'inverse)

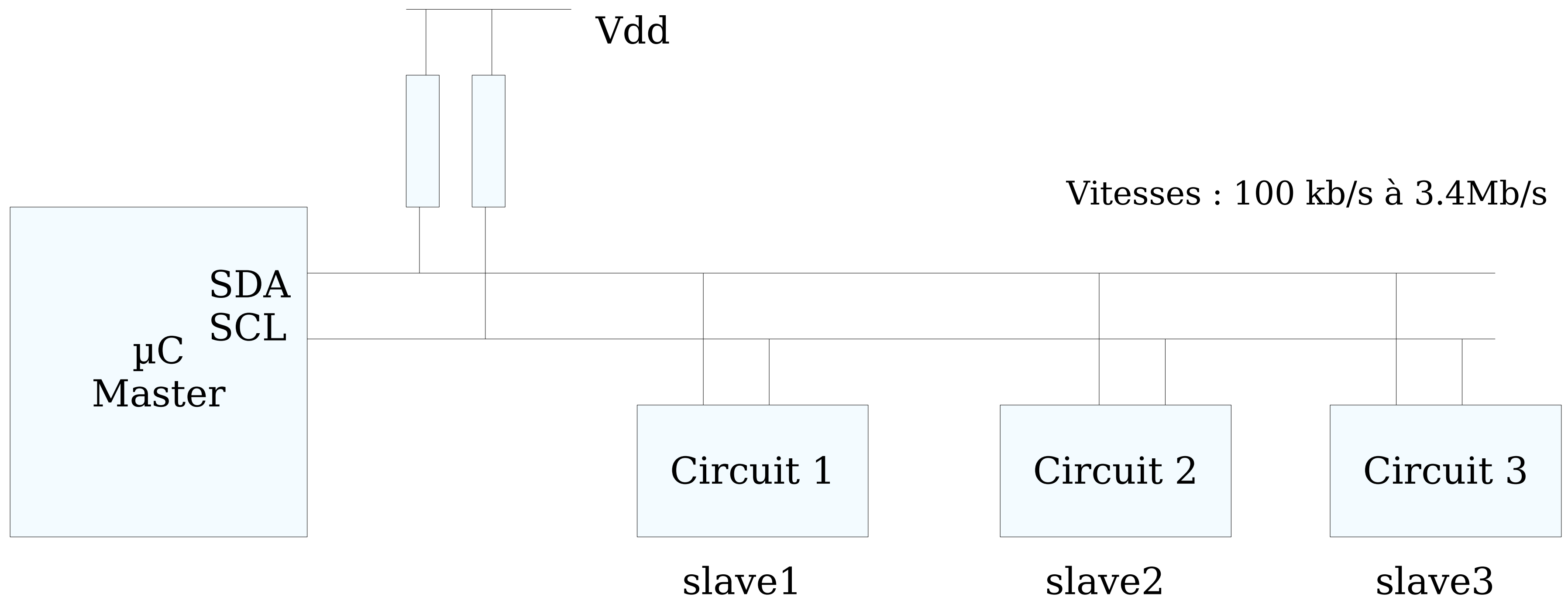
ASCII :
- plus long car plus de valeur à transmettre **:-(**
- lisible directement avec un terminal **:-)**
- compatible avec tous les systèmes   **:-)**

Binaire :
- plus rapide car moins d'octet à transmettre **:-)**
- mise au point plus difficile (pas lisible directement) **:-(**
- on doit tenir compte de l'ordre des octets lors de
la réception : problème Big Endian / Little Endian. **:-(**

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Le Bus I2C

Il permet l'échange bidirectionnel d'informations (octets) entre différents circuits intégrés (I2C = IIC = Inter Integrated Circuit) connectés sur un bus à deux fils SCL (horloge) et SDA (données).



Configuration classique : un µC maître et plusieurs esclaves.
Chaque esclave possède une adresse unique.
(7bits=112 adresses possibles ou 10bits=plus de 1000 adresses possibles)

document de référence : The I2C-bus specification Philips Semiconductors VERSION 2.1 JANUARY 2000 www.nxp.com

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Bus I2C : Terminologie

Transmitter : le circuit qui émet des données sur le bus

Receiver : le circuit qui reçoit des données sur le bus.

Master : le circuit qui est à l'initiative de l'échange sur le bus; Il génère l'horloge et termine le transfert.

Slave : le circuit adressé par un Master.

Lors d'un échange on peut avoir :

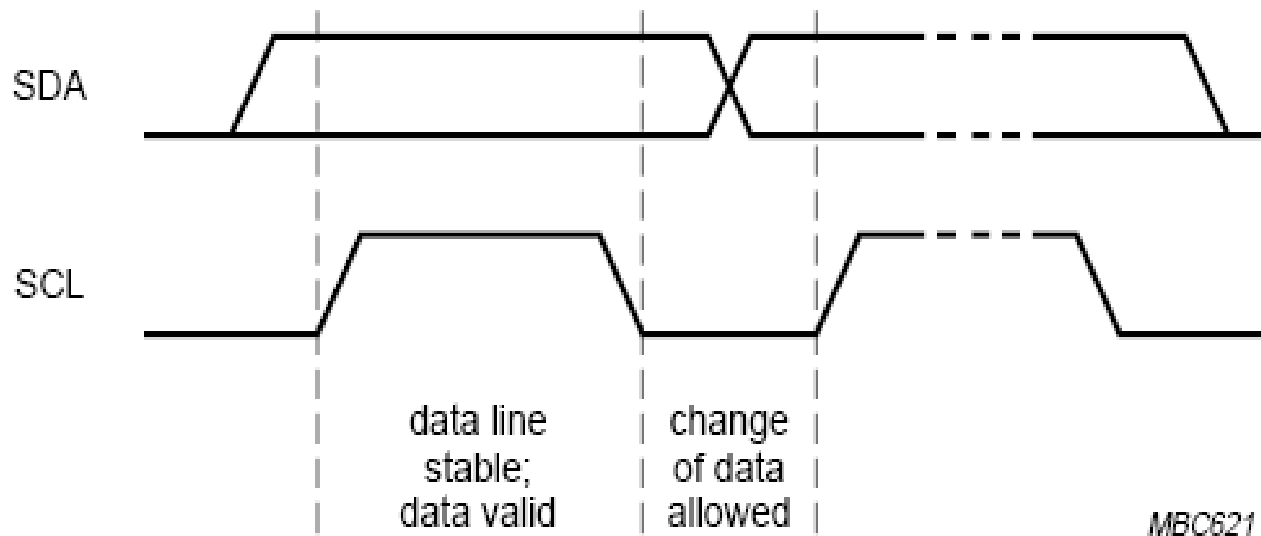
Master/Transmitter Slave/Receiver
puis Master/Receiver Slave/Transmitter

Par exemple lecture d'une mesure sur un capteur I2C.

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Bus I2C : Transfert d'un bit

L'état du bit (sur SDA) est lu lorsque l'horloge (SCL) est à l'état haut. Le changement d'état n'a lieu que lorsque SCL est au niveau bas.

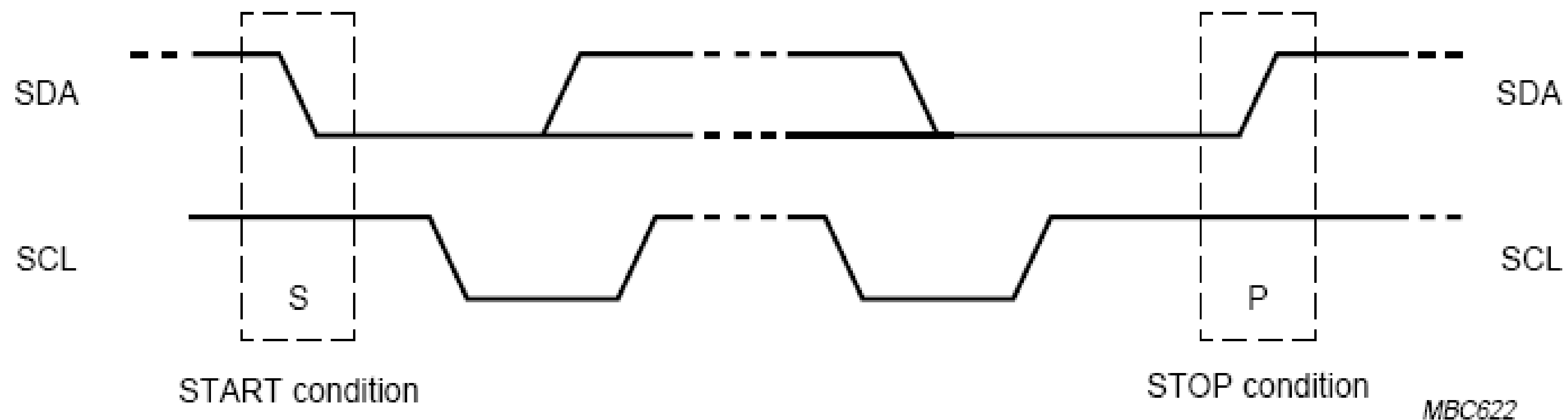


Les échanges se font toujours par octets complets.
(8bits = 8 fronts de SCL)

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Bus I2C : Conditions START et STOP

Lorsque SDA change d'état alors que SCL est au niveau haut, il s'agit de conditions spéciales appelées :
START (S) et STOP (P)

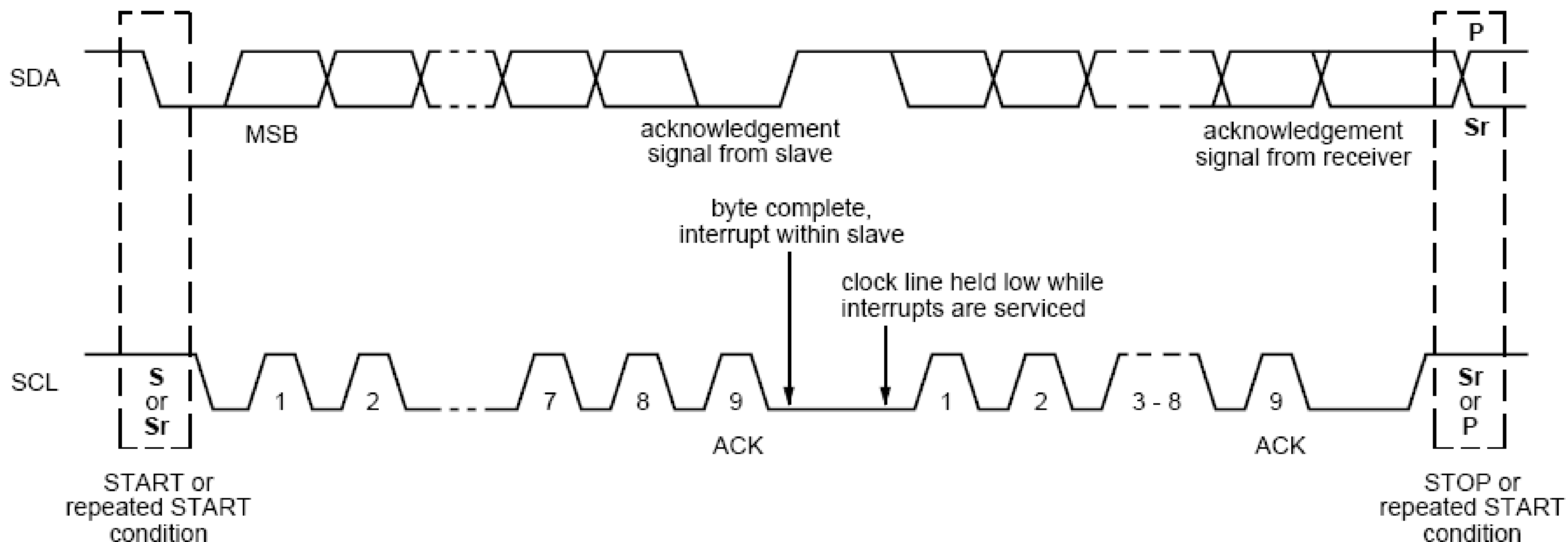


Start et stoP sont toujours générées par un MASTER.
Le bus est occupé après Start et libre après stoP.

Le bus reste occupé si un S remplace un P à la fin d'un échange.
On appelle cette configuration REAPETED START notée Sr.

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

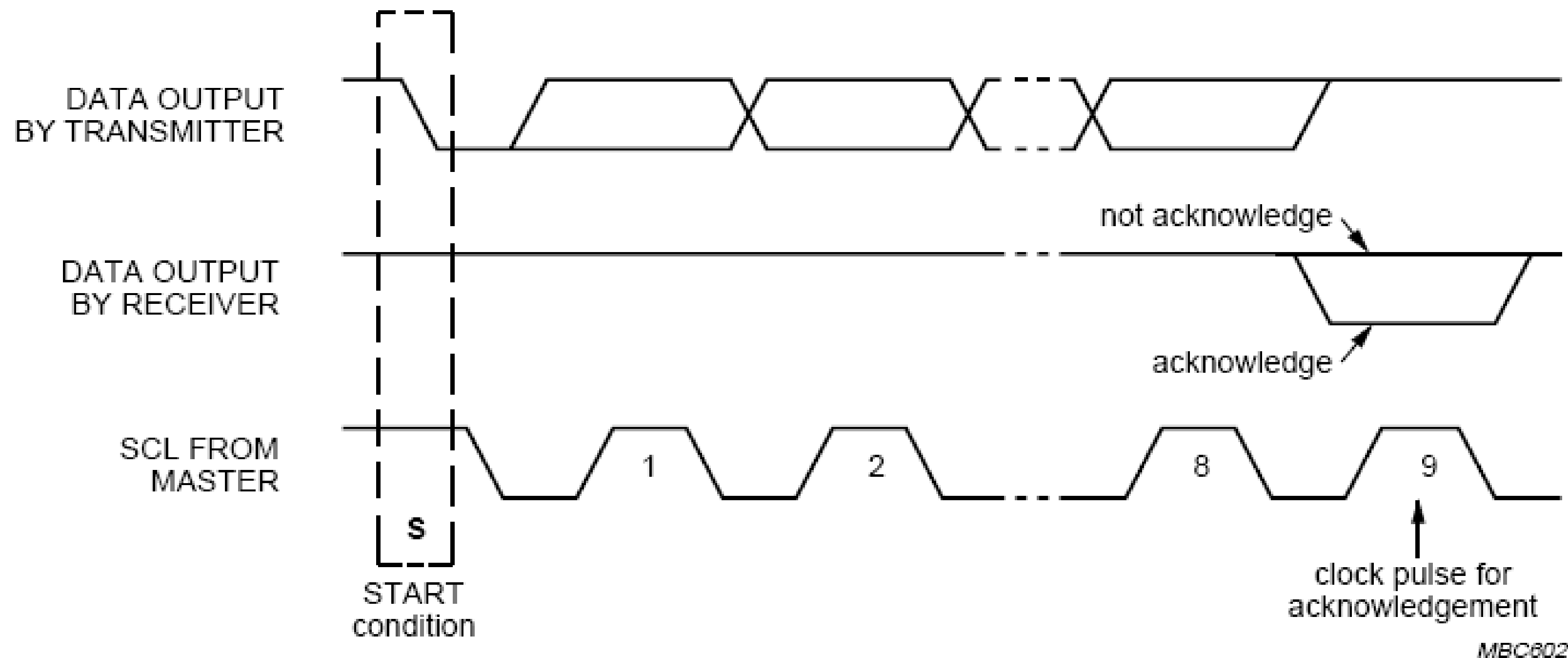
Bus I2C : Acknowledge



Lors du transfert d'un octet une neuvième période d'horloge est générée pour le bit Acknowledge.
Acknowledge = état bas de SDA
Not Acknowledge = état haut de SDA

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Bus I2C : Acknowledge



Acknowledge = état bas de SDA
Not Acknowledge = état haut de SDA

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Bus I2C : Acknowledge

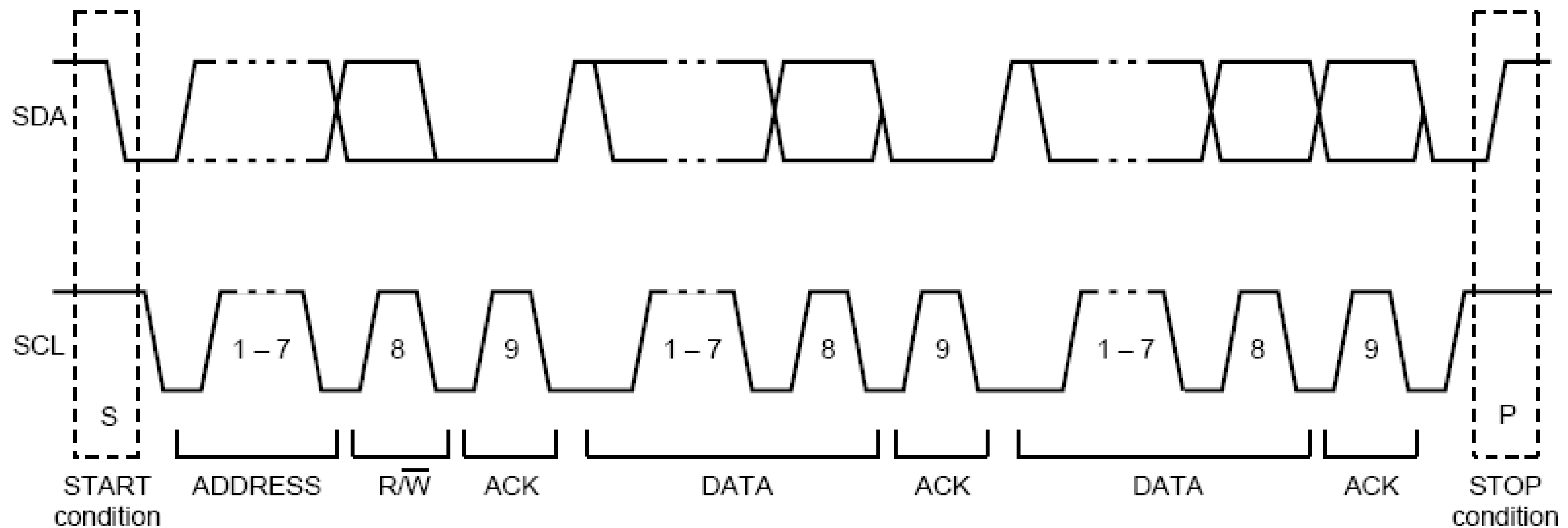
L'acknowledge est toujours généré par un receiver.
Il peut (doit) être testé par le transmitter pour décider de la suite des opérations sur le bus.

Slave receiver :
Un slave receiver génère toujours un Acknowledge à la fin de chaque octet reçu. Si l'acknowledge n'est pas généré, c'est que la donnée n'a pas été reçue correctement (esclave occupé, adresse inconnue, désynchronisation, etc...). Aucune autre donnée ne peut être alors reçue. Le maître doit alors interrompre l'échange par un P (ou un Sr).

Master receiver :
Le master receiver génère un Acknowledge à la fin de chaque octet reçu sauf pour le dernier. Il génère alors un Not Acknowledge pour signaler au slave transmitter la fin d'un échange. Ensuite il génère un P (ou un Sr) afin de clore la transaction sur le bus.

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

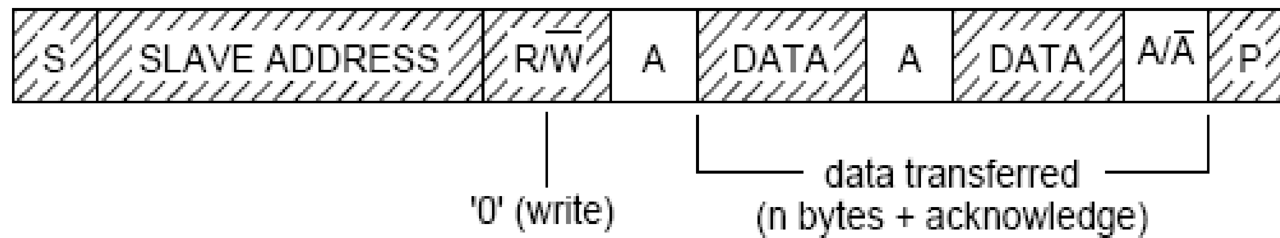
Bus I2C : Adressage 7 bits



Chaque esclave possède une adresse unique sur 7 bits.
L'adresse est transmisse par le maître en ajoutant un 8ème bit R/W: 0 pour écrire (/W) , 1 pour lire (R).
Exemple : adresse 7bits esclave 1001 010.
écriture : 1001 0100 = 0x94 (toujours paire)
lecture : 1001 0101 = 0x95 (toujours impaire)
Par abus de langage on dit que l'esclave possède deux "adresses", l'une en écriture (0x94) l'autre en lecture (0x95)

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Bus I2C : Exemple de transfert (1)



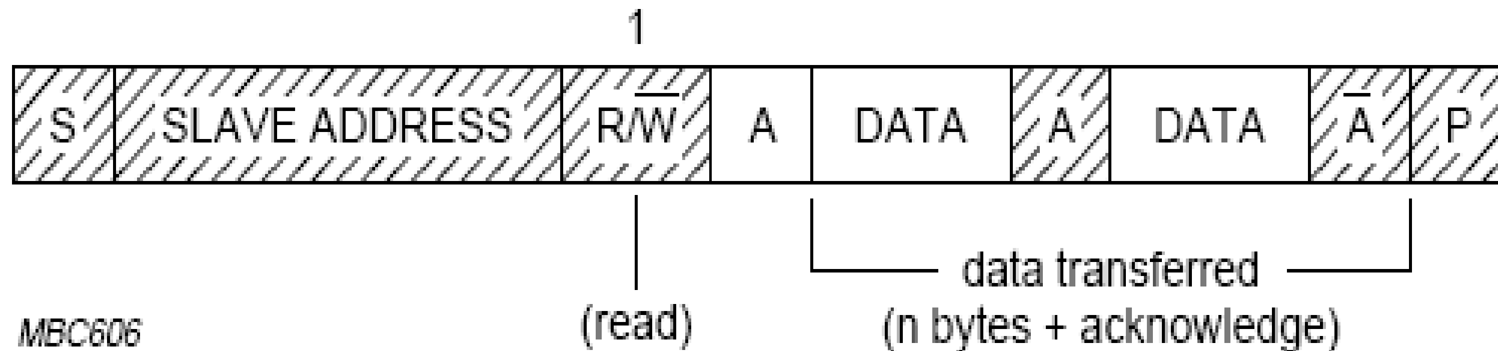
MBC605

A = acknowledge (SDA LOW)
Ā = not acknowledge (SDA HIGH)
S = START condition
P = STOP condition

Un maître envoie des données (master transmitter) à un esclave (slave receiver).

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Bus I2C : Exemple de transfert (2)



from master to slave

from slave to master

Un maître lit des octets d'un esclave.
Master receiver / Slave transmitter.

Remarquez le "not ackonwledge" après la lecture du dernier octet (/A).

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

I2C Arduino

On utilise la librairie Wire :

```
#include <Wire.h>

uint8_t address = 0x12 ;

void setup() {

  Wire.begin();
}
```

Écrire 8 bits

```
uint8_t byteW ;

Wire.beginTransmission(address);
Wire.write(byteW);
Wire.endTransmission();
```

Écrire plusieurs octets

```
uint8_t data[10] ;

Wire.beginTransmission(address);
Wire.write(data,10);
Wire.endTransmission();
```

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

I2C Arduino

Lecture d'un octet :

```
uint8_t data ;

Wire.requestFrom(address, 1); // request 1 byte from slave

  while (Wire.available()) {
    data = Wire.read(); // receive a byte
 }
```

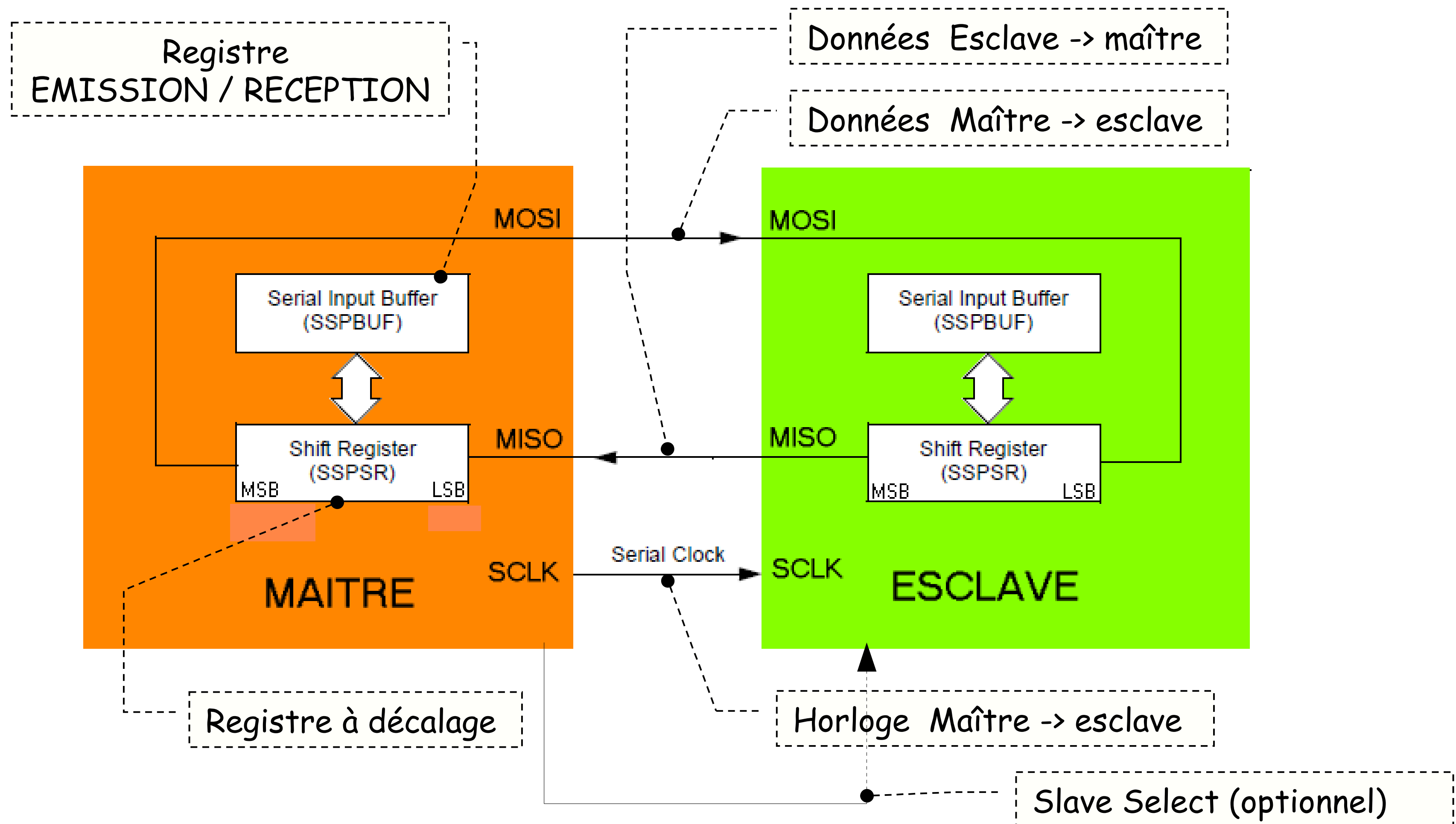
Lecture de deux octets Msb puis Lsb :

```
uint16_t data ;

Wire.requestFrom(address, 2); // request 2 bytes from slave
delay(mini) ;                // wait for data is ready
if (Wire.available()>=2) {
    data = Wire.read(); // read msb
    data <<= 8 ; // shift msb
    data |= Wire.read(); // read lsb and add to msb
 }
```


© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Le Bus SPI permet l'échange bidirectionnel de données sur le mode Maître/Esclave par une transmission *série synchrone full duplex*.



© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Les signaux

- **SCLK** (clock) : horloge

- **MOSI** (**M**aster **O**utput, **S**lave **I**nput)
  sortie donnée maître, entrée donnée esclave

- **MISO** (**M**aster **I**nput, **S**lave **O**utput)
  entrée donnée maître, sortie donnée esclave

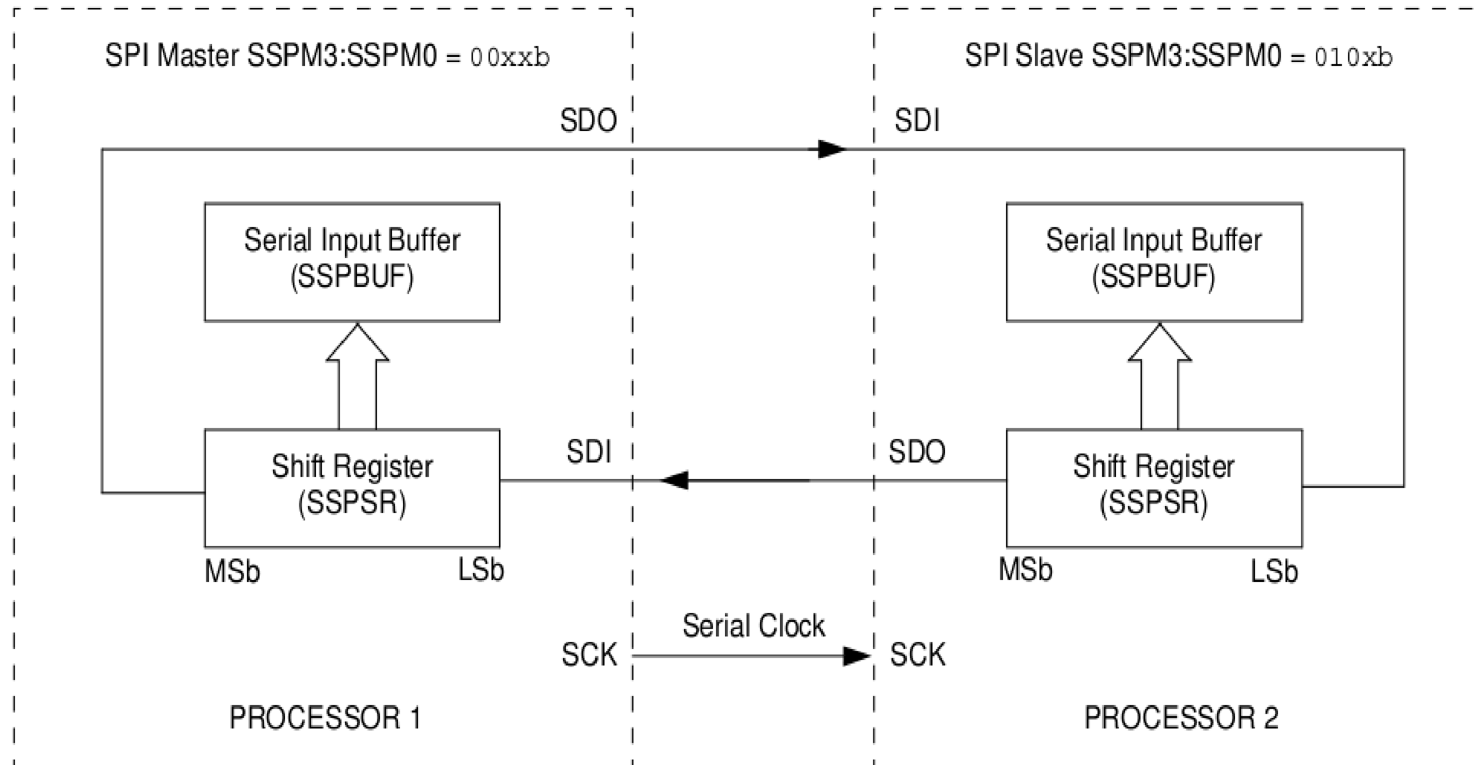
- **SS** (**S**lave **S**elect) sélection esclave

Les paramètres CPOL et CPHA déterminent quatre "modes" spi possibles. Il faut bien sûr que le mode soit indentiques sur le maître et l'esclave.

CPOL : polarité de l'horloge.
CPHA : front sur lequel le bit est échantillonné.

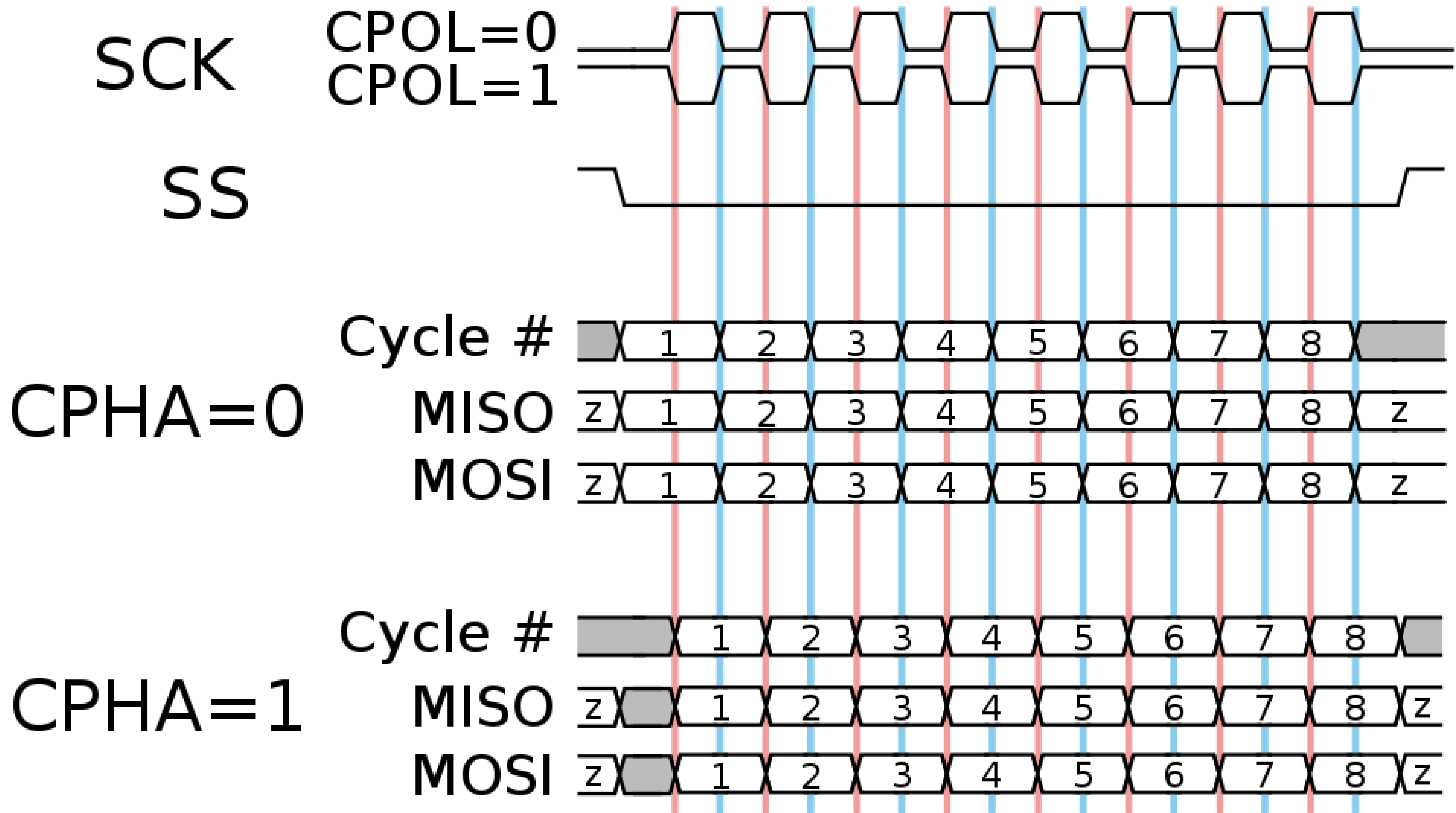
© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Périphérique SPI



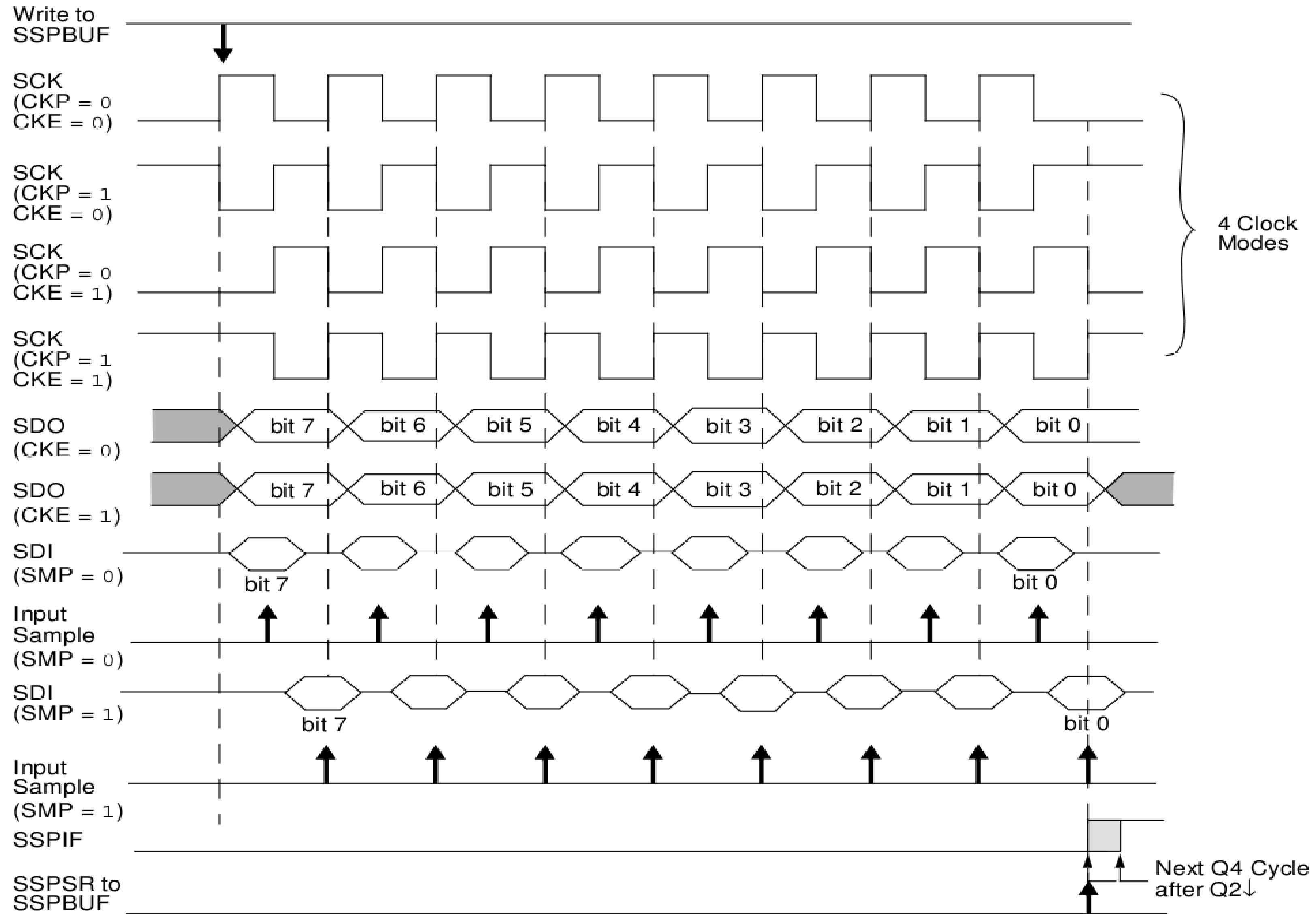
© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Chronogrammes des signaux



© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

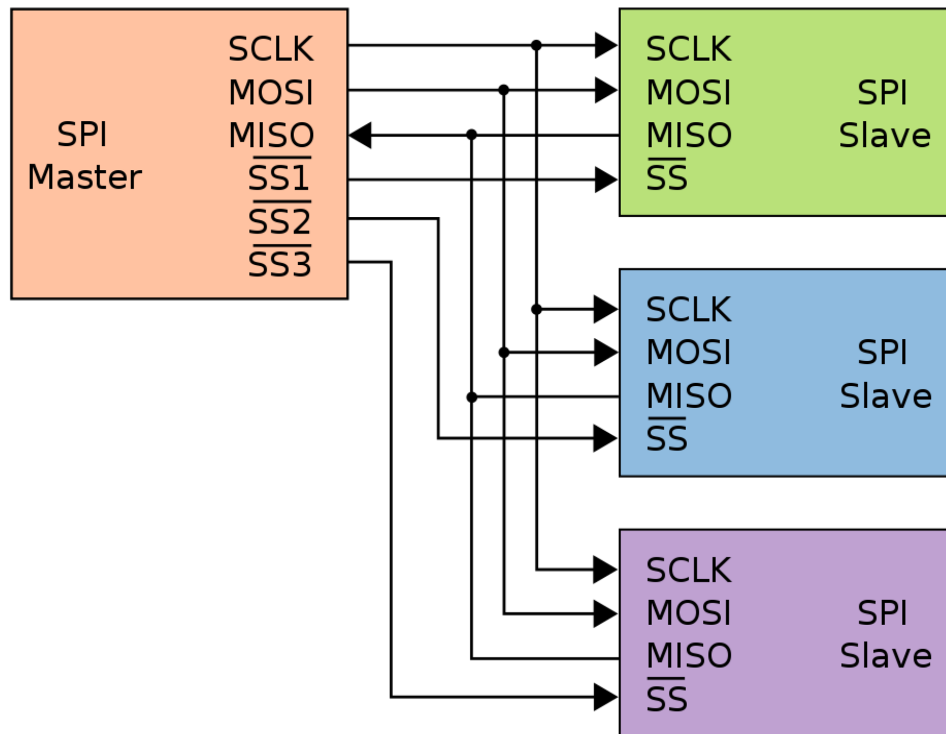
Chronogrammes des signaux



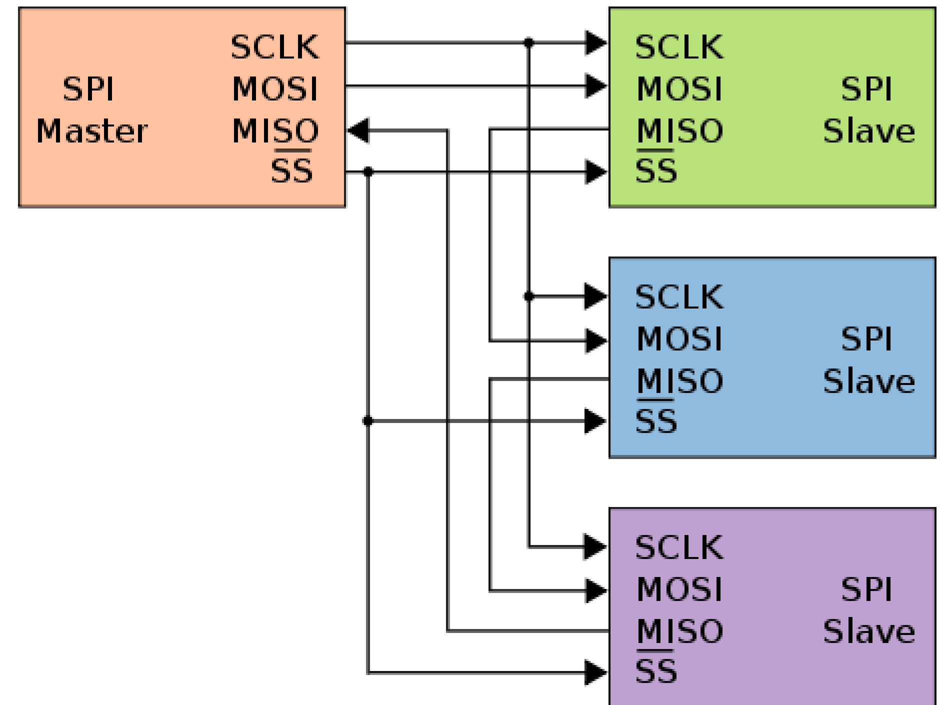
© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Connexion de plusieurs esclaves

Esclaves en parallèle
Sélection par /SS
(la plus courante)



Daisy Chain : la sortie d'un esclave alimente l'entrée du suivant.



© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

SPI Arduino

On utilise la librairie SPI : (voir https://www.arduino.cc/en/reference/SPI)

```
#include <SPI.h>
void setup() {
  SPI.begin();
  SPI.beginTransaction(SPISettings(14000000, MSBFIRST, SPI_MODE0))

}
```

Ensuite on utilise la méthode transfert() pour lire écrire simultanément (transfer16() pour des données 16bits):

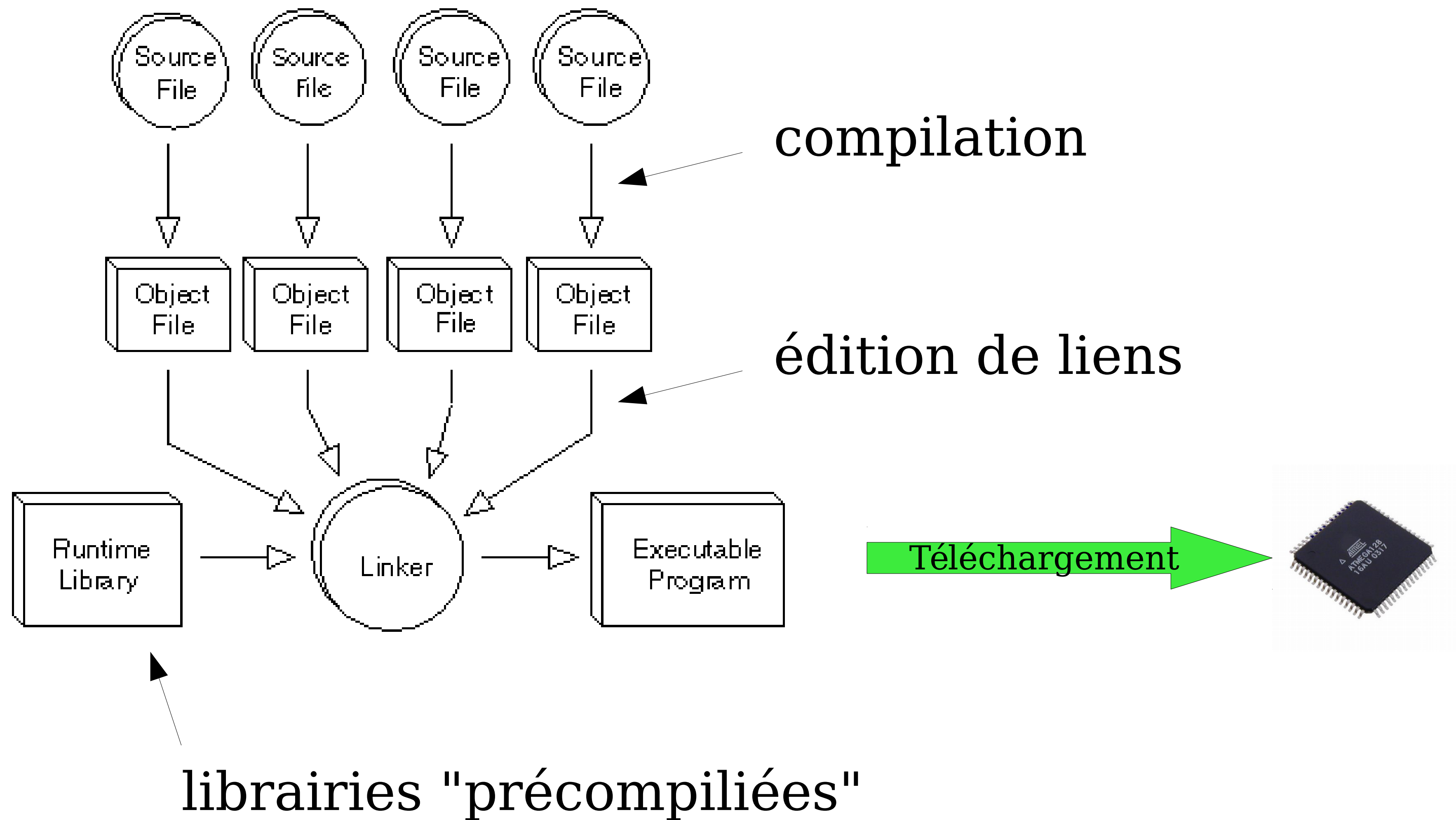
```
uint8_t valTx , valRx ;

 digitalWrite(SS,LOW);  // sélection du circuit esclave
 valRx = SPI.transfer(valTx); // transfert
 digitalWrite(SS,HIGH); // désélection de l'esclave
```

Rq : si on veut seulement lire on peut envoyer 0x00,
si on veut seulement écrire on ne tient pas compte de la valeur lue.

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

Chaîne de développement



© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license

commande make, Makefile

La commande make permet d'automatiser la production du fichier exécutable en prenant la description des opérations à réaliser dans un fichier Makefile.

```
hello: hello.o main.o
	gcc -o hello hello.o main.o

hello.o: hello.c
	gcc -o hello.o -c hello.c -W -Wall -ansi -pedantic

main.o: main.c hello.h
	gcc -o main.o -c main.c -W -Wall -ansi -pedantic

helloS: hello.o main.o
	gcc -static -o hello hello.o main.o

clean:
	rm -rf *.o
```

© Copyright 2005, Philippe Arlotto http://arlotto.univ-tln.fr
Creative Commons Attribution-ShareAlike 2.0 license