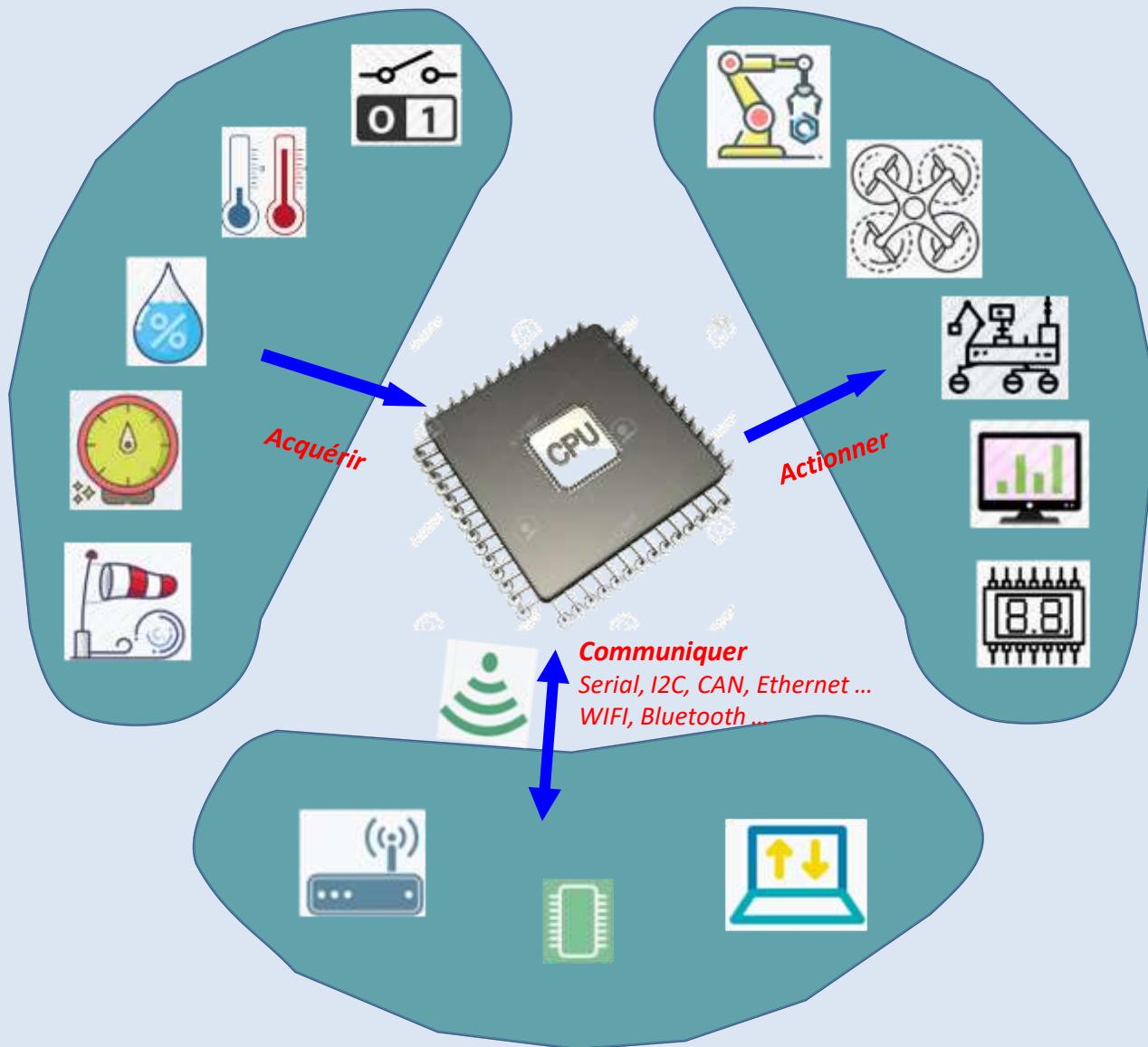
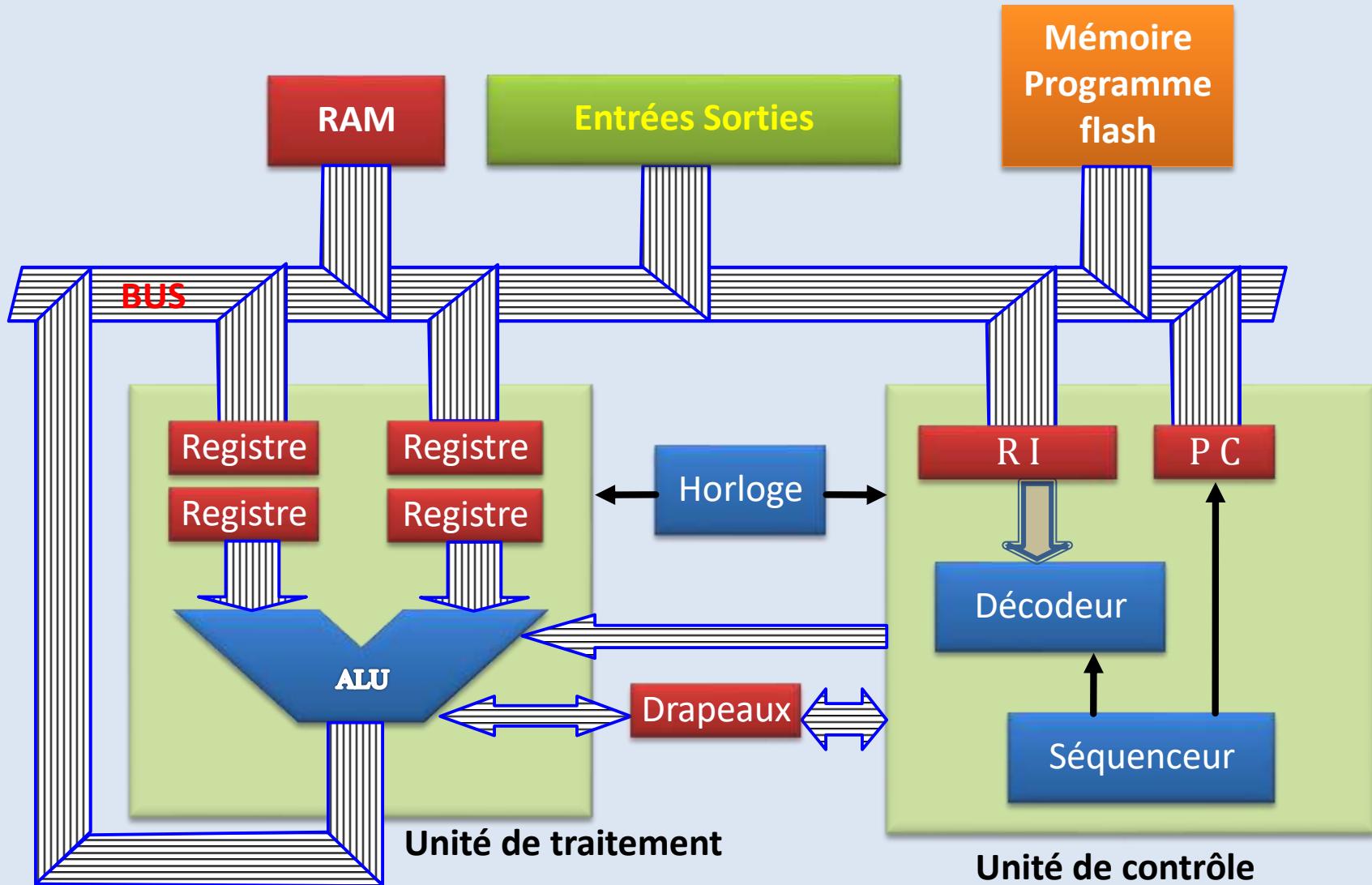


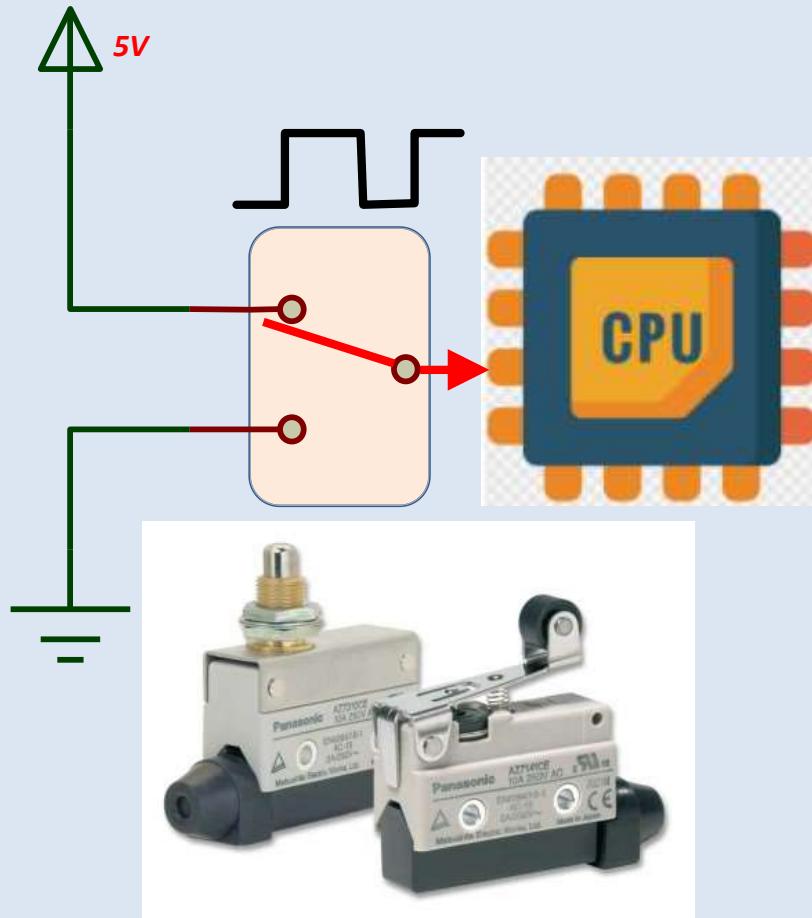
Structure générale d'un Système embarqué



Le CPU -> Microcontrôleur



Entrée Numérique



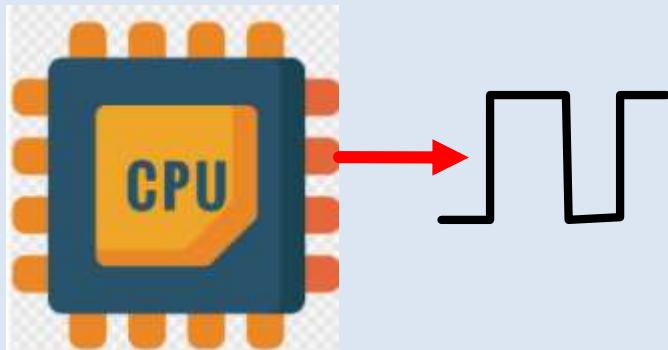
- **Entrée:** imposée par un équipement Externe
- **Numérique:** Ne peut prendre que deux valeurs; 0V ou 5V
- Bouton, interrupteur, contact de fin de course, capteur numérique ...

Sortie Numérique

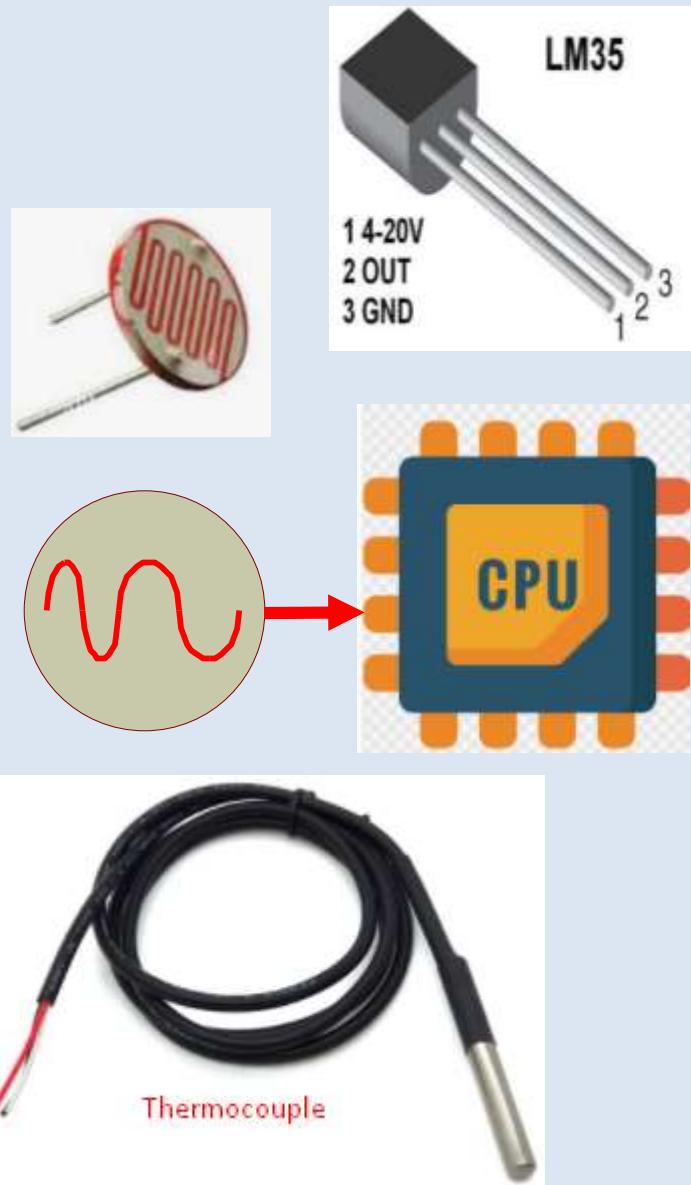
● **Sortie**: C'est le processeur qui la génère

● **Numérique**: Ne peut prendre que deux valeurs; 0V ou 5V

● **Usage**: Allumer/éteindre une LED, Démarrer/Arrêter un moteur ...



Entrée Analogique

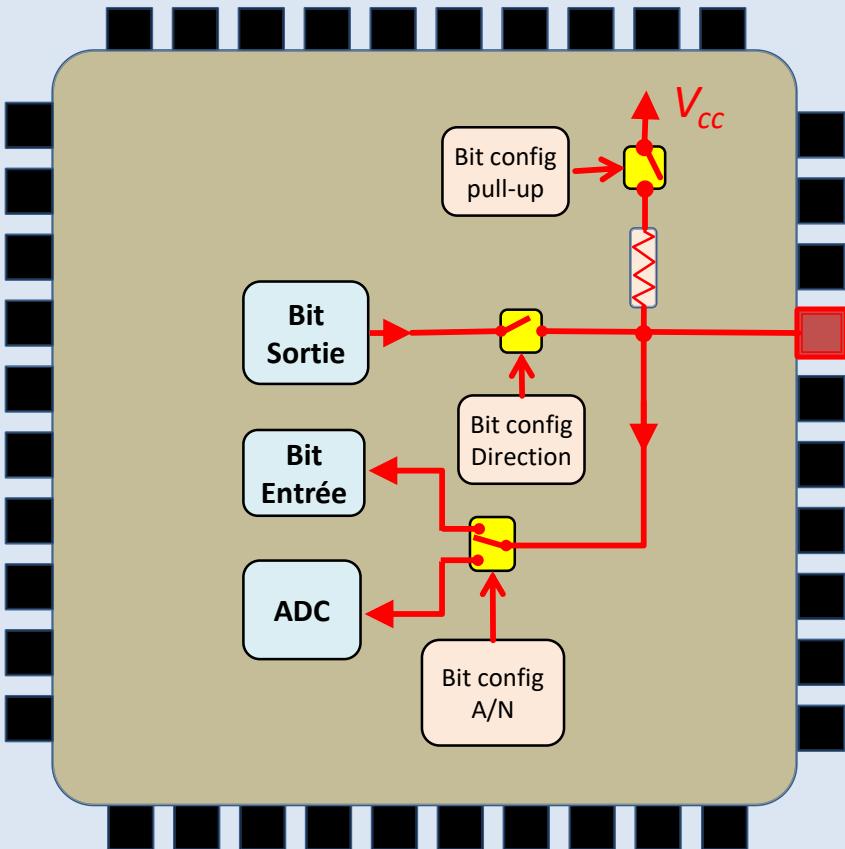


- **Entrée:** imposée par un équipement Externe
- **Analogique:** Peut prendre n'importe quelle valeur entre 0V et 5V
- **Source:** Capteur de température, Humidité, Pression, Lumière ...

Sortie Analogique

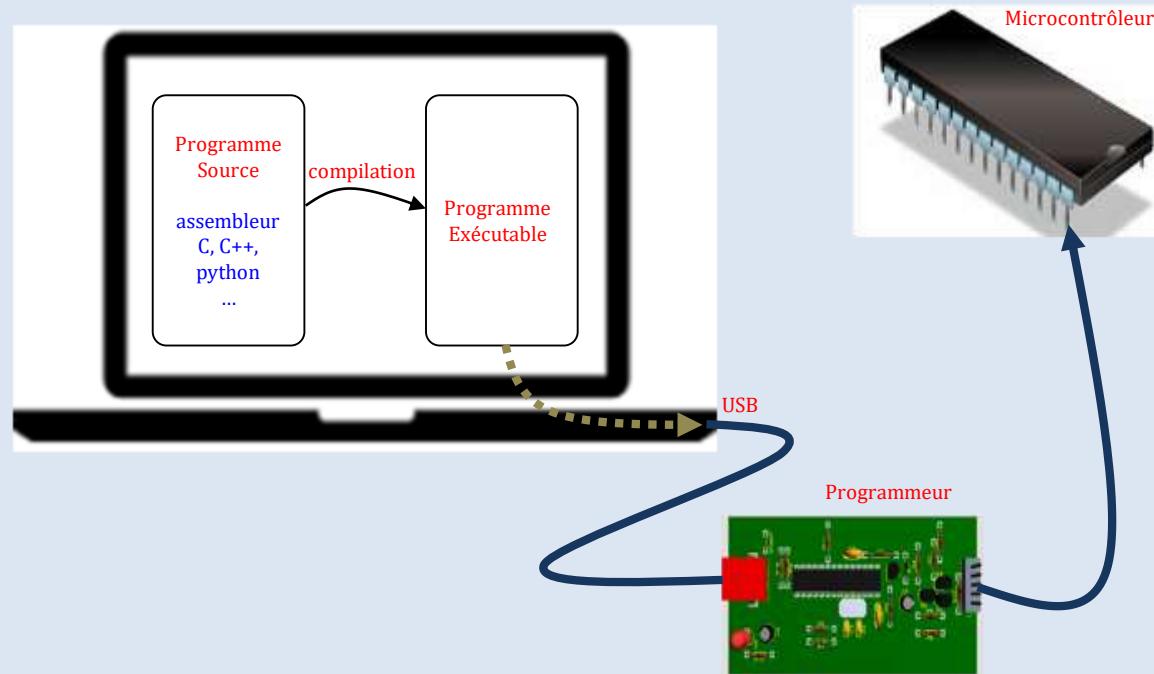
- En général, il n'y a pas de sortie analogique sur les Microcontrôleurs courants
- Les sorties PWM qui génèrent un signal carré à rapport cyclique réglable sont souvent appelées sorties pseudo-analogiques. On en parlera le moment venu

Entrée / Sortie



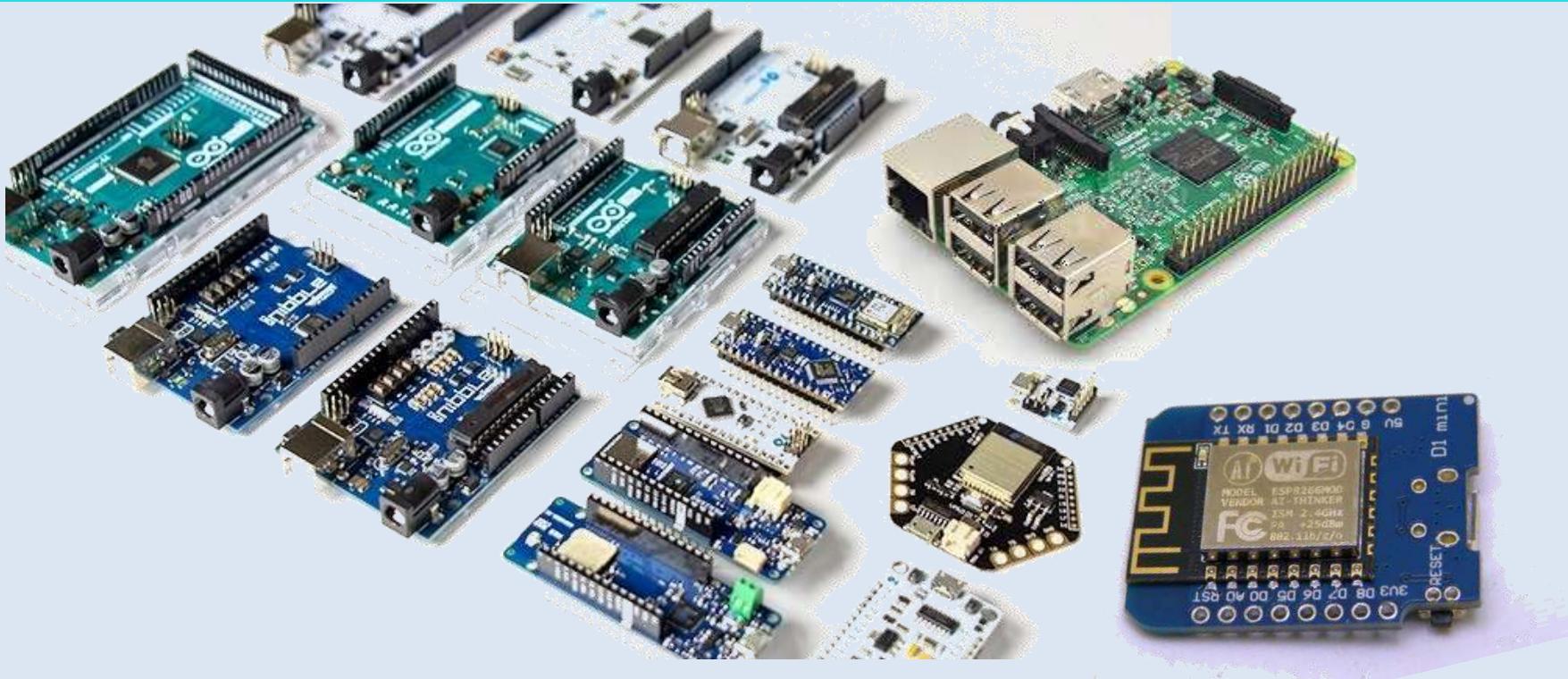
- ➊ Une Broche **E/S** peut être utilisée comme:
 - Sortie numérique
 - Entrée numérique
 - Entrée numérique avec Pull-Up
 - Entrée analogique

Travailler directement sur un microcontrôleur



- Permet d'exploiter toutes les ressources du microcontrôleur
- Pas très commode sur le plan pratique
- Peut souffrir de l'absence d'une communauté d'utilisateurs

Travailler sur une carte de prototypage



- Très grand éventail de choix de cartes
- Abondance de modules préfabriqués, capteurs, commande moteurs, affichage, communication ...
- Grande communauté d'utilisateurs
- Programmation haut niveau, portage facile

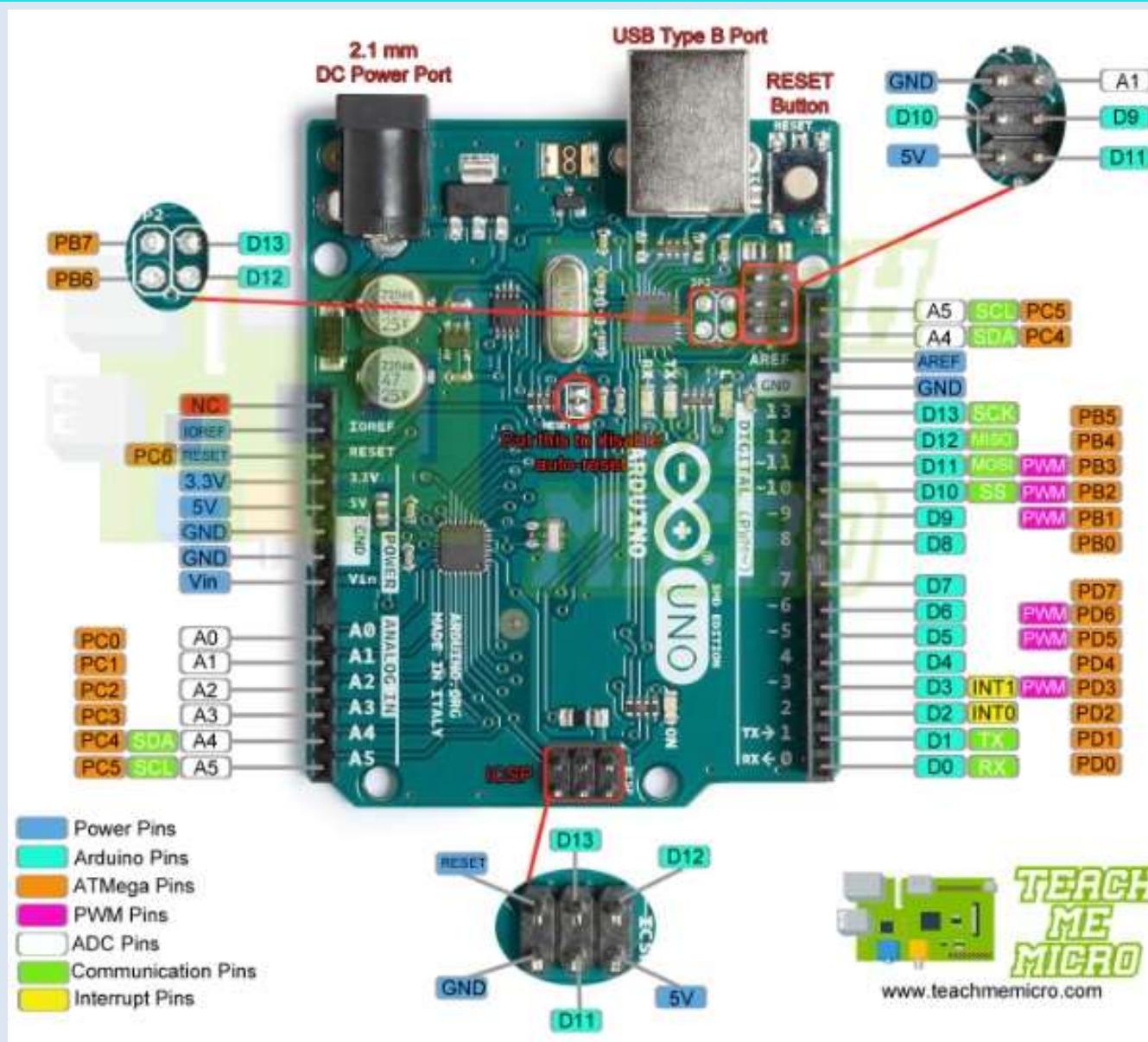
La carte Arduino UNO

- Processeur Atmega 328,
16MHz, 5V
- 32Ko Flash, 2Ko RAM
- 14 E/S numériques dont 6
PWM
- 6 entrées analogiques A0..A5
**(Peuvent être utilisée comme
E/S numériques)**
- Un bus de communication
Série Asynchrone (UART)
- Un Bus de communication
Synchrone I2C/SPI



- Une sortie peut fournir
jusqu'à 40 mA max
- l'ensemble des sorties
peut fournir 200 mA max
- l'ensemble des sorties
peut recevoir 400 mA max

Arduino UNO pinout

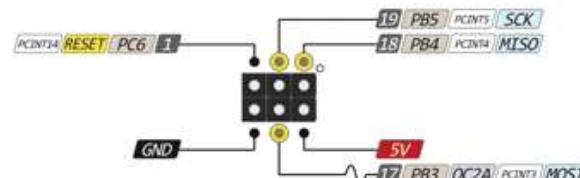
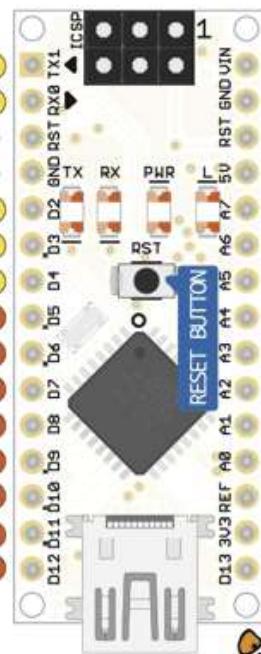


Arduino nano

NANO PINOUT

1
0
2
3
4
5
6
7
8
9
10
11
12

PCINT17 TXD PD1 31
PCINT16 RXD PD0 30
PCINT14 RESET PC6 29
GND
PCINT18 INT0 PD2 32
OC2B PCINT19 INT1 PD3 1
XCK PCINT20 T0 PD4 2
OC0B PCINT21 T1 PD5 9
OC0A PCINT22 AIN0 PD6 10
PCINT23 AIN1 PD7 11
ICP1 PCINT9 CLK0 PB0 12
PCINT1 OC1A PB1 13
SS PCINT2 OC1B PB2 14
MOSI PCINT3 OC2 PB3 15
MISO PCINT4 PB4 16



VIN The input voltage to the board when it is running from external power. Not USB bus power.

5V

A7

A6

19 A5

18 A4

17 A3

16 A2

15 A1

14 A0

13

- Power
- GND
- Serial Pin
- Analog Pin
- Control
- INT
- Physical Pin
- Port Pin
- Pin function
- Interrupt Pin
- PWM Pin
- Port Power

The power sum for each pin's group should not exceed 100mA

Absolute MAX per pin 40mA recommended 20mA

Absolute MAX 200mA for entire package



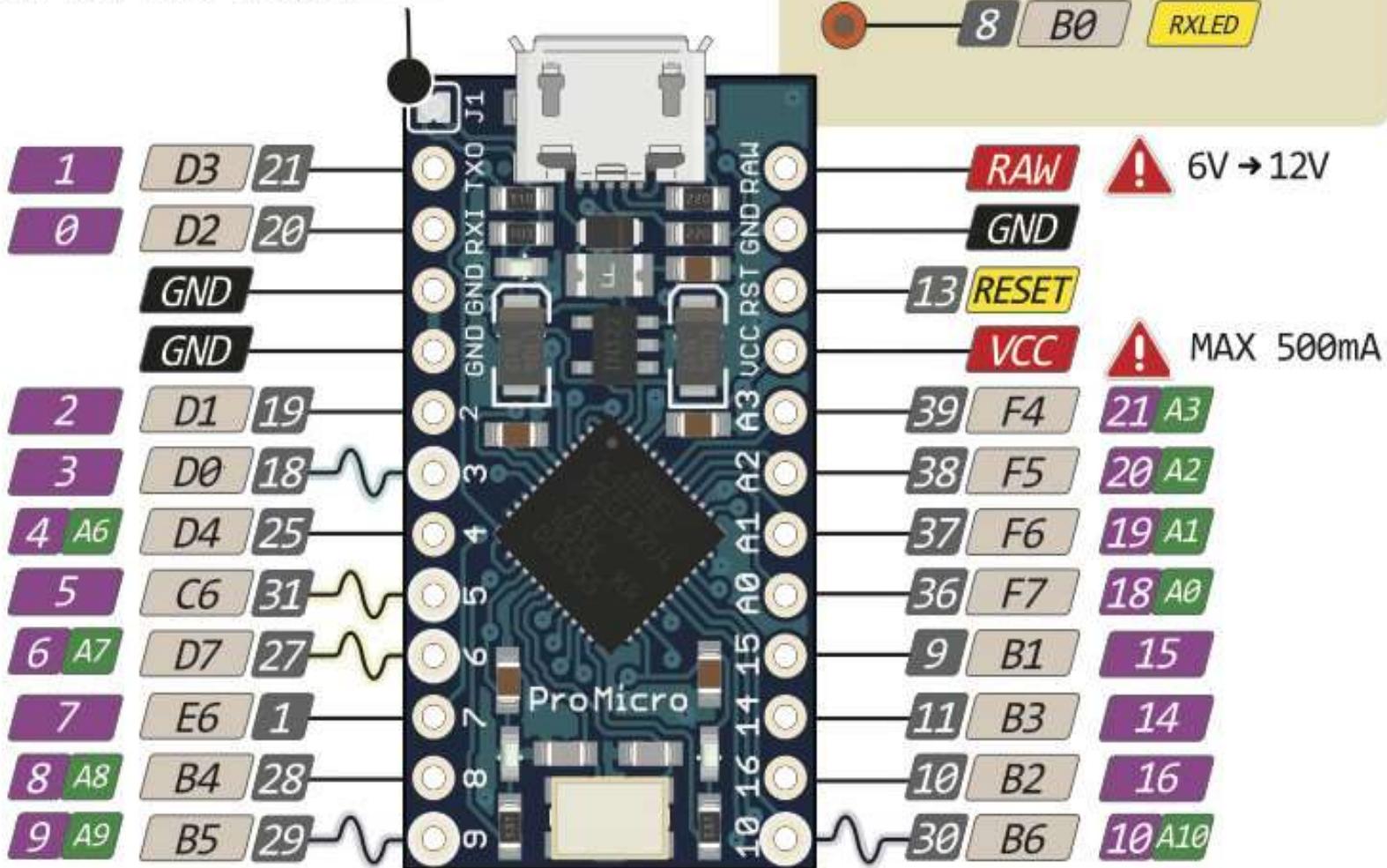
Analog exclusively Pins



19 AUG 2014
ver 3 rev 1

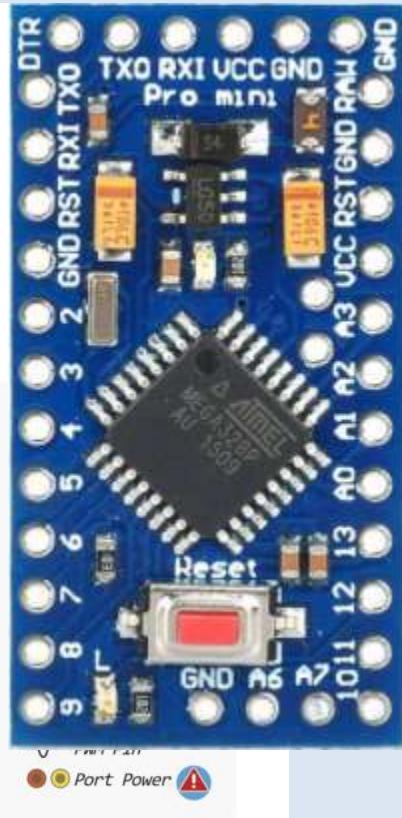
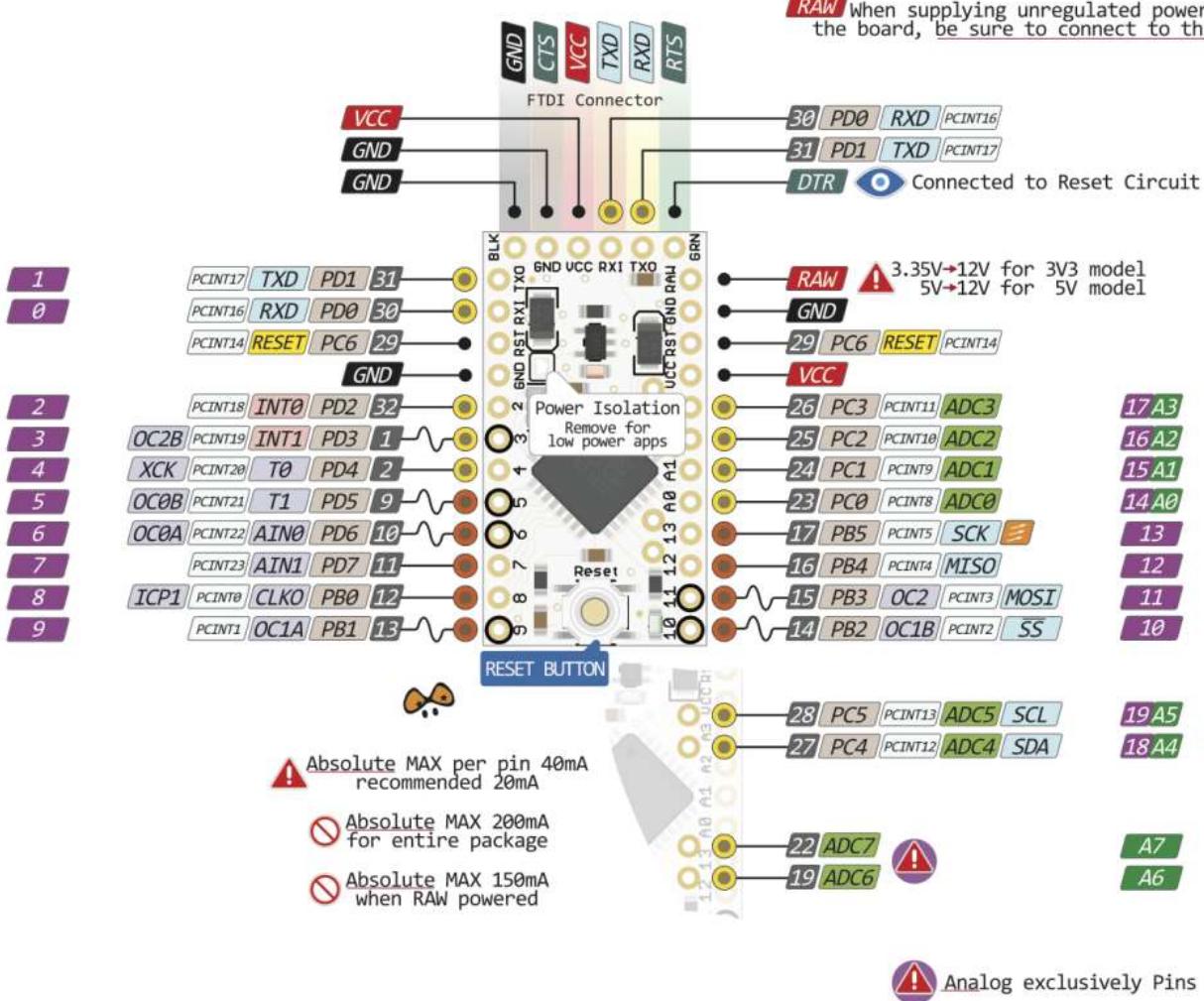
Arduino Pro Micro

Closed for 5V boards
Open for 3.3V boards



Arduino Pro mini

PRO MINI PINOUT



bq
www.bq.com
18 JUL 2014
ver 3 rev 0

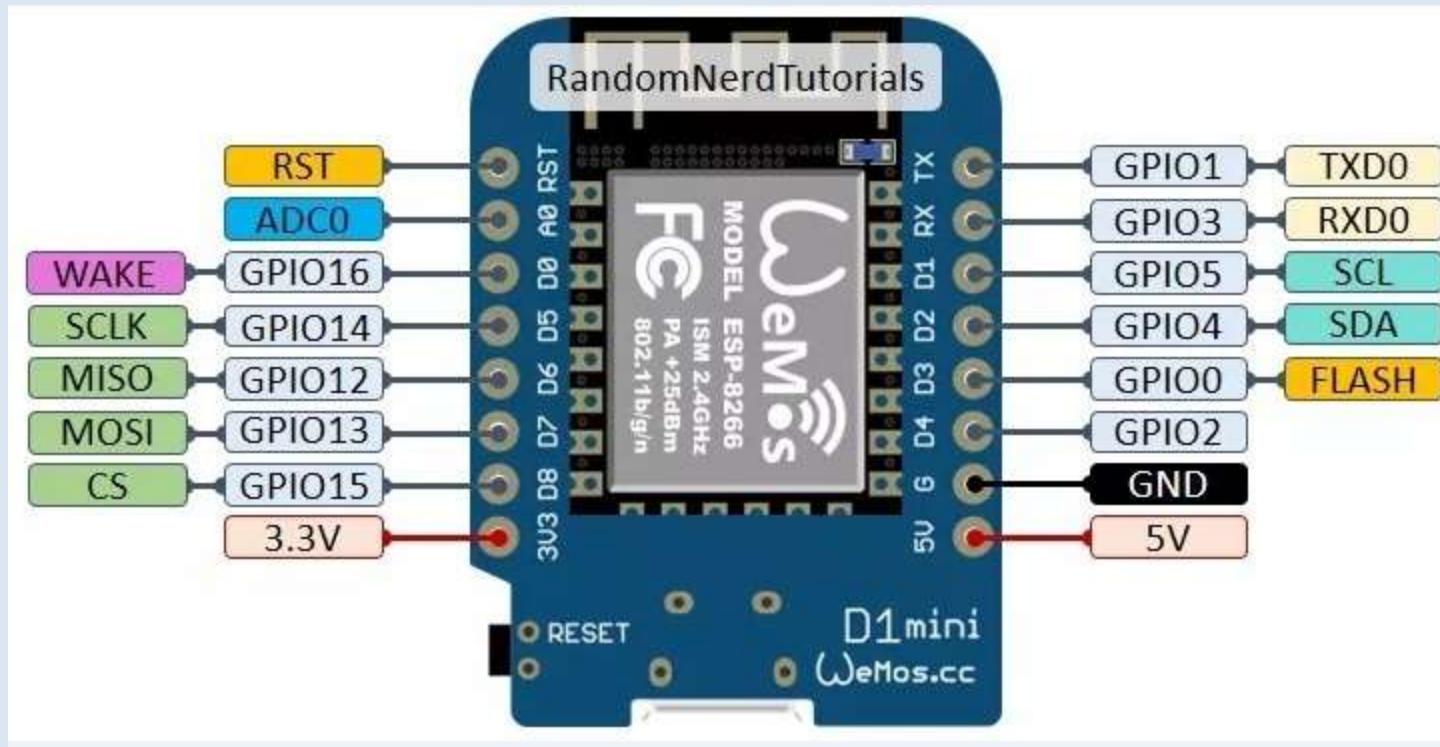
La carte Wemos D1-mini

- Processeur ESP8266 32bits, 80MHz/160MHz
- 4Mo Flash, 64ko RAM instructions, 96ko RAM données
- 11 E/S numériques 3.3V, tous PWM sauf la D0
- 1 entrée analogiques (3.2Vmax)
- Un bus de communication Série Asynchrone (UART)
- Un Bus de communication Synchrone I2C/SPI



- WIFI 802.11 b/g/n
- Peut être programmé comme:
 - Client,
 - Serveur,
 - Point d'accès

D1-mini pinout



Comparaison

Boards	Microcontroller	Operating Voltage/s (V)	Digital I/O Pins	PWM Enabled Pins	Analog I/O Pins	DC per I/O (mA)	Flash Memory (KB)	SRAM (KB)	EEPROM (KB)	Clock (MHz)	Length (mm)	Width (mm)	Cable	Native Network Support
Uno	ATmega328	5	14	6	6	20	32	2	1	16	68.6	53.4	USB A-B	None
Leonardo	ATmega32u4	5	20	7	12	40	32	2.5	1	16	68.6	53.3	micro-USB	None
Micro	ATmega32u4	5	20	7	12	40	32	2.5	1	16	48	18	micro-USB	None
Nano	ATmega328	5	22	6	8	40	32	2	0.51	16	45	18	mini-B USB	None
Mini	ATmega328	5	14		6	20	32	2	1	16	30	18	USB-Serial	None
Due	Atmel SAM3X8E ARM Cortex-M3 CPU	3.3	54	12	12	800	512	96	X	84	102	53.3	micro-USB	None
Mega	ATmega2560	5	54	15	16	20	256	8	4	16	102	53.3	USB A-B	None
M0	Atmel SAMD21	3.3	20	12	6	7	256	32	X	48	68.6	53.3	micro-USB	None
Yun Mini	ATmega32u4	3.3	20	7	12	40	32	2.5	1	400	71.1	23	micro-USB	Ethernet/Wifi
Uno Ethernet	ATmega328p	5	20	4	6	20	32	2	1	16	68.6	53.4	Ethernet	Ethernet
Tian	Atmel SAMD21	5	20	12	0	7	16000	64000	X	560	68.5	53	micro-USB	Ethernet/Wifi
Mega ADK	ATmega2560	5	54	15	16	40	256	8	4	16	102	53.3	USB A-B	None
M0 Pro	Atmel SAMD21	3.3	20	12	6	7	256	32	X	48	68.6	53.3	micro-USB	None
Industrial 101	ATmega32u4	5	7	2	4	40	16000	64000	1	400	51	42	micro-USB	Ethernet/Wifi
Uno Wifi	ATmega328	5	20	6	6	20	32	2	1	16	68.6	53.4	USB A-B	Wifi
Leonardo Ethernet	ATmega32u4	5	20	7	12	40	32	2.5	1	16	68.6	53.3	USB A-B	Ethernet
MKR1000	Atmel SAMD21	3.3	8	12	7	7	256	32	X	48	64.6	25	micro-USB	Wifi

Source: Arduino Boards, Compared - Tutorial Australia

Logiciel de programmation

- Il existe différents Environnements de programmation utilisant différents langages de programmation: assembleur, C, C++, Python...
- On va utiliser l'environnement de programmation gratuit **ARDUINO-IDE**

<https://www.arduino.cc/en/Main/Software>

- Supporte un grand nombre de carte de développement,
- Disponibilité d'une multitudes de librairies,

Configuration de l'IDE

- Choisir un dossier pour vos programmes:
 - Utilisez l'explorateur Windows pour créer un dossier de votre choix
 - Dans l'IDE Arduino: *fichier → préférences → Sélectionnez votre dossier dans le cadre: Emplacement du carnet de croquis → OK*
- Choisir la carte sur laquelle on travaille: *outils → type de carte → sélectionner votre carte dans la liste*
- Définir le port sur lequel est connecté la carte : *outils → port → sélectionner le port (Voir gestion périphériques si vous ne savez pas)*

Travailler avec une E/S Numérique

● ***pinMode(pin, State);***

Configurer une E/S numérique en Entrée ou en Sortie

- ***pin:*** numéro de l'E/S (type = Integer)
- ***State:*** nature de la broche d'E/S
 - **OUTPUT** pour Sortie
 - **INPUT** pour Entrée,
 - **INPUT_PULLUP** pour Entrée avec pull-Up interne vers 5V

● ***digitalWrite(pin, State);***

Forcer une sortie au niveau haut (5V) ou au niveau bas (0V)

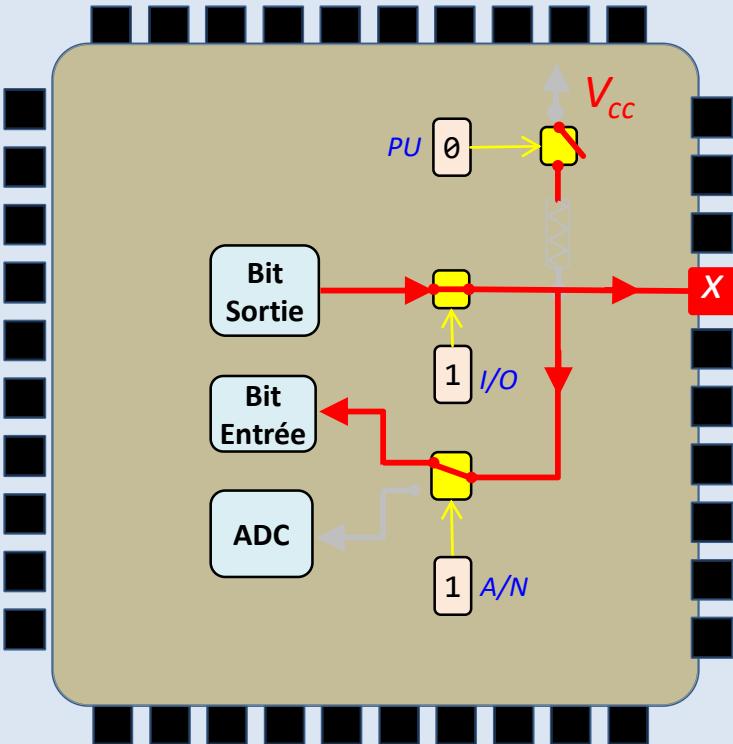
- ***pin:*** numéro de l'E/S (type = Integer)
- ***State:*** **HIGH** ou **LOW** (HIGH=1, LOW=0 -> constantes prédéfinies)

● ***val = digitalRead(pin);***

Retourne l'état d'une entrée numérique (0 ou 1)

- ***val:*** variable qui reçoit l'état de l'entrée
- ***pin:*** numéro de l'entrée (type = Integer)

Sortie: *pinMode(X,OUTPUT)*



● *digitalWrite(X,HIGH)*

Bit Sortie =1 => sortie=5V

● *digitalWrite(X,LOW)*

Bit Sortie=0 => sortie=0V

● *digitalRead(X)*

retourne la valeur du bit

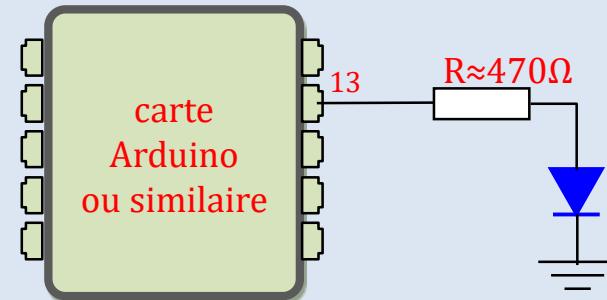
Entrée qui est égale au bit de sortie=état actuel de la sortie

- Il interdit d'appliquer une tension externe sur la sortie

Premier Programme

- ➊ Pour tester l'IDE et la carte. On écrit un programme pour faire clignoter la LED intégrée sur la carte (LED_BUILTIN)
- ➋ Avec l'IDE-ARDUINO, un programme doit avoir ces 2 fonctions: **setup()** et **loop()**
- ➌ La fonction **setup()** est exécutée une seule fois
- ➍ La fonction **loop()** est répétée à l'infini
- ➎ Sur l'Arduino, la LED est branchée sur la puce 13. Sur une autre carte on peut ne pas le savoir. On utilise la constante *LED_BUILTIN* qui prend la bonne valeur en fonction de la carte utilisée
- ➏ Ecrire et sauvegarder le programme 
- ➐ cliquez sur le bouton **téléverser**  pour envoyer le programme sur la carte
- ➑ Admirer le travail

```
void setup() {  
    pinMode(LED_BUILTIN, OUTPUT);  
}  
  
void loop() {  
    digitalWrite(LED_BUILTIN, HIGH);  
    delay(500);  
    digitalWrite(LED_BUILTIN, LOW);  
    delay(500);  
}
```



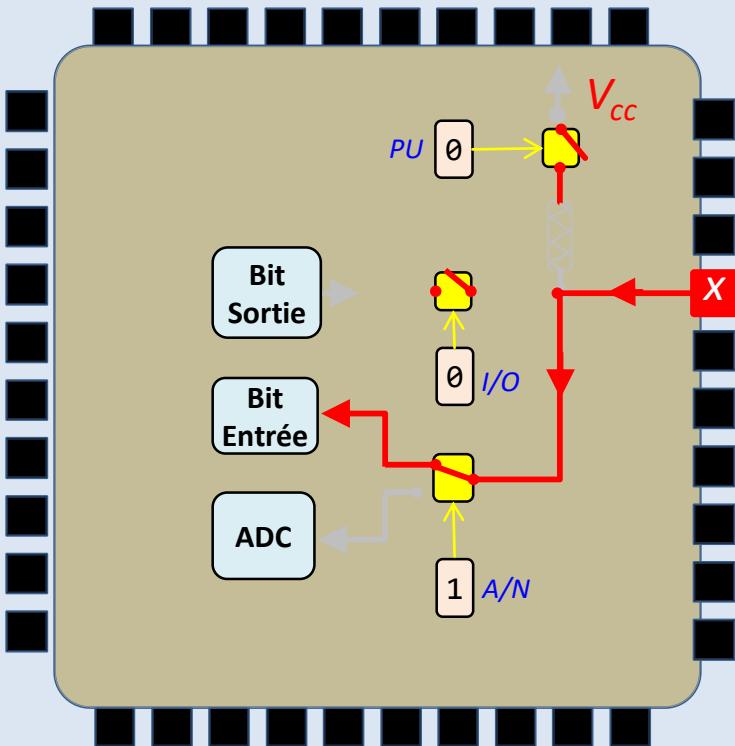
Analyse du programme

- Sur la carte Arduino, la LED intégrée est branchée sur la pate 13. Sur la carte D1mini, elle est branchée sur la pate 2. Pour cette raison, on utilise la constante `LED_BUILTIN` qui prend la bonne valeur selon la carte choisie
- Dans la fonction `setup()`, il y a une seule instruction qui configure la pate `LED_BUILTIN` en sortie,
- Dans la fonction `loop()`:
 - l'instruction `digitalWrite(LED_BUILTIN,HIGH)` place la sortie au niveau **haut** (Arduino->5V, D1mini->3.3V) => La LED s'allume ($R \approx 0.5k$)
 - L'instruction `delay(500);` arrête le programme pendant $\frac{1}{2}$ seconde
 - l'instruction `digitalWrite(LED_BUILTIN,LOW)` place la sortie au niveau **bas** (->0V) => La LED s'éteint
 - L'instruction `delay(500);` maintient la LED éteinte pendant $\frac{1}{2}$ s
 - la fonction `loop()` recommence ce qui fait clignoter La LED

Je n'ai pas de carte pour l'instant

- On peut utiliser le logiciel de simulation *Proteus ISIS*. Il suffit de lui ajouter une librairie Arduino. On peut en télécharger une ici <http://www.instructables.com/id/How-to-add-Arduino-Library-in-to-Proteus-7-8/>
- Copier les deux fichiers **.lib** et **.idx** dans le dossier ... \Labcenter Electronics\Proteus 7 Professional\LIBRARY
- Dans, l'IDE-ARDUINO, il ne faut pas téléverser. Il faut générer un programme exécutable **.hex**
croquis → Exporter les binaires compilés
- Le programme .hex doit être implanté dans l'Arduino au niveau du logiciel ISIS

Entrée: *pinMode(x, INPUT)*



◎ *digitalRead(x)*

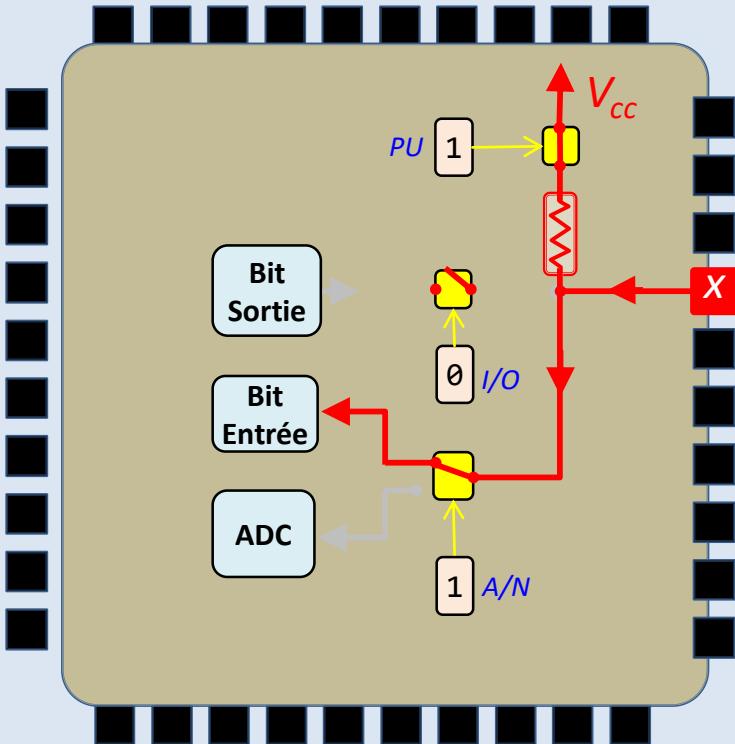
retourne la valeur du bit
Entrée qui est égale au niveau logique externe appliqué sur la broche x

◎ *digitalWrite(x,...)*

n'a aucun effet, bit Sortie est déconnecté

- ◎ Si on ne connecte rien sur la broche x , son état n'est pas défini. *digitalRead(x)* retourne une valeur aléatoire

Entrée: *pinMode(x,INPUT_PULLUP)*



○ *digitalRead(x)*

retourne la valeur du bit
Entrée qui est égale au niveau logique externe appliqué sur la broche x

○ *digitalWrite(x,...)*

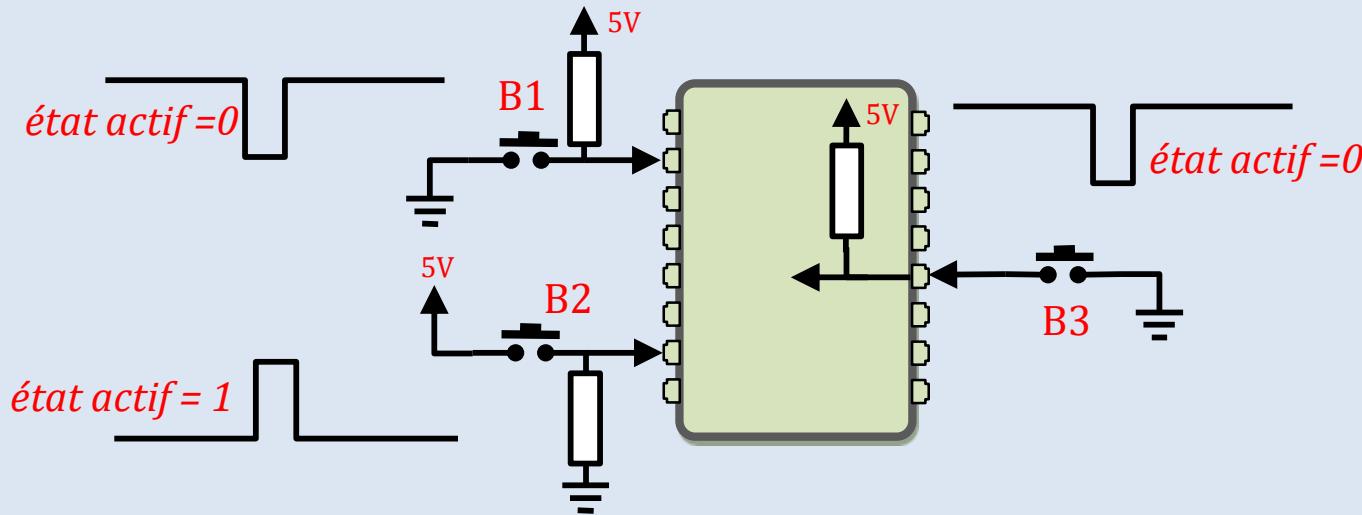
n'a aucun effet, bit Sortie est déconnecté

- Si on ne connecte rien sur la broche x , son état est fixé au niveau haut par la résistance de pullup.

digitalRead(x) retourne 1

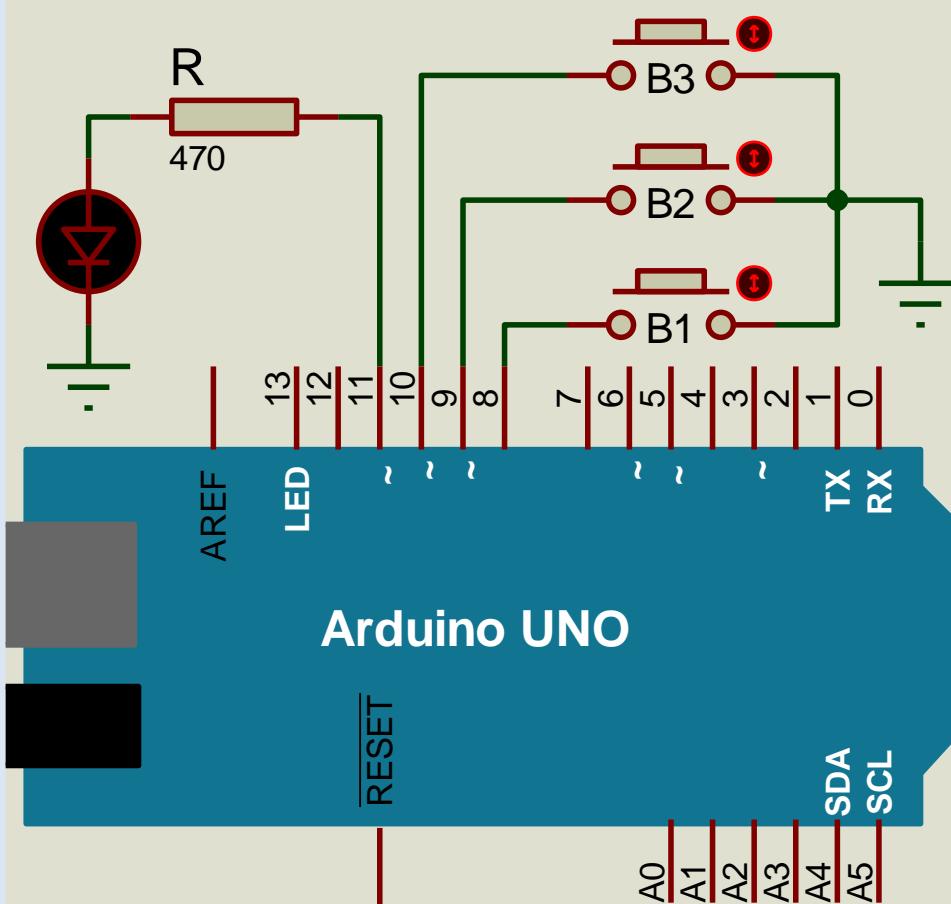
Boutons et interrupteurs

Travailler avec les boutons n'est pas chose aisée !!
dans la suite, nous considérons les boutons sans rebonds



- Selon le branchement, le contact peut délivrer soit un niveau haut soit un niveau bas
- On peut utiliser la résistance de Pull-Up interne
- L'instruction ***digitalRead()*** retourne l'état du contact

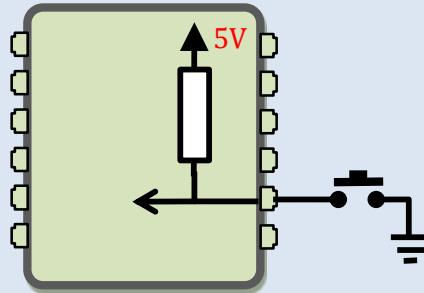
Exemple: 3 boutons, trois actions



- 3 boutons B1, B2, B3 sur les pates 8, 9, 10. Une LED sur la pate 11
- Clic sur B1, la LED clignote 5 fois au rythme 1s/1s
- Clic sur B2, la LED clignote 10 fois au rythme $\frac{1}{2}s / \frac{1}{2}s$
- Clic sur B3, la LED clignote 20 fois au rythme 200ms/200ms

Exemple: 3 boutons, trois actions (2)

- Pour les boutons, on choisit le branchement active-low avec résistance de pull-up interne



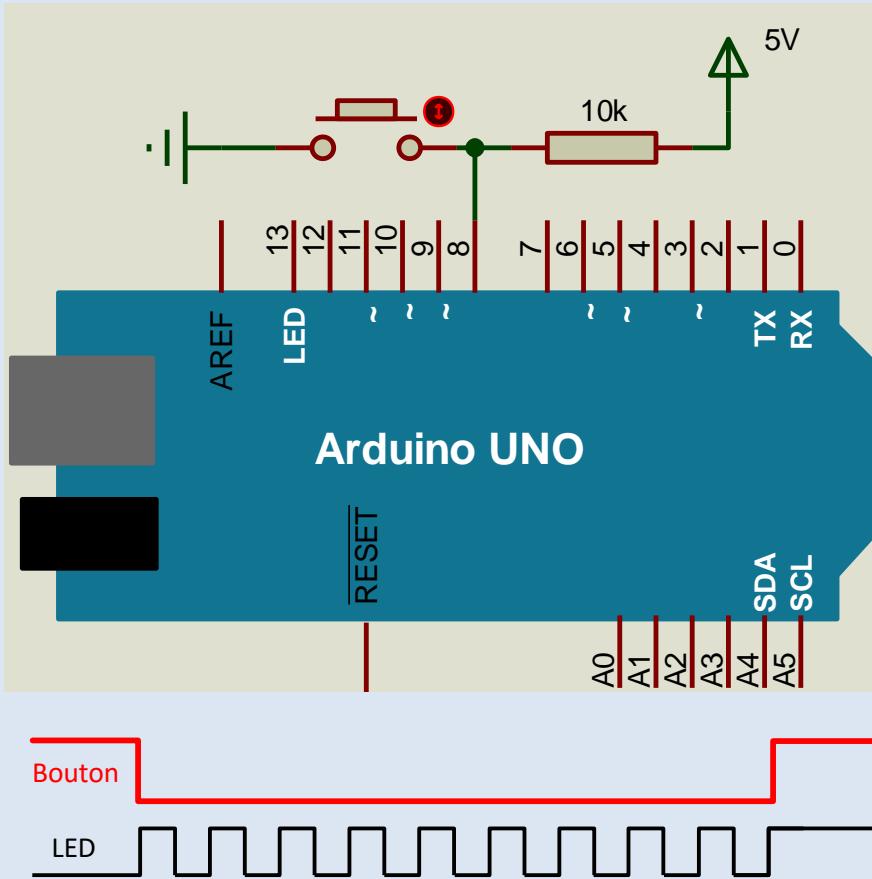
- Les pates 8, 9 et 10 doivent être configurées en conséquence

pinMode(num, INPUT_PULLUP)

- La pate 11 doit être configurée en sortie

```
#define B1PIN 8
#define B2PIN 9
#define B3PIN 10
#define LEDPIN 11
void setup() {
    pinMode(B1PIN, INPUT_PULLUP);
    pinMode(B2PIN, INPUT_PULLUP);
    pinMode(B3PIN, INPUT_PULLUP);
    pinMode(LEDPIN, OUTPUT);
}
void loop() {
    if(digitalRead(B1PIN)==0){
        ledflash(5,1000);
    }
    if(digitalRead(B2PIN)==0){
        ledflash(10,500);
    }
    if(digitalRead(B3PIN)==0){
        ledflash(20,200);
    }
}
void ledflash(int N, int T){
    for(int i=0; i<N; i++){
        digitalWrite(LEDPIN,HIGH);
        delay(T);
        digitalWrite(LEDPIN,LOW);
        delay(T);
    }
}
```

Exemple: Bouton allumer éteindre



On va utiliser un bouton sur la broche 8 pour allumer/éteindre la LED embarquée. (action courte)

```
#define BP_PIN 8
int LED_STATE = LOW;
void setup() {
    pinMode(BP_PIN, INPUT);
    pinMode(LED_BUILTIN, OUTPUT);
    digitalWrite(LED_BUILTIN, LED_STATE);
}

void Loop() {
    if(digitalRead(BP_PIN)==LOW){
        LED_STATE = 1 - LED_STATE;
        digitalWrite(LED_BUILTIN, LED_STATE);
    }
}
```

Ce programme ne marchera pas:

`digitalRead()` $\approx 3.5\mu s$, `digitalWrite()` $\approx 3.5\mu s \rightarrow \text{loop}() \approx 7\mu s$

1 clic normal $\approx 10ms$ à $100ms$ = plus de 1500 boucle `loop()` \rightarrow

Pendant un clic, la LED change d'état des millier de fois

Bouton allumer éteindre: proposition 1

Chaque fois qu'on détecte un clic, on change l'état de la LED et on attend que le bouton soit relâché avant de continuer.

while(digitalRead(BP_PIN)==LOW); →

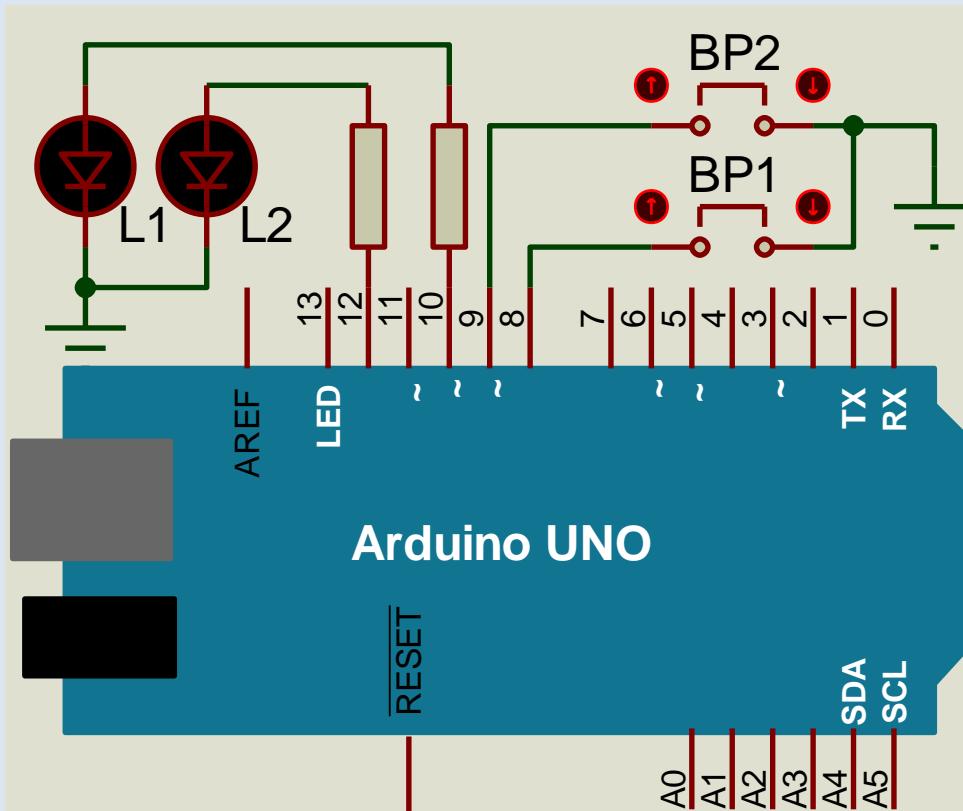
Tant que le bouton poussoir est appuyé, on ne fait rien, on attend

Cette action est bloquante et peut empêcher le programme de réaliser d'autres tâches si on reste appuyé trop longtemps sur un bouton

```
#define BP_PIN 8
int LED_STATE = LOW;
void setup() {
    pinMode(BP_PIN, INPUT);
    pinMode(LED_BUILTIN, OUTPUT);
    digitalWrite(LED_BUILTIN, LED_STATE);
}

void loop() {
    if(digitalRead(BP_PIN)==LOW){
        LED_STATE = 1 - LED_STATE;
        digitalWrite(LED_BUILTIN, LED_STATE);
        while(digitalRead(BP_PIN)==LOW);
    }
}
```

Problème de la proposition1



Si on reste appuyé sur un bouton, le programme ne voit plus ce qui se passe sur l'autre bouton

```
#define BP1 8
#define BP2 9
#define LD1 10
#define LD2 12
void setup() {
    pinMode(BP1, INPUT_PULLUP);
    pinMode(BP2, INPUT_PULLUP);
    pinMode(LD1,OUTPUT);
    pinMode(LD2,OUTPUT);
    digitalWrite(LD1,LOW);
    digitalWrite(LD2,LOW);
}
void loop() {
    static int LD1_ST = LOW, LD2_ST=LOW;
    if(digitalRead(BP1)==LOW){
        LD1_ST = 1 - LD1_ST;
        digitalWrite(LD1,LD1_ST);
        while(digitalRead(BP1)==LOW);
    }
    if(digitalRead(BP2)==LOW){
        LD2_ST = 1 - LD2_ST;
        digitalWrite(LD2,LD2_ST);
        while(digitalRead(BP2)==LOW);
    }
}
```

Proposition2: détection de transition

Au lieu de tester le niveau du bouton, on détecte les **transitions**,

A chaque passage dans `loop()`, on compare l'état du bouton avec l'état mémorisé lors du passage précédent.

Si 1 0 -> Transition ↓

Si 0 1 -> Transition ↑

chaque passage $\approx 4\mu s$



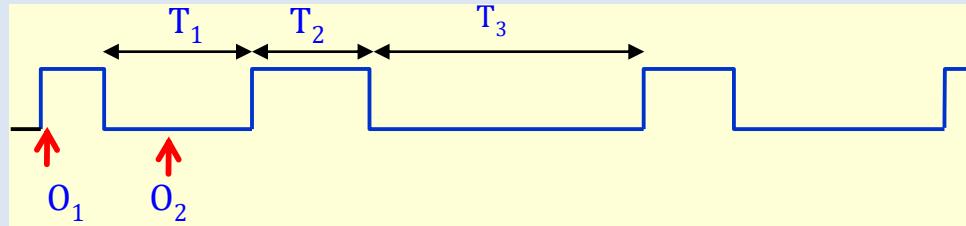
```
#define BP1 8
#define BP2 9
#define LD1 10
#define LD2 12
void setup() {
    pinMode(BP1, INPUT_PULLUP);
    pinMode(BP2, INPUT_PULLUP);
    pinMode(LD1,OUTPUT);
    pinMode(LD2,OUTPUT);
    digitalWrite(LD1,LOW);
    digitalWrite(LD2,LOW);
}
int LD1_ST = LOW, LD2_ST=LOW;
int B1_OLD_ST = HIGH, B2_OLD_ST=HIGH;
int B1_NEW_ST, B2_NEW_ST;
void loop() {
    B1_NEW_ST=digitalRead(BP1);
    if((B1_OLD_ST - B1_NEW_ST) == 1){
        LD1_ST = 1 - LD1_ST;
        digitalWrite(LD1,LD1_ST);
    }
    B1_OLD_ST=B1_NEW_ST;

    B2_NEW_ST=digitalRead(BP2);
    if((B2_OLD_ST - B2_NEW_ST) == 1){
        LD2_ST = 1 - LD2_ST;
        digitalWrite(LD2,LD2_ST);
    }
    B2_OLD_ST=B2_NEW_ST;
}
```

Gestion du temps

- ***delay(N);*** Arrête l'exécution du programme pendant ***N*** millisecondes (utilise Timer0)
- ***delayMicroseconds(N);*** Arrête l'exécution du programme pendant ***N*** microsecondes (n'utilise pas de timer)
- ***UL = millis();*** Retourne le nombre (unsigned long) de millisecondes écoulé depuis le début d'exécution du programme . Ce nombre va déborder (revenir à zéro) après environ 50 jours, (utilise Timer0)
- ***UL = micros();*** Retourne le nombre (unsigned long) de microsecondes depuis le début d'exécution du programme . Ce nombre va déborder (revenir à zéro) après environ 70mn. Sur Arduino UNO 16MHz, cette fonction avance avec un step de 4 μ s. (utilise Timer0).
- On peut aussi utiliser directement Timer0, Timer1 ,Timer2

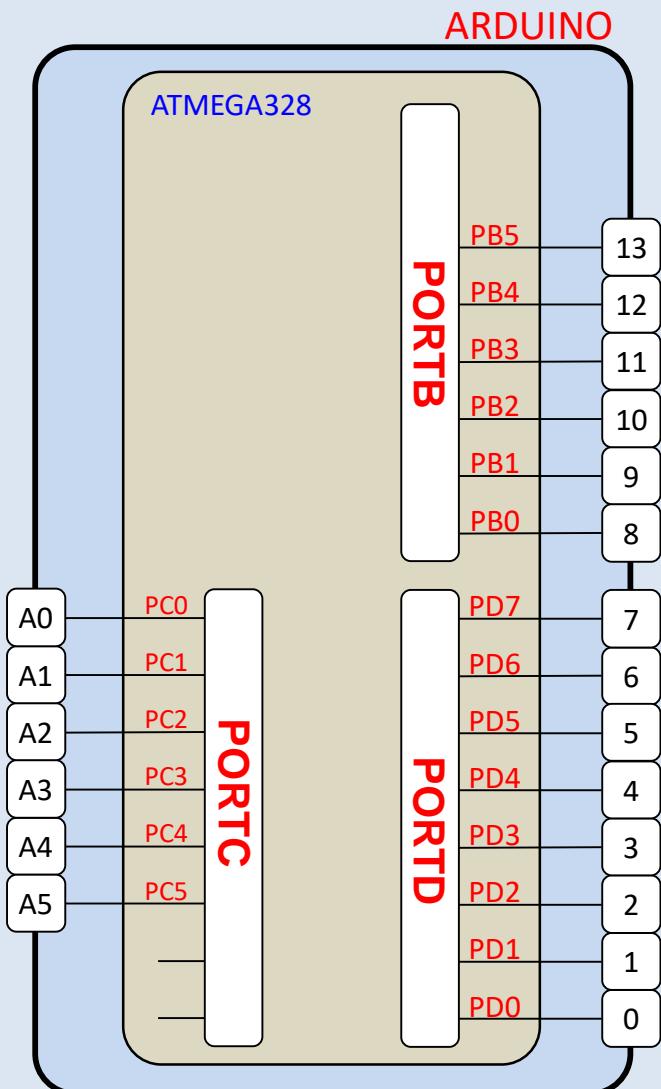
la fonction pulseIn()



Attend et retourne la largeur d'une impulsion positive ou négative en microsecondes.

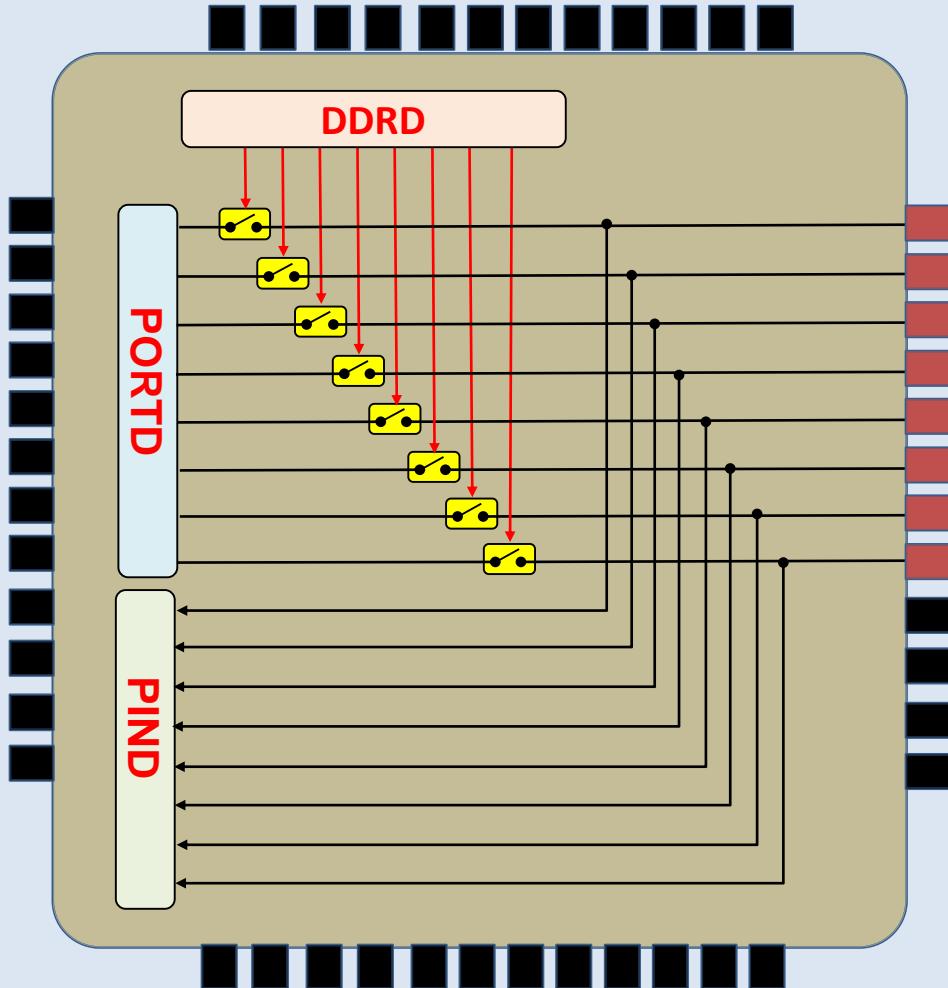
- $T = \text{pulseIn}(\text{pin}, \text{HIGH});$ L'appel de cette fonction à l'instant O_1 ou O_2 retourne T_2
- $T = \text{pulseIn}(\text{pin}, \text{LOW});$ L'appel de cette fonction à l'instant O_1 retourne T_1 . Son appel à l'instant O_2 retourne T_3 .
- Retourne 0 si aucune impulsion avant un timeout de 1s. on peut ajouter un paramètre pour fixer le timeout (en μs):
 $T = \text{pulseIn}(\text{pin}, \text{HIGH}, 10000);$

Accès direct aux ports d'E/S



- à l'intérieur du microcontrôleurs (μ C), les bits d'E/S sont regroupés dans des registres de 8 bits (Ports)
- La structure interne dépend du μ C
- Le Arduino UNO est construit autour du μ C ATMEGA328
 - les pates 0 à 7 → PORTD
 - les pates 8 à 13 → PORTB
 - Les pates A0 à A5 → PORTC
- En fait, chaque port est géré par 3 registres comme indiqué sur le slide suivant
- L'accès direct aux ports permet une exécution beaucoup plus rapide que l'utilisation des fonctions *pinMode()*, *digitalWrite()* et *digitalRead()*

Accès direct aux ports d'E/S (2)

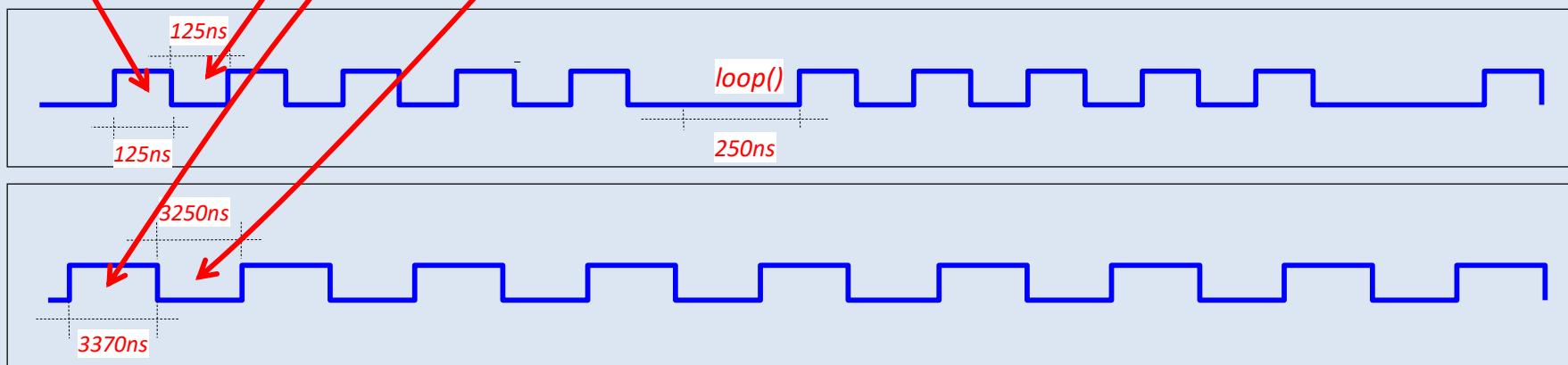


- DDRD est le registre de direction. L'instruction:
 - $DDRD = B11111111$; configure les 8 pates en sorties
 - $DDRD = B11110000$; configure 4 pates en sorties et 4 pates en entrées.
- PORTD est le registre de sortie. L'instruction:
 - $PORTD = B11111111$; force les 8 sorties à 1 (5V)
- PIND est le registre d'entrée. Si le port est configuré en entrée à l'aide du registre DDRD, L'instruction:
 - $x = PIND$; affecte le contenu des entrée à la variable x

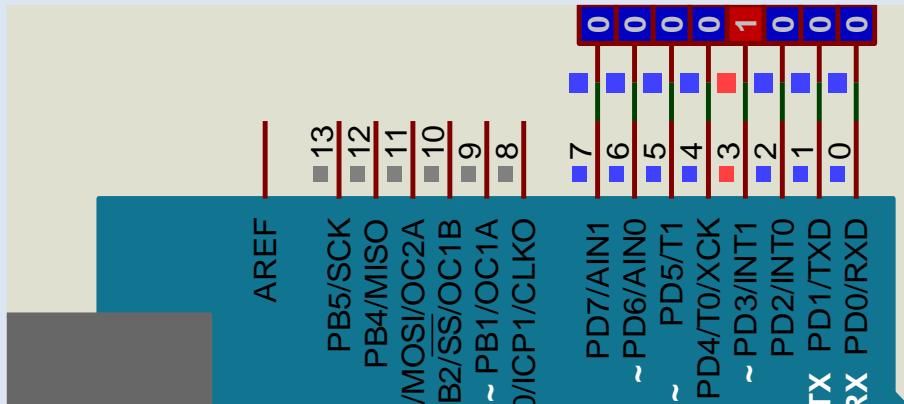
Plus rapide

```
void setup() {  
    DDRD = B00000001;  
}  
  
void loop() {  
    PORTD = PORTD | B00000001; // allumer  
    PORTD &= B11111110; // éteindre  
    PORTD |= B00000001;  
    PORTD &= B11111110;  
    PORTD |= B00000001;  
    PORTD &= B11111110;  
    PORTD |= B00000001;  
    PORTD &= B11111110;  
    PORTD |= B00000001;  
    PORTD &= B11111110;  
}
```

```
void setup() {  
    pinMode(0, OUTPUT);  
}  
  
void loop() {  
    digitalWrite(0, HIGH); // allumer la LED  
    digitalWrite(0, LOW); // éteindre LED  
    digitalWrite(0, HIGH); // allumer la LED  
    digitalWrite(0, LOW); // éteindre LED  
    digitalWrite(0, HIGH); // allumer la LED  
    digitalWrite(0, LOW); // éteindre LED  
    digitalWrite(0, HIGH); // allumer la LED  
    digitalWrite(0, LOW); // éteindre LED  
    digitalWrite(0, HIGH); // allumer la LED  
    digitalWrite(0, LOW); // éteindre LED  
}
```



Plus concis



```

int p;
void setup(){
    for(p = 0; p < 8; p++){
        pinMode(p, OUTPUT);
        digitalWrite(p, LOW);
    }
}
void loop(){
    for(p = 0; p < 8; p++){
        digitalWrite(p, HIGH);      // allumer
        delay(300);               // attendre un peu
        digitalWrite(p, LOW);     // éteindre
    }
}

```

- On branche 8 voyants sur les pates 0 à 7 (PORTD)
- On fait circuler un voyant allumé à droite ou à gauche

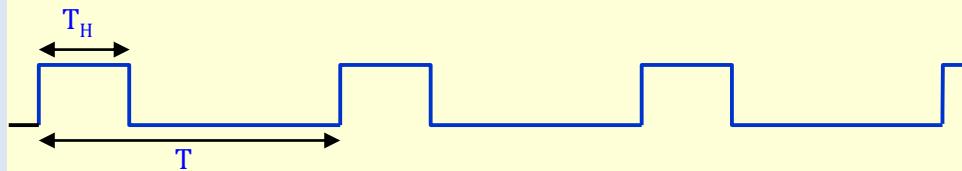
```

void setup() {
    DDRD = 0xFF; //PORTD sortie
    PORTD = B00000001;
}

void loop() {
    delay(300);
    PORTD = PORTD << 1 ;
    if (PORTD == B00000000)PORTD = B00000001;
}

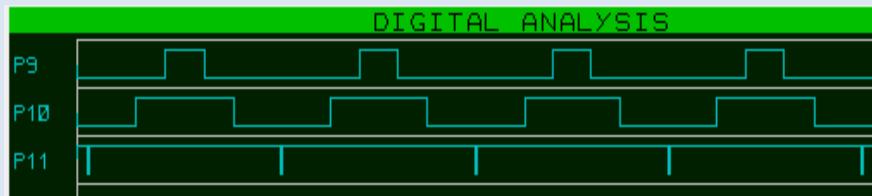
```

Signal PWM

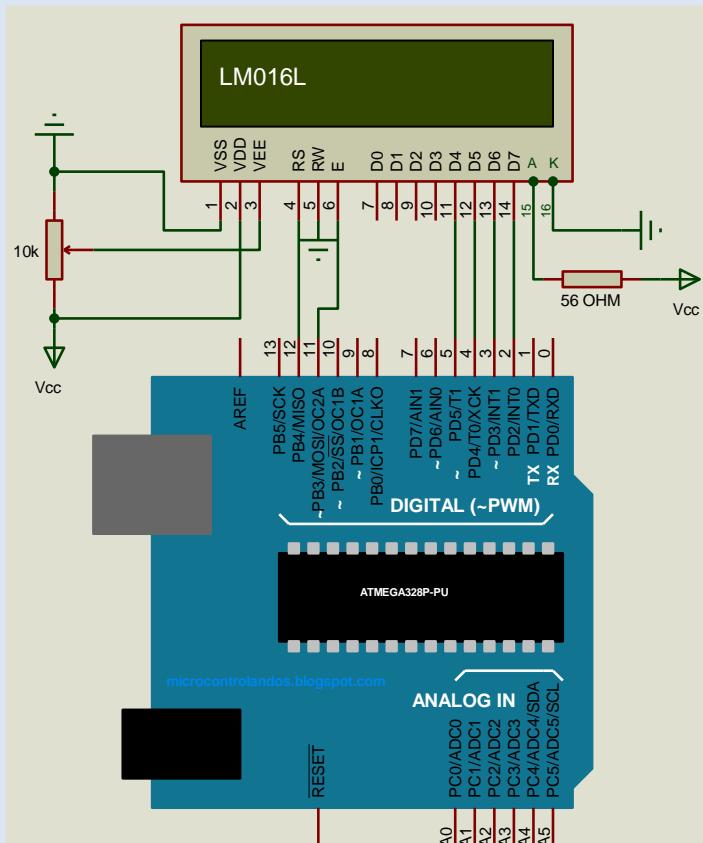
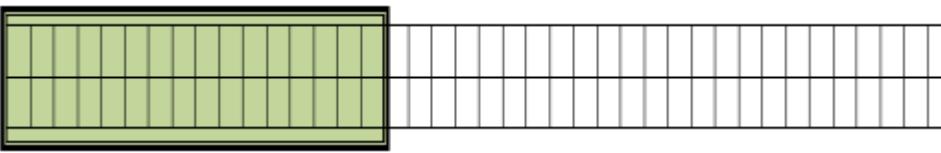


- Un signal PWM ou MLI est un signal dont le rapport cyclique T_H/T est réglable
- Contrôler la vitesse d'un moteur DC
Contrôler Servomoteur
Contrôler Moteur brushless
- Sur Arduino UNO, on peut générer facilement un signal PWM sur les pates 3, 5, 6, 9, 10 et 11
- `analogWrite(pin, R);`
 - pin : numéro de la pate PWM
 - R: entier compris entre 0 et 255 qui définit le rapport cyclique
- Pour arrêter le signal PWM, il suffit d'appeler `digitalWrite()`

```
/*
il me semble que la librairie ARDUINO pour ISIS
a un problème PWM sur les pates 3, 5 et 6
nous allons utiliser les pates 9, 10 et 11
*/
void setup()
{
    analogWrite(9,50);
    analogWrite(10,127);
    analogWrite(11,253);
}
void loop()
{}
```



Afficheur LCD



- Les afficheurs les plus courants sont les 2×16. Deux lignes de 40 caractères dont 16 sont visibles,
- Les caractères et les commandes sont envoyés sur un bus de 8 bits, mais on peut les utiliser en mode 4 bits,
- L'IDE Arduino intègre la bibliothèque *liquidcrystal* qui permet d'utiliser ces afficheurs sans en connaître le fonctionnement interne. L'afficheur est utilisé en mode 4 bits,

La librairie LiquidCrystal

● *LiquidCrystal lcd(RS, E, d4, d5, d6, d7)*

- Crée une classe (objet) pour utiliser l'afficheur
- *lcd*: nom l'objet. On n'est pas obligé de l'appeler *lcd*, on peut l'appeler ce que l'on veut
- RS, E, d4, d5, d6, d7 sont les numéros des pins utilisés pour contrôler l'afficheur
- Exemple: *LiquidCrystal lcd(12, 11, 5, 4, 3, 2);*

● *lcd.begin(Nc, Nl)*

- Initialise l'afficheur
- Nc: Nombre de caractères par ligne
- Nl: Nombre de ligne de l'afficheur

● *lcd.clear()*

- Efface l'afficheur et ramène le curseur au début de la première ligne

● *lcd.setCursor(col, lign)*

- Place le curseur à la position (col, lign), les deux paramètres commencent par 0

La librairie LiquidCrystal

● *Lcd.write(caractère)*

- Affiche un caractère. Exemples:
 - *Lcd.write('X');* // affiche le caractère X
 - *Lcd.write(65);* // Affiche le caractère A

● *Lcd.print(donnée,[base])*

- Affiche une chaîne ou un nombre. Exemples:
 - *Lcd.print("Hello world");* // affiche la chaîne Hello World
 - *Lcd.print(1024);* // Affiche le nombre 1024
 - *Lcd.print(85, BIN);* // affiche le nombre 85 en binaire --> 1010101
 - *Lcd.print(255, HEX);* // affiche le nombre 255 en hexadécimal --> FF
 - *Lcd.print(243.8765333);* // affiche le nombre réel avec 2 chiffres à droite de la virgule → 243.88
 - *Lcd.print(243.8765333,4);* // affiche le nombre réel avec 4 chiffres à droite de la virgule → 243.8765

La librairie LiquidCrystal

● *lcd.scrollDisplayLeft();*

- décaler l'affichage d'un caractère vers la gauche

● *lcd.scrollDisplayRight();*

- décaler l'affichage d'un caractère vers la droite

● *lcd.cursor(); ou lcd.noCursor();*

- Montrer ou cacher le curseur

● *lcd.blink(); ou lcd.noBlink();*

- *Clignotement du curseur*

● *lcd.display(); ou lcd.noDisplay();*

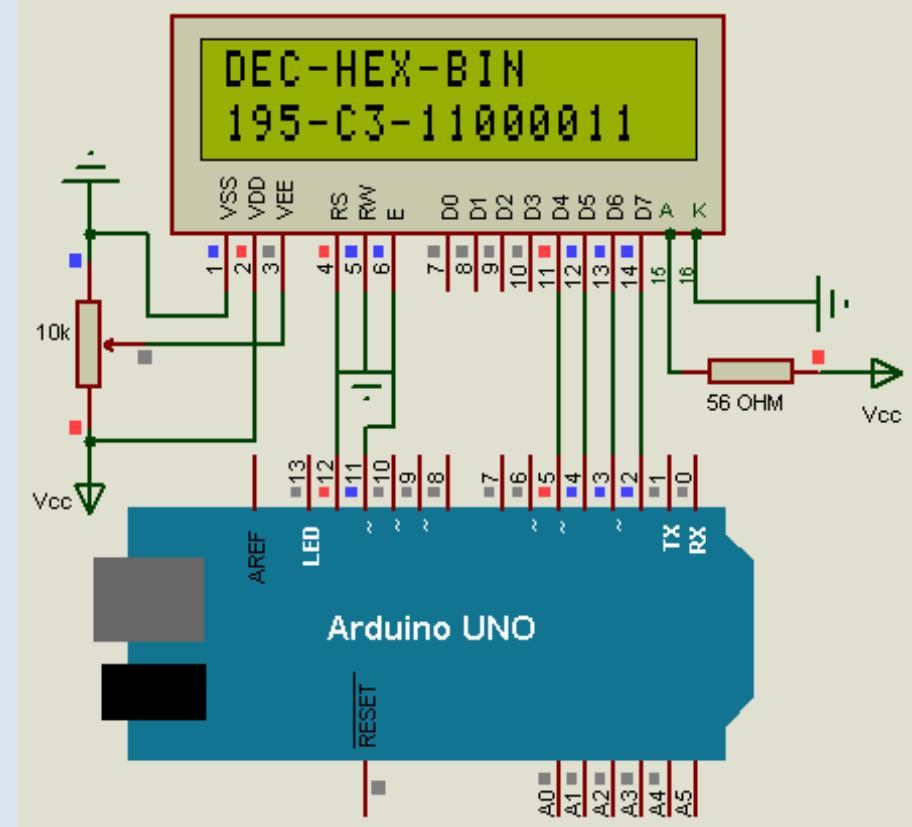
- Allumer ou éteindre l'afficheur. Le texte n'est pas perdu

LCD - Exemple

```
#include <LiquidCrystal.h>
LiquidCrystal lcd1(12, 11, 5, 4, 3, 2);

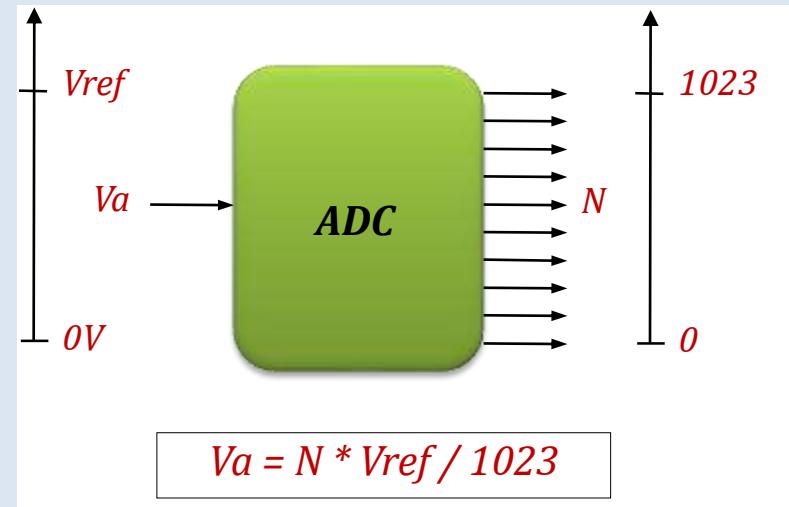
void setup() {
    int N = 195;
    lcd1.begin(16, 2);
    lcd1.print("DEC-HEX-BIN");
    lcd1.setCursor(0,1);
    lcd1.print(N);
    lcd1.write('-');
    lcd1.print(N,HEX);
    lcd1.write('-');
    lcd1.print(N,BIN);
}

void loop() { }
```



Lecture d'une entrée analogique

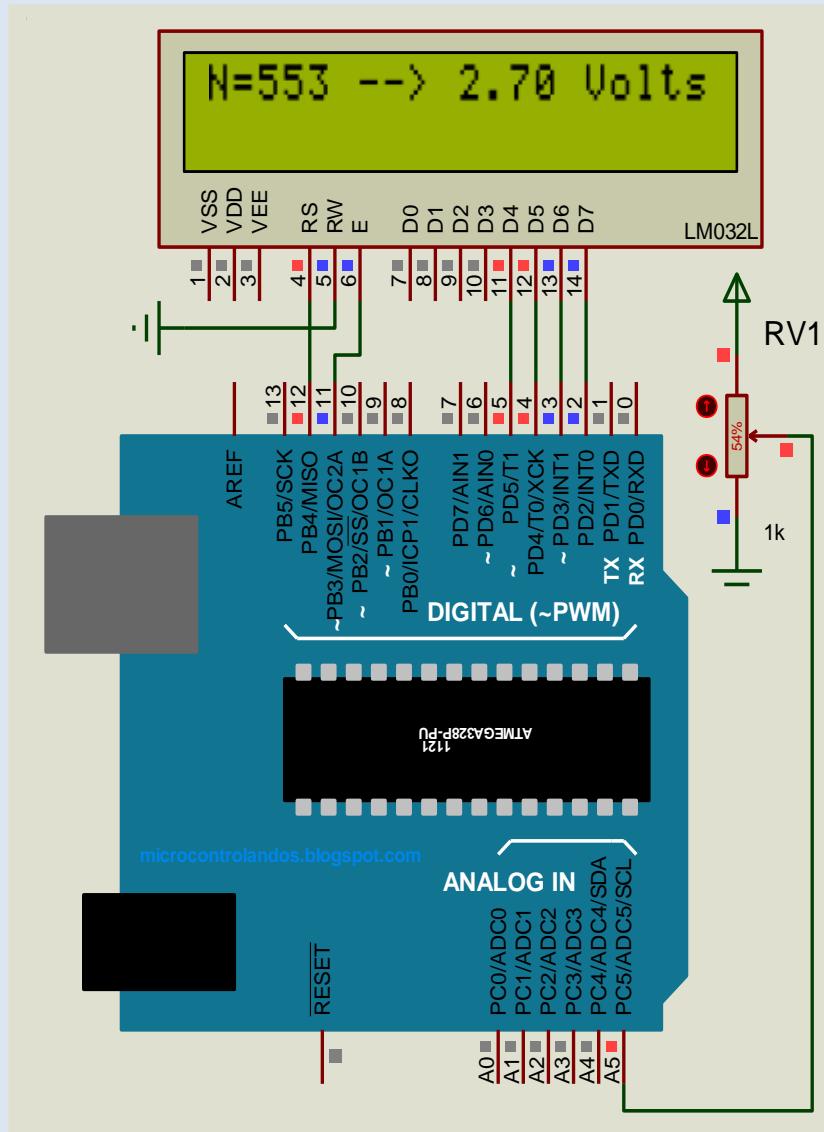
- **analogRead(Ai)** retourne un entier compris entre 0 et 1023 correspondant à la tension analogique sur l'entrée *Ai*. Le temps d'exécution de cette instruction est voisin de 100µs



- La tension analogique doit être comprise entre 0 et *Vref*. Par défaut, *Vref*=5V,
- La résolution = précision du convertisseur dépend de *Vref*:
$$\text{Res} = \text{Vref} / 1023$$

- On peut modifier *Vref* en utilisant l'Entrée *Vref* et la fonction `analogReference()`
 - `analogReference(DEFAULT);` -> 5V (3.3V)
 - `analogReference(EXTERNAL);` -> entrée *Vref*
 - `analogReference(INTERNAL);` 1.1V

Mesurer Entrée analogique: Exemple

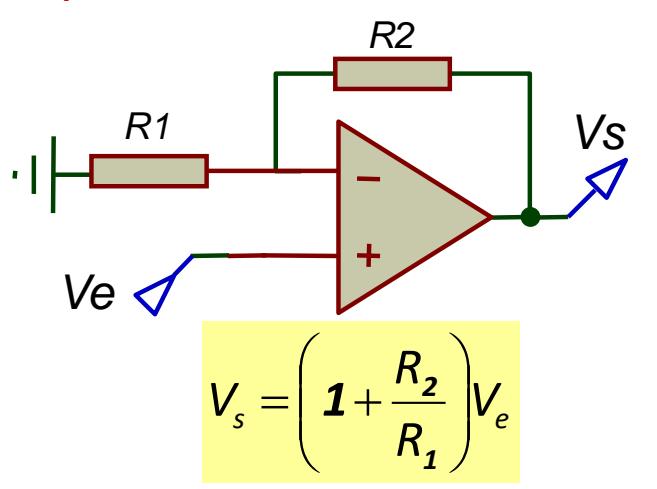


```
#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);
int N;
float v;
void setup() {
    lcd.begin(20, 2);
}

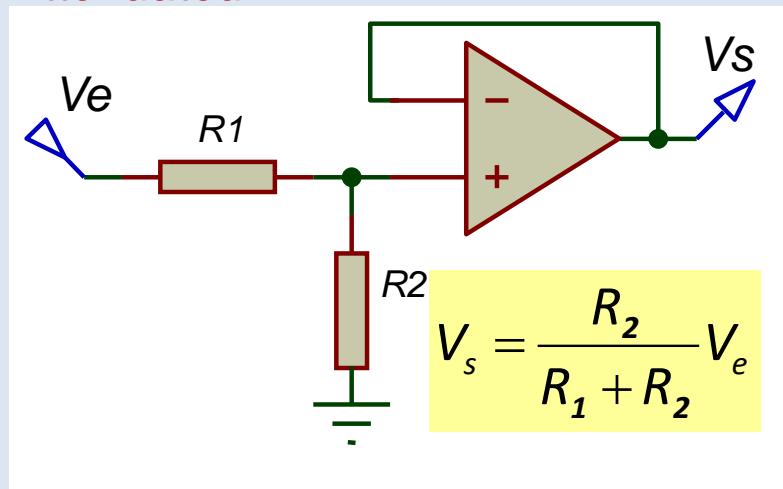
void loop() {
    N = analogRead(A5);
    v = N * 5.0 / 1023.0;
    lcd.clear();
    lcd.print("N=");
    lcd.print(N);
    lcd.print(" --> ");
    lcd.print(v);
    lcd.print(" Volts");
    delay(2000);
}
```

Mise en forme du signal analogique

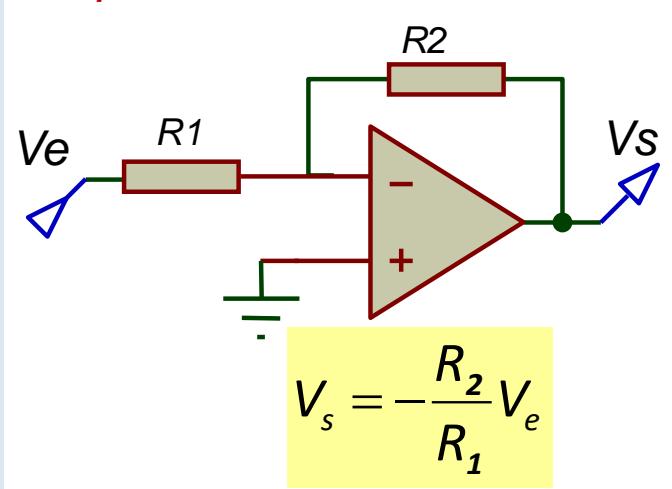
Amplificateur non inverseur



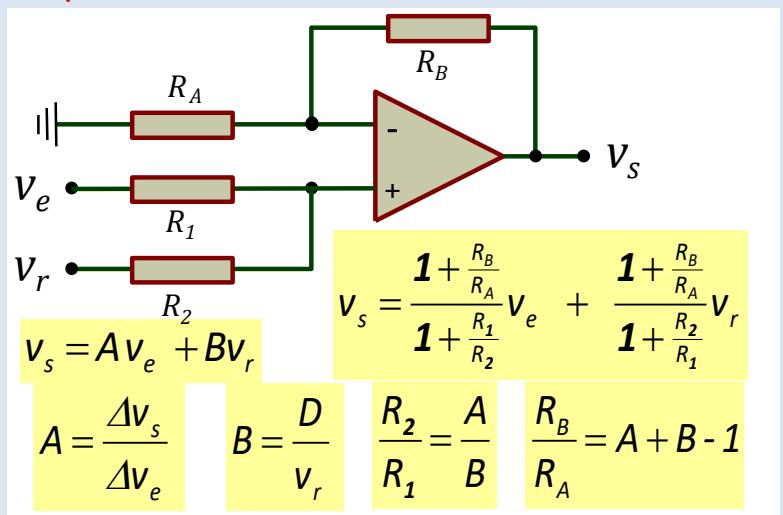
Atténuateur



Ampli/Atténuateur inverseur

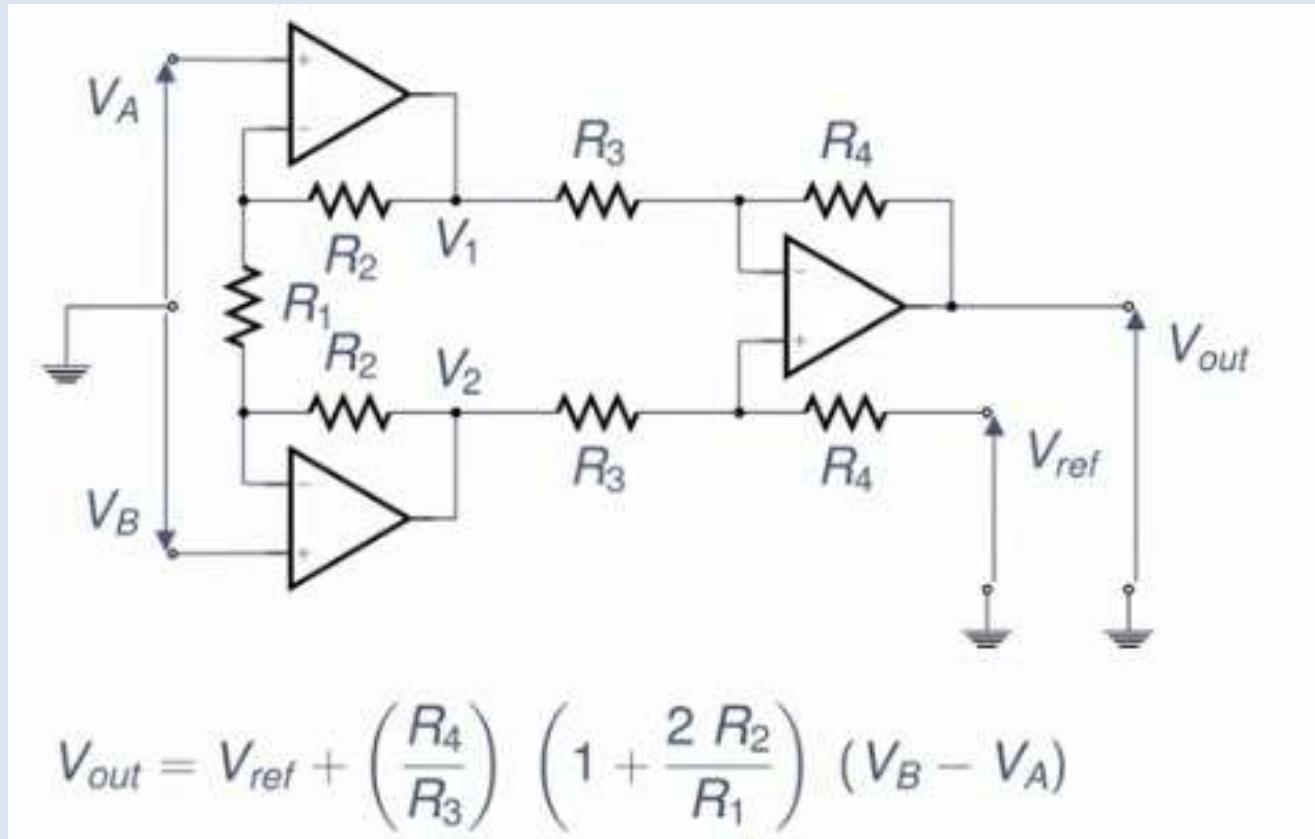


Ampli Décaleur



Ampli d'instrumentation

- A utiliser avec les capteurs qui délivrent un signal différentiel

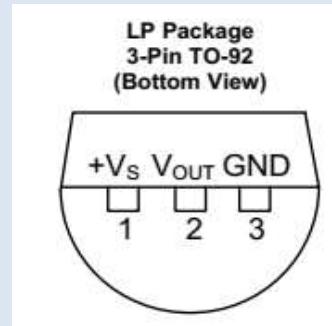


Le capteur de température LM35

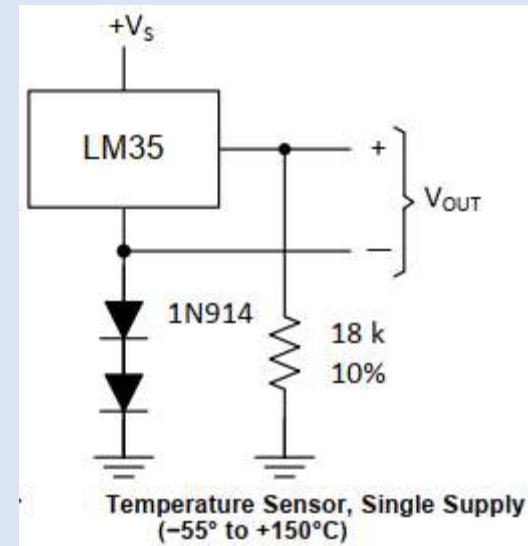
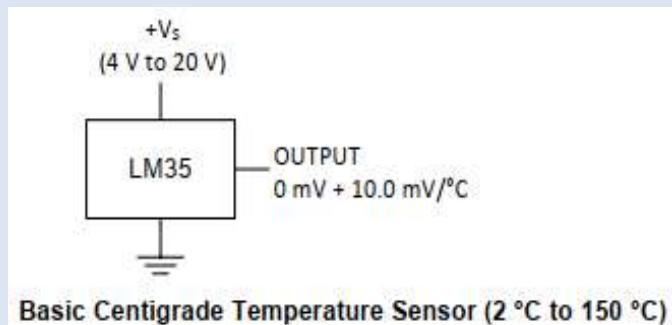
- Capteur analogique
- délivre tension proportionnelle à la température: $10\text{mV /}^{\circ}\text{C}$

$$T_{({}^{\circ}\text{C})} = \frac{V_{(\text{mV})}}{10} = V_{(\text{v})} \times \mathbf{100}$$

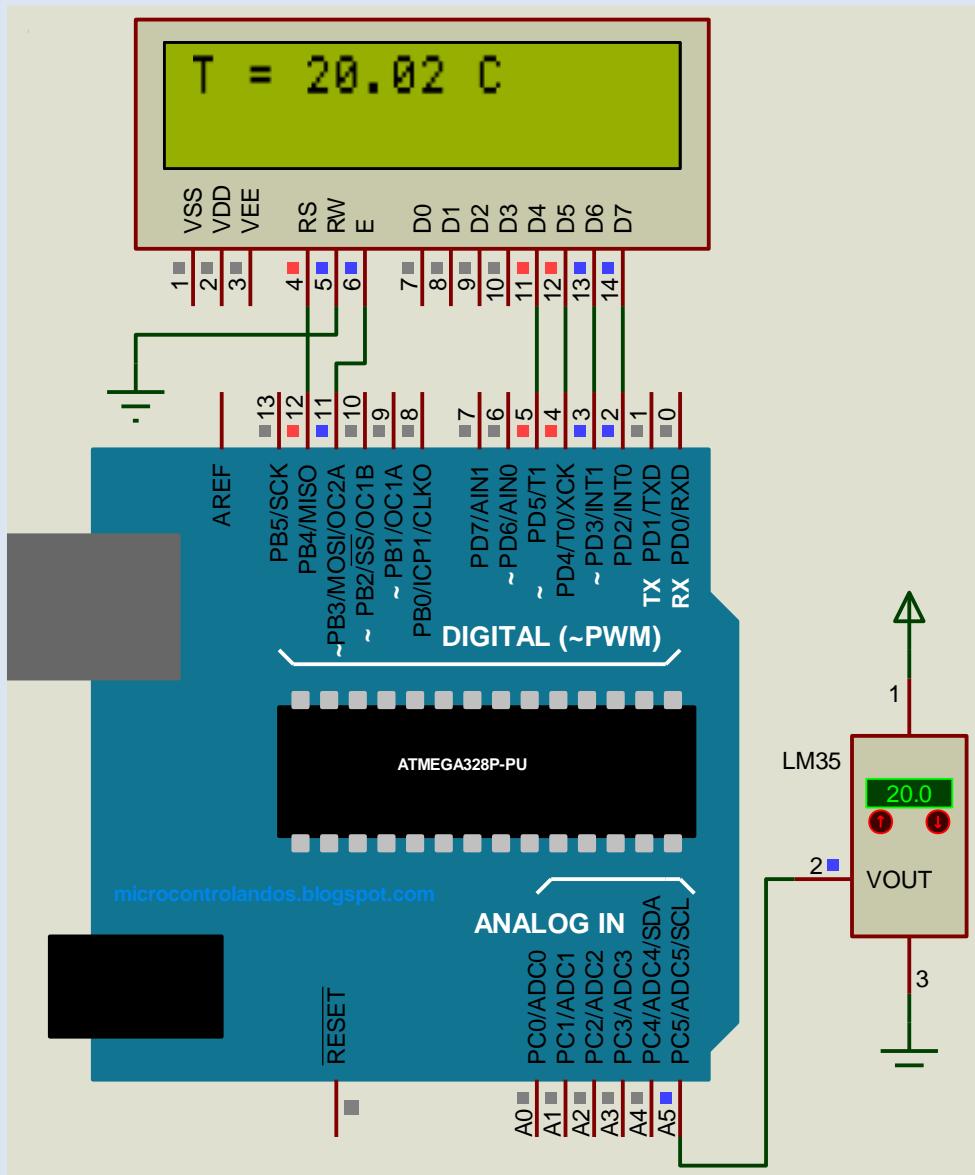
- Si on travaille dans la plage $[0, 100]^{\circ}\text{C}$, la valeur max de la tension délivrée par le capteur est 1V. Donc, il est conseillé d'utiliser un ADC avec une tension de référence réduite pour améliorer la précision. Sur Arduino, on peut utiliser $V_{ref\ interne} = 1.1\text{V}$



LM35, LM35A	-55 à 150°C
LM35Cxx	-40 à 110°C
LM35Dxx	0 à 100°C



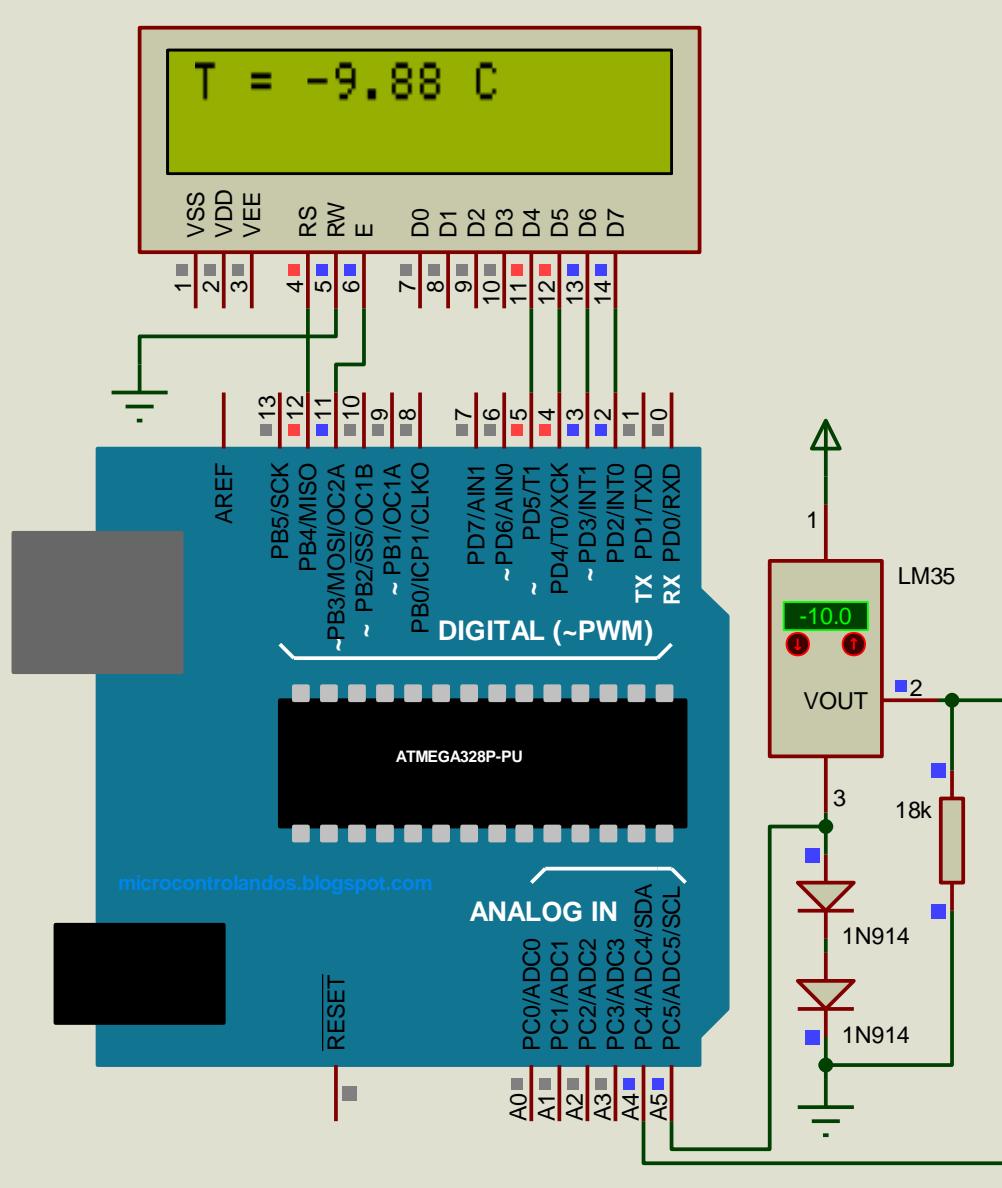
LM35 [0 à +100°C]



```
#include <LiquidCrystal.h>
LiquidCrystal lcd(12,11,5,4,3,2);
void setup() {
    analogReference(INTERNAL);
    lcd.begin(16,2);
}

void loop() {
    int N=analogRead(A5);
    float V = N * 1.1 / 1023.0; // volts
    float T = V *100 ; // °C
    lcd.clear();
    lcd.print("T = ");
    lcd.print(T);
    lcd.print(" C");
    delay(200);
}
```

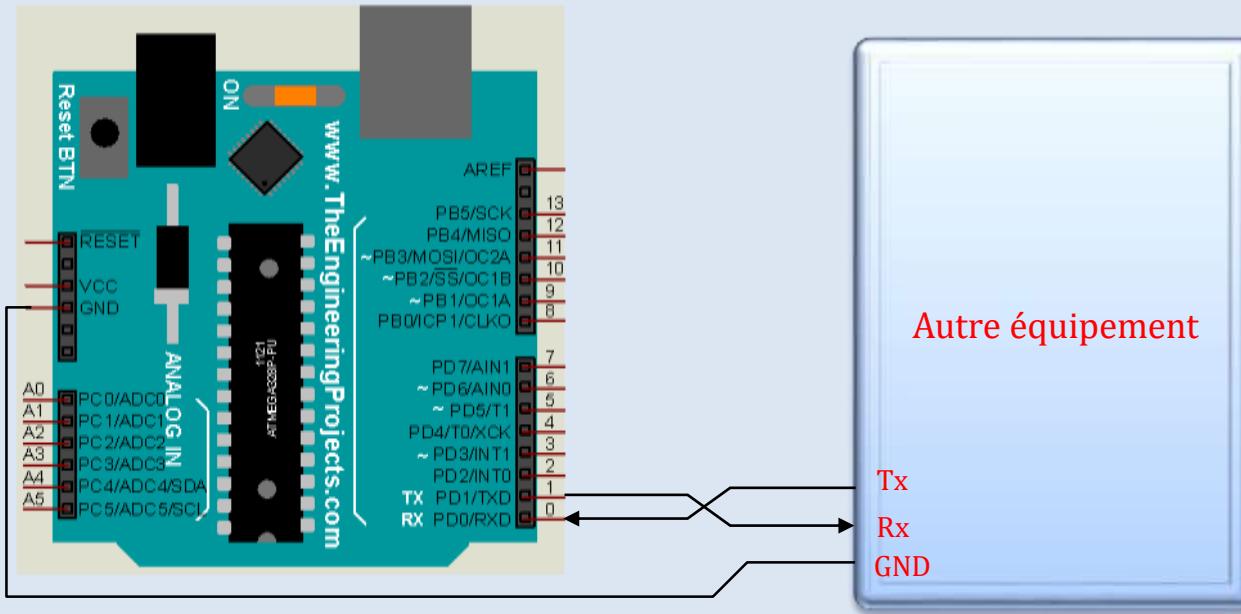
LM35 [-55 à +100°C]



```
#include <LiquidCrystal.h>
LiquidCrystal lcd(12,11,5,4,3,2);
void setup() {
    analogReference(INTERNAL);
    lcd.begin(16,2);
}

void loop() {
    int N=analogRead(A4)-analogRead(A5);
    float T = N*110.0/1023.0;
    lcd.clear();
    lcd.print("T = ");
    lcd.print(T);
    lcd.print(" C");
    delay(200);
}
```

Communication série



- Communication série Asynchrone bidirectionnelle (full duplex).
Transmission sur **Tx**, Réception sur **Rx**
- Les deux extrémités doivent avoir la même config: **8bits données**, 1 bit de stop, vitesse parmi 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, ou 115200 bauds,
- Les échanges sont empaquetés **octet** par **octet**
- Logique positive (0V/5V) ou (0V/3.3V)
- Aucun protocole de contrôle Hardware

● *Serial.begin(speed);*

- Initialise le port série et définit la vitesse de communication
- *Serial.begin(9600);*

● *Serial.write(data);*

- *Transmet un octet, une chaîne ou les octets d'un tableau de type char[] ou byte[]*
- *Serial.write('A');* transmet l'octet 65 = caractère A
- *Serial.write(66);* transmet l'octet 66 = caractère B
- *Serial.write("Bonjour");* transmet la chaîne Bonjour (octet par octet)
- *Serial.write(856);* le nombre 856 est tronqué à un octet puis transmis
- *byte B[] = {0x11, 0x22, 0x33, 0x44, 0x55}; Serial.write(B,3);* transmet 3 octets du tableau B

La librairie Serial

● ***Serial.print(data, [format]);***

- Transmet une donnée formatée. Le plus souvent vers un équipement d'affichage
- *Serial.print(2048);* Transmet la chaîne "2048" → 4 octets: '2' , '0' , '4' , '8' -> octets 50, 48, 52, 56
- *Serial.print(23456,HEX);* Transmet la chaîne "5BA0" qui est la représentation (papier) du nombre 23456 en hexadécimal (base 16) → octets 53, 66, 65 et 48 = codes ascii des caractères '5', 'B', 'A' et '0'
- *Serial.print(240,BIN);* Convertit le nombre 240 en binaire et le transmet en tant que chaîne "11110000". Un nombre négatif est représenté sur 32 bits quelque soit son type
- *Serial.print(235.4567);* Transmet le nombre réel 235.4567 en tant que chaîne en arrondissant à 2 chiffres décimaux "235.46"
- *Serial.print(235.4567,4);* Transmet en tant que chaîne avec 4 chiffres décimaux → "235.4567"
- *Serial.print("Bonjour ");* Transmet la chaîne "Bonjour"
- *Serial.print('X');* Transmet le caractère X

● **Serial.println(data);**

- Identique à Serial.print() mais transmet en plus un retour ligne (CR LF)=(13 10)
- *Serial.println(); // transmet un retour à la ligne seul*

● **n = Serial.available();**

- Retourne le nombre d'octets disponible dans le buffer de réception

● **b = Serial.read();**

- Lit un octet à partir du buffer de réception. L'octet est retiré du buffer. Si le buffer de réception est vide, la fonction retourne -1 (255).
- Le receveur **b** peut être de type *byte*, *char* ou *int*

La librairie Serial

○ ***n = Serial.readBytes(B, length);***

- Lit un ensemble d'octets à partir du buffer de réception
- **B**: nom de la variable qui reçoit les données. Type `char[]` ou `byte[]`
- **length**: nombre d'octets à lire (int)
- retourne **n** = nombre d'octets effectivement lus. Peut être inférieur à *length* en cas de timeout

○ ***n = Serial.readBytesUntil(terminator, B, length);***

- lit un ensemble d'octets et les place dans la variable B
- La réception s'arrête quand on reçoit l'octet *terminator* ou après la réception de *length* octets ou après un timeout.
- retourne le nombre **n** d'octets effectivement lus. Peut être inférieur à *length* en cas de timeout

La librairie Serial

● **S = Serial.readString();**

lit les caractères qui arrivent et les place dans la chaîne S (type string). S'arrête après un Timeout,

● **S = Serial.readStringUntil(C);**

lit les caractères qui arrivent et les place dans la chaîne S (type string). S'arrête à la réception du caractère C ou après un Timeout,

● **Serial.setTimeout(tms);**

Définit la durée de timeout en lecture. Par défaut le timeout est fixé à 1000 ms = 1s

● **Serial.end();**

Désactive l'UART. Les broches 0 et 1 peuvent de nouveau être utilisées comme des E/S normales

● **SerialEvent(){ }**

A chaque repassage dans la fonction loop(), si le buffer de réception contient quelque chose, cette fonction est exécutée.

Mode texte / mode binaire

- Soit le nombre entier $N = 50981 = B\textcolor{red}{11000111} \textcolor{blue}{00100101}$
- Transmettre le nombre N en **mode texte** consiste à transmettre 5 octets correspondant aux codes ASCII de chaque chiffre 53, 48, 57, 56, 49 \leftrightarrow ('5', '0', '9', '8', '1'). Ce mode est adapté à l'affichage. La fonction **Serial.print(N)** le fait automatiquement
- Transmettre le nombre N en **mode binaire** consiste à transmettre les 2 octets qui constituent le nombre. Ce mode est adapté à la transmission de données. La fonction la plus adaptée est **serial.write()**. Plusieurs méthodes sont possibles:
 - On peut considérer la variable N comme un tableau de bytes,
`Serial.write((byte*) &N , 2); // little Endian`
 - On peut utiliser les fonctions *LowByte()* et *highByte()* de la librairie Arduino
`int N;
N = ...;`
`Serial.write(LowByte(N));`
`Serial.write(highByte(N));`

Coller des octets pour former des nombres

- Il n'y a pas une seule façon de coller des **bytes** pour former un **int** ou un **float**

```
// Coller des bytes pour former des entiers
```

```
void setup() {  
    unsigned int N;  
    unsigned long K;  
    byte B[2] = { 0x11, 0x88};  
    byte C[4] = { 0x33, 0x55, 0x77 ,0x99};  
    memcpy(&N, B, 2); //N=0x8811=34833  
    memcpy(&K, C, 4); //K=0x99775533=2574734643  
// Pour un integer, on peut aussi faire  
    Serial.begin(9600);  
    byte L = Serial.read();  
    byte H = Serial.read();  
    N = (H<<8) | L; // logique  
//ou encore  
    N = H*256 + L; // arithmétique  
//ou encore  
    N = word(H,L);  
}  
void loop() {}
```

```
// Coller des bytes pour former un réel
```

```
void setup() {  
    float R;  
    byte B[4] = { 0x79 , 0xE9, 0xF6, 0x42};  
    memcpy(&R, B, 4);  
// on affiche tout pour vérifier  
    Serial.begin(9600);  
    Serial.println(B[3],HEX);  
    Serial.println(B[2],HEX);  
    Serial.println(B[1],HEX);  
    Serial.println(B[0],HEX);  
    Serial.println(R,3);  
}  
void loop() {}
```

10100000

0101101100000000

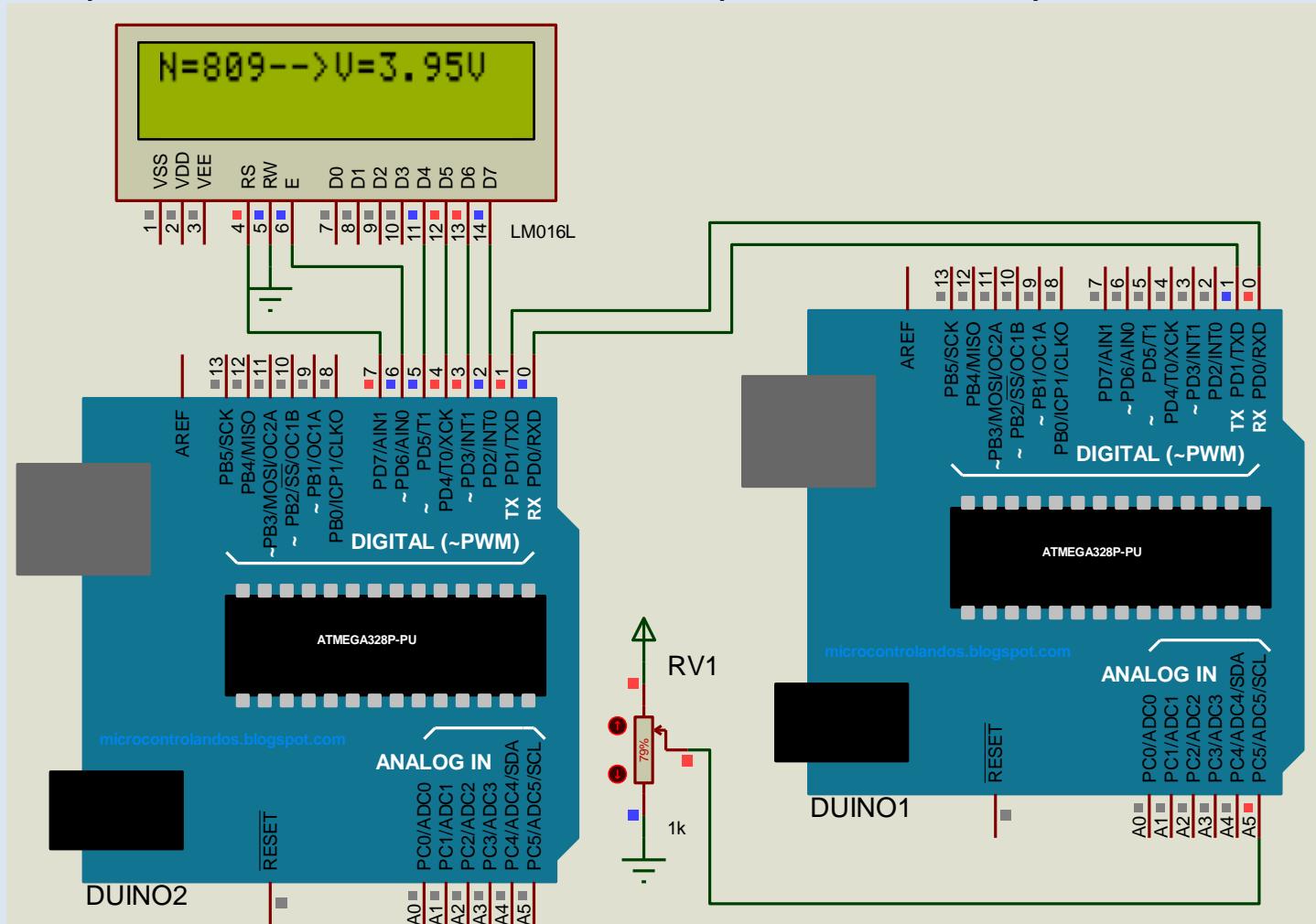
0101101110100000****

7=111

14=1110

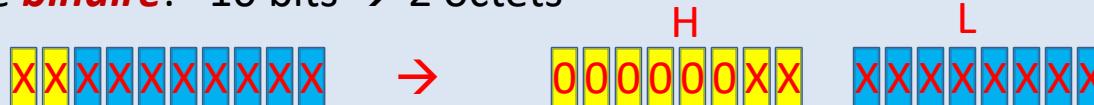
Exemple Communication Série

Arduino1 mesure l'entrée A5 et transmet le résultat vers Arduino2 qui fait l'affichage. Nous allons nous attarder sur cet exemple car il y a plusieurs aspects à considérer surtout le problème de synchronisation



Transmission en mode binaire

- ☐ `analogRead()` → mesure de tension à l'aide ADC 10 bits → entier 10 bits
- ☐ Port Série → Transmission octet par octet
- ☐ Mode **binnaire**: 10 bits → 2 octets



- ☐ Si adopte le mode *Big Endian* et on envoie des mesures d'une façon répétitive



- ☐ L'arduino qui reçoit commence à lire à un moment indépendant . On a une chance sur deux qu'il commence à lire à l'octet L. Les nombres reconstitués seront faux



- ☐ On décide d'ajouter un octet particulier au début de chaque mesure → **entête**



- ☐ Un paquet normal: #HL, mais comme L (8 bits) peut prendre la valeur # on peut tomber (très peu probable) sur ##HL. On peut écarter ce cas en faisant un test sur l'octet juste après le 1^{er} # car H ne peut prendre que les valeurs 0, 1, 2 ou 3

- ☐ Nous venons d'inventer notre propre protocole de communication. Ce n'est probablement pas le meilleur, mais il marche

Mode binaire Version 1

// réception et affichage

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(7,6,5,4,3,2);
void setup() {
    Serial.begin(9600);
    lcd.begin(16, 2);
}
void loop() {
    byte H,L;
    while(Serial.available() < 3);
    if( Serial.read() == '#'){
        H = Serial.read();
        L = Serial.read();
        uint16_t N = word(H,L);
        float V = N * 5.0 / 1023.0 ;
        lcd.clear();
        lcd.print("N=");
        lcd.print(N);
        lcd.print("-->");
        lcd.print("V=");
        lcd.print(V);
        lcd.print("V");
    }
}
```

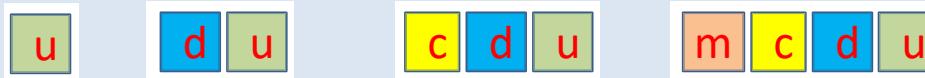
// Acquisition et transmission

```
void setup() {
    Serial.begin(9600);
}

void loop() {
    uint16_t N = analogRead(A5);
    Serial.write('#');
    Serial.write(highByte(N));
    Serial.write(lowByte(N));
    delay(100);
}
```

Essayons le mode texte

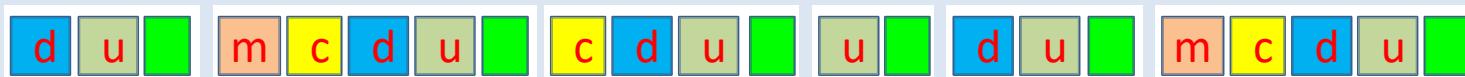
- ☐ **analogRead()** → mesure de tension à l'aide ADC 10 bits → entier 10 bits
- ☐ Port Série → Transmission octet par octet
- ☐ Mode **texte**: 10 bits → nombre compris entre 0 et 1023 → 1 à 4 chiffres



- ☐ Si on envoie (en mode texte) des mesures d'une façon répétitive en commençant par le chiffre de plus fort poids (*Serial.print*)



- ☐ L'arduino qui reçoit commence à lire à un moment indépendant . Il y a peu de chance qu'il commence au bon endroit. En plus, il ne sait pas combien de chiffre il doit lire
- ☐ On peut envisager plusieurs solutions. Je propose d'ajouter un séparateur en guise terminateur (=chiffre) pourquoi pas le caractère '**'space'** comme sur papier



- ☐ Pour récupérer le nombre sur la machine destination, on a plusieurs méthodes:
 - ☐ *On le place dans une chaîne en lisant tous les caractères jusqu'au prochain séparateur*
 - ☐ *On peut le lire et le transformer en mode binaire à l'aide de la fonction *Serial.parseInt();**
Avec cette méthode, on peut faire des opérations dessus

Mode texte

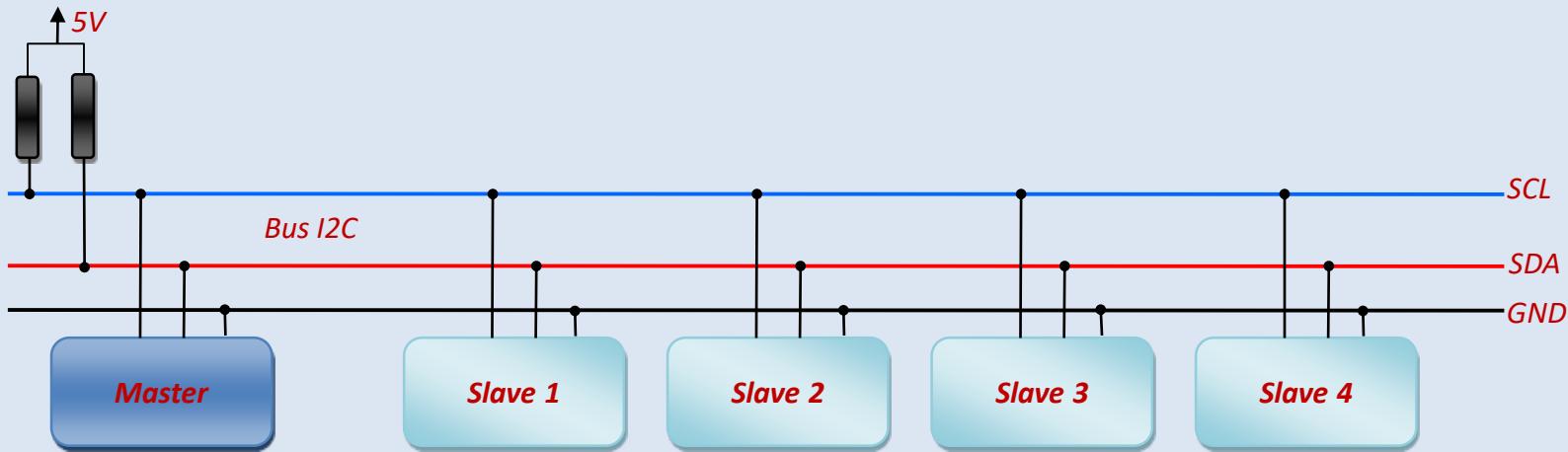
```
// réception et affichage  
#include <LiquidCrystal.h>  
LiquidCrystal lcd(7,6,5,4,3,2);  
void setup() {  
    Serial.begin(9600);  
    lcd.begin(16, 2);  
}  
  
void loop() {  
    int N = Serial.parseInt();  
    float V = (N * 5.0)/1023.0;  
    lcd.clear();  
    lcd.print(N);  
    lcd.setCursor(0,1);  
    lcd.print(V);  
}
```

```
// réception et affichage  
#include <LiquidCrystal.h>  
LiquidCrystal lcd(7,6,5,4,3,2);  
void setup() {  
    Serial.begin(9600);  
    lcd.begin(16, 2);  
}  
  
void loop() {  
    String S = Serial.readStringUntil(' ');  
    lcd.clear();  
    lcd.print(S);  
}
```

```
// Acquisition et  
transmission  
  
void setup() {  
    Serial.begin(9600);  
}
```

```
void loop() {  
    int N = analogRead(A5);  
    Serial.print(N);  
    Serial.write(' ');  
    delay(100);  
}
```

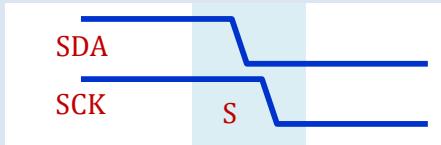
L'interface de communication I2C



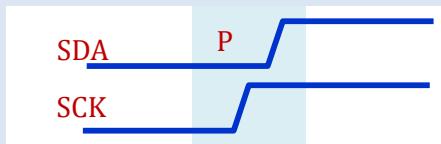
- BUS de communication série Synchrone.
- Un fil de données SDA (Serial DAta) et un fil d'horloge SCL (Serial CLock). Les deux fils doivent être tirés au 5V à l'aide de résistances de pull-up ($\approx 2.2k$ à $4.7k$)
- Les échanges de bits se font aux rythme de l'horloge qui est générée par le master. Les autres circuits sont des slaves, chacun est identifié par une adresse unique.
- Dans la norme de base, l'adresse est codée sur 7 bits, ce qui signifie que l'on peut brancher jusqu'à 128 slave sur le même bus
- Un bus peut avoir plusieurs masters, mais dès que l'un prend le contrôle du bus, les autres doivent attendre jusqu'à ce que le bus se libère
- Le standard I2C supporte 5 vitesses de communication: 100kHz, 400kHz, 1MHz, 3.4Mhz et 5MHz

Les signaux I2C

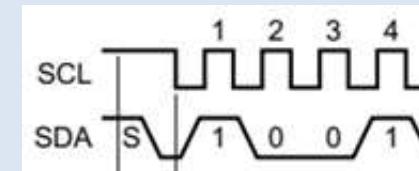
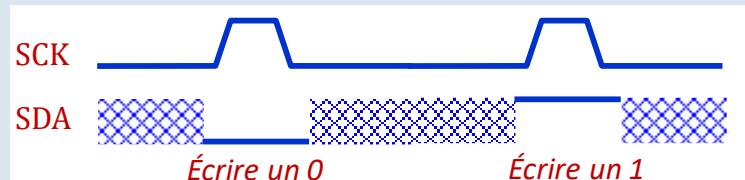
- **Bus libre**: Quand le bus est libre, les deux lignes SDA et SCL sont au niveau haut
- **Start Condition**: Un master démarre une séquence de communication en générant un Start condition



- **stop Condition**: Un master termine une séquence de communication en générant un stop condition



- **Ecrire un bit sur le bus**: Le circuit qui écrit force la ligne SDA à 0 ou à 1 pendant l'impulsion de l'horloge (c'est toujours le master qui génère l'horloge)



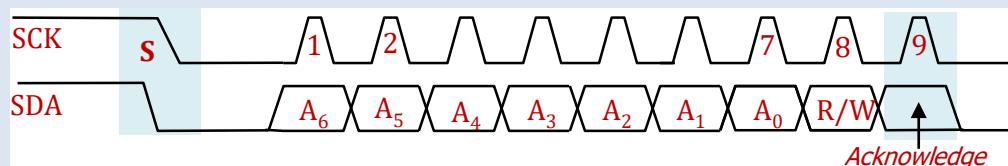
Les signaux I2C (2)

● **L'acknowledge**: L'acknowledge est l'accusé de réception. Il est placé par le circuit qui vient de recevoir un octet sur la ligne SDA pendant le 9ème coup d'horloge.

- SDA=0 → acknowledge positif (ACK),
- SDA=1 → acknowledge négatif (NoACK),

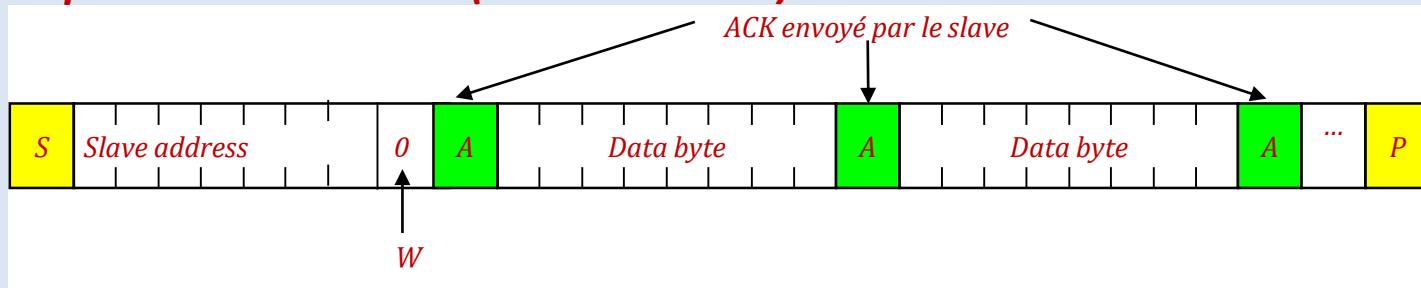
● **Séquence de communication**: Une séquence de communication doit toujours commencer comme suit:

- Le master envoie un Start,
- Le master envoie l'adresse du Slave sur 7 bits
- Le master envoie le bit R/W pour indiquer s'il veut envoyer des données vers le slave ou s'il veut en recevoir
 - R/W = 0 => il veut envoyer
 - R/W = 1 => il veut recevoir
- Le slave place l'acknowledge pendant le 9ème coup d'horloge

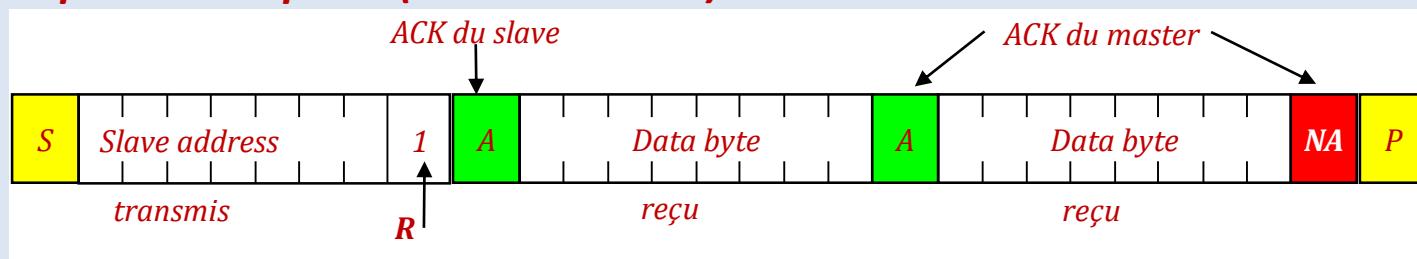


Séquences de communication

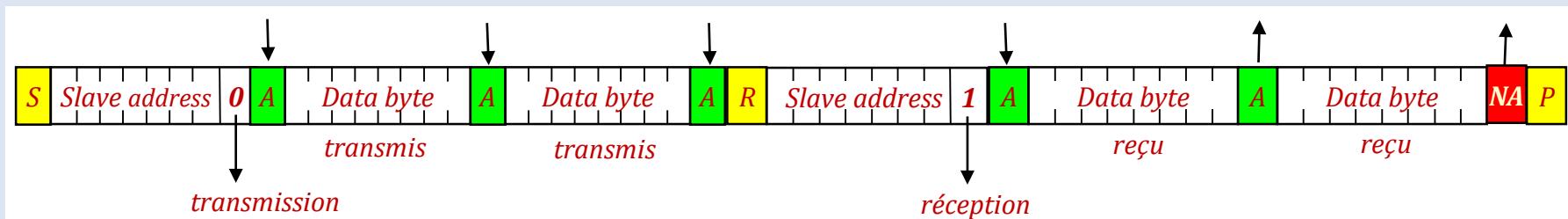
Séquence transmission (mater → slave)



Séquence réception (mater ← slave)



Séquence transmission réception



La librairie Wire

- **Wire.begin():** Initialise l'interface I2C. La vitesse est fixée à 100kHz.
- **Wire.setClock(Fhz):** Permet de modifier la fréquence de communication. Pour Arduino UNO, les valeurs acceptées sont 100000 et 400000
- **Wire.beginTransmission(SlaveAdr):** Envoie le *Start* suivi de l'adresse *SlaveAdr* suivi du bit *R/W* = 0
- **Wire.write(value);**
Wire.write(string);
Wire.write(tab, length); Envoie des octets de données.
- En fait, l'adresse et les données sont placés dans une file d'attente (*transmit buffer*) et ne sont envoyés sur le bus I2C que quand on exécute l'instruction
Wire.endTransmission();

La librairie Wire (2)

● **S=Wire.endTransmission([b]):** Déclenche la transmission de la file d'attente puis envoie un *Stop* ou un *RepeatStart* selon la valeur du paramètre logique b:

- Si b = true (valeur par défaut) => envoie un Stop pour libérer le bus I2C
- Si b = false => envoie un RepeatStart pour garder le contrôle du bus I2C

Cette fonction retourne un status S:

- 0: *success*
- 1: *data too long to fit in transmit buffer*
- 2: *received NoACK on transmit of address*
- 3: *received NoACK on transmit of data*
- 4: *other error*

La librairie Wire (3)

● ***k = Wire.requestFrom(adr, N, [b]):***

Réalise une séquence de réception complète:

Envoie le Start, l'adresse *adr* avec R/W=1, récupère *N* octets, et clôture la séquence selon la valeur du paramètre logique *b* (optionnel):

- Si *b=true* (défaut) la séquence est clôturée par un *stop*
- Si *b=false* la séquence est clôturée par un *RepeatStart*

Les octets récupérés sont placés dans le buffer de réception de l'interface I2C

La fonction retourne le nombre *k* d'octets effectivement lus qui peut être inférieur à *N*

La librairie Wire (4)

- ***n = Wire.available();***

Retourne le nombre d'octets en attente dans le buffer de réception

- ***B = Wire.read();***

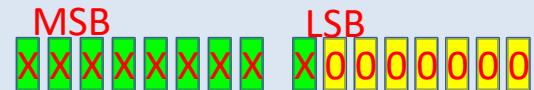
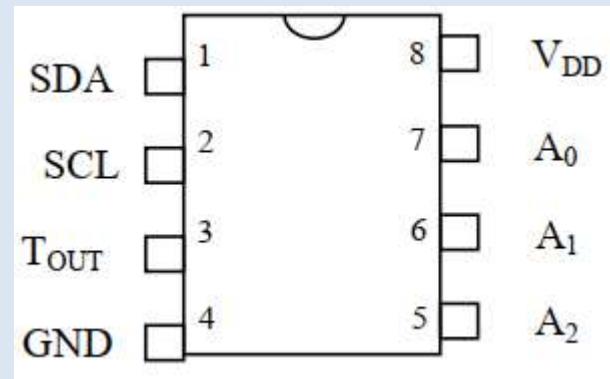
Permet de retirer un octet du buffer de réception

- ***n=Wire.readBytes(tab,N);***

Permet de retirer ***N*** octets du buffer de réception et les placer dans le tableau ***tab***. Retourne le nombre ***n*** d'octets effectivement lus qui peut être inférieur à ***N*** si le buffer de réception ne contient pas suffisamment d'octets

Le capteur de température DS1621

- Capteur Numérique. Il a son propre convertisseur analogique numérique. Il communique avec l'extérieur sur un bus I2C
- Possède 3 broches $A_2 A_1 A_0$ permettant de fixer son adresse sur le bus I2C. Les 4 bits de plus fort poids sont fixé en interne à 1001, l'adresse du circuit est donc **1001 A₂A₁A₀**. Ainsi avec 3 bits d'adresse utilisables, on peut brancher jusqu'à 8 capteurs sur le même bus
- Plage de température: -55°C à +125°C avec un incrément de 0.5°C
- La température est codée sur 9 bits qui sont placés dans le registre de température 16bits.
= 2 octets MSB et LSB
- Le MSB contient la valeur entière (signée) de la température. Le 8ème bit du LSB indique s'il faut rajouter 0.5 ou non. Le tableau ci-dessous donne quelques exemples



TEMPERATURE	DIGITAL OUTPUT
+125°C	MSB: 011101, LSB: 00000000
+25°C	MSB: 00011001, LSB: 00000000
+½°C	MSB: 00000000, LSB: 10000000
+0°C	MSB: 00000000, LSB: 00000000
-½°C	MSB: 11110000, LSB: 00000000
-25°C	MSB: 11100011, LSB: 00000000
-55°C	MSB: 11001001, LSB: 00000000

Commandes du DS1621

Le circuit a 2 modes de fonctionnement:

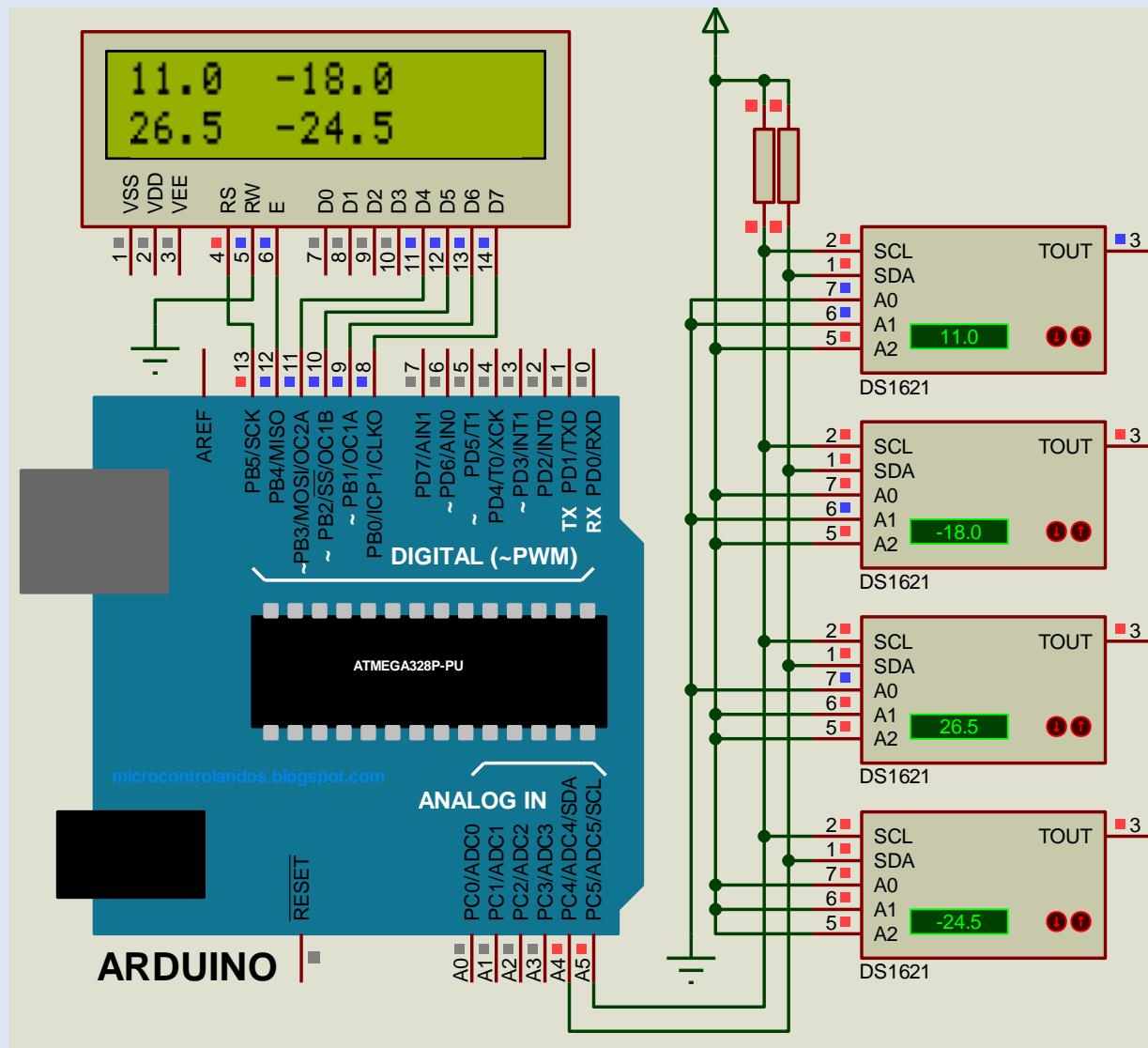
- **mode continu**: dès qu'on lui envoie la commande *Start Convert* (0xEE), il commence à mesurer la température d'une façon continue, chaque nouvelle mesure écrase l'ancienne dans le registre de température,
- **mode single shot**: si on lui envoie la commande *Start Convert* (0xEE), il réalise une seule mesure.
- Pour basculer d'un mode à l'autre il faut utiliser la commande 0xAC pour changer le bit m du registre de configuration 0000000m
- Par défaut, le circuit est en mode continu. Au démarrage, il est en mode standby. Si on lui envoie la commande 0xEE, il commence la conversion continue
- Si on lui envoie la commande *Read Temperature* (0xAA), le circuit nous envoie les deux octets de température en commençant par le MSB

TEMPERATURE CONVERSION COMMANDS		
Read Temperature	Read last converted temperature value from temperature register.	AAh
Read Counter	Reads value of Count_Remain	A8h
Read Slope	Reads value of the Count_Per_C	A9h
Start Convert T	Initiates temperature conversion.	EEh
Stop Convert T	Halts temperature conversion.	22h
THERMOSTAT COMMANDS		
Access TH	Reads or writes high temperature limit value into TH register.	A1h
Access TL	Reads or writes low temperature limit value into TL register.	A2h
Access Config	Reads or writes configuration data to configuration register.	ACh

Exemple: 4 x DS1621 (schéma)

- 4 capteurs DS1621 adresses:
 - 4 → 1001 **100=0x4C**
 - 5 → 1001 **101=0x4D**
 - 6 → 1001 **110=0x4E**
 - 7 → 1001 **111=0x4F**

- Affichage sur LCD



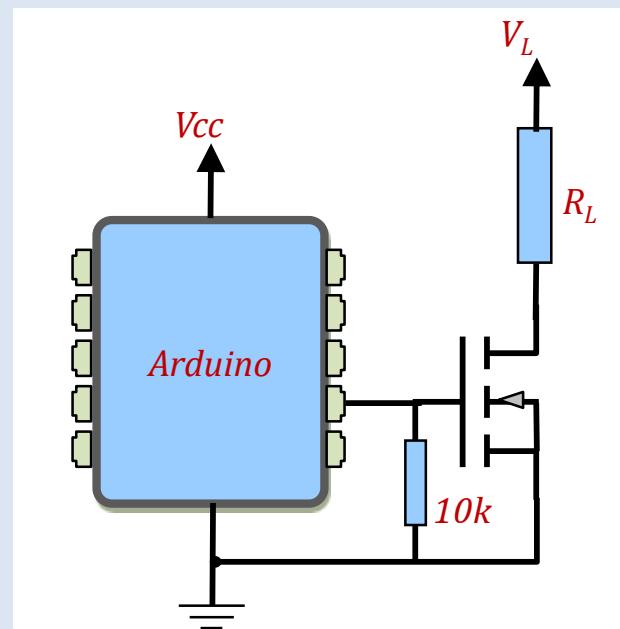
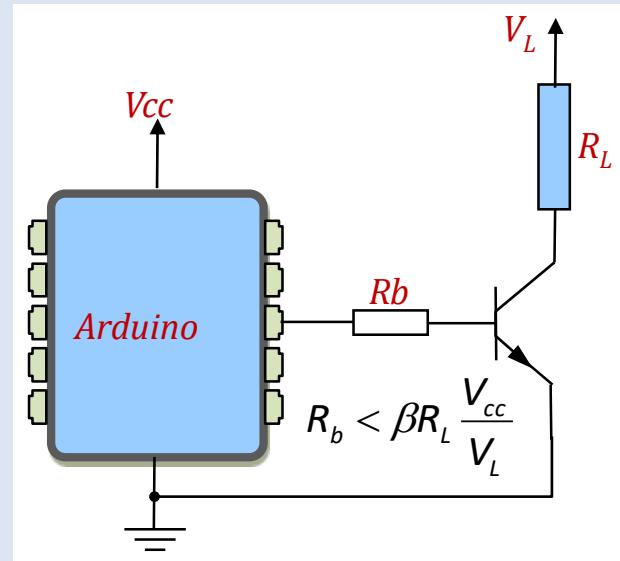
Exemple: 4 x DS1621 (code)

```
#include <Wire.h>
#include <LiquidCrystal.h>
int8_t msb; // signé
uint8_t lsb;
float T;
byte devadr[4] = {0x4C, 0x4D, 0x4E, 0x4F};
LiquidCrystal lcd(13,12,11,10,9,8);
void setup(){
    Wire.begin();
    lcd.begin(16, 2);
    for (int i = 0; i < 4; i++) {
        Wire.beginTransmission(devadr[i]);
        Wire.write(0xEE);
        Wire.endTransmission();
    }
    delay(750); //premier échantillon
}
```

```
void loop(){
    for (int i = 0; i < 4; i++) {
        Wire.beginTransmission(devadr[i]);
        Wire.write(0xAA);
        Wire.endTransmission();
        Wire.requestFrom(devadr[i], 2);
        msb = Wire.read();
        lsb = Wire.read();
        T = msb;
        if (lsb == 128)T = T + 0.5;
        if(i == 0)lcd.clear();
        if(i == 2)lcd.setCursor(0,1);
        lcd.print(T,1);
        lcd.print(" ");
    }
    delay(1000); // période de répétition
}
```

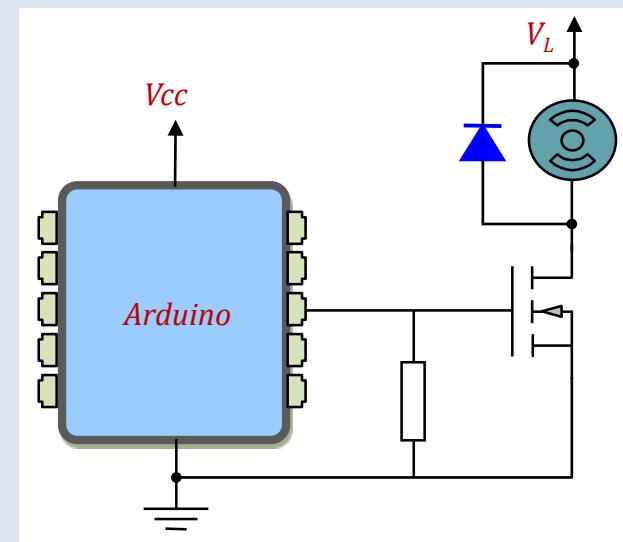
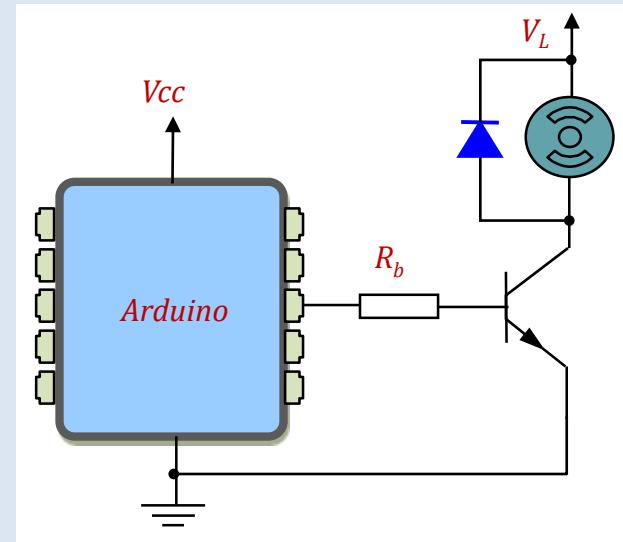
Commande d'une charge CC de puissance moyenne

- Une sortie Arduino ne peut pas contrôler directement une charge ($>5V$) ou ($> 200mA$)
- On peut utiliser un transistor bipolaire comme interrupteur
 - 2N2222 → 800 mA
 - TIP120, BDX53 → 8A
- On peut aussi utiliser un transistor MOS
 - 30N06L → 30A , 0.035Ω
 - IRF520 → 9A , 0.27Ω



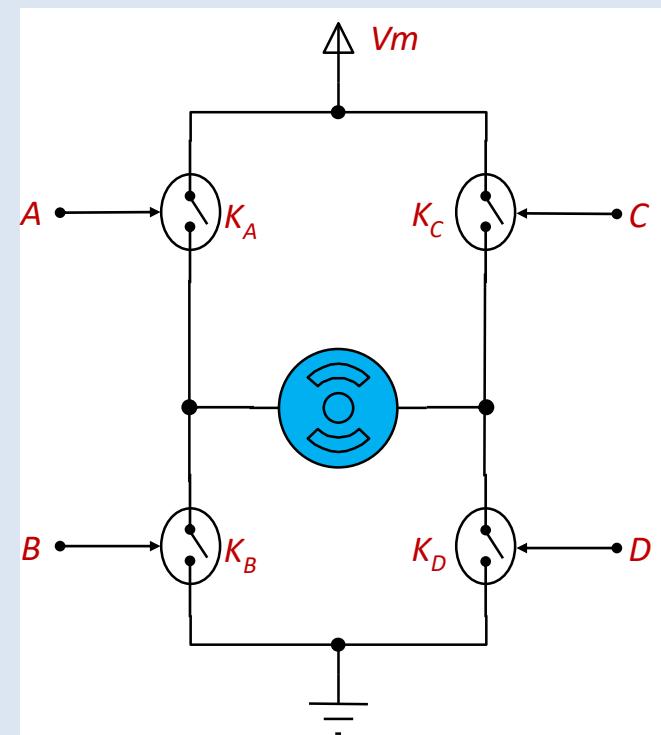
Cas d'un moteur CC

- C'est une charge inductive, Il ne faut pas oublier la diode récupération, sinon le transistor sera détruit
- Le moteur ne peut tourner que dans un seul sens
- Pour contrôler la vitesse, il faut utiliser une commande PWM



Commande d'un moteur à l'aide d'un pont en H

A	B	C	D	Moteur
0	0	0	0	Roue libre
1	0	0	1	gauche
0	1	1	0	droite
1	1	x	x	interdit
x	x	1	1	interdit
1	0	1	0	frein
0	1	0	1	frein
1	0	0	PWM	Gauche variable
0	PWM	1	0	Droite variable

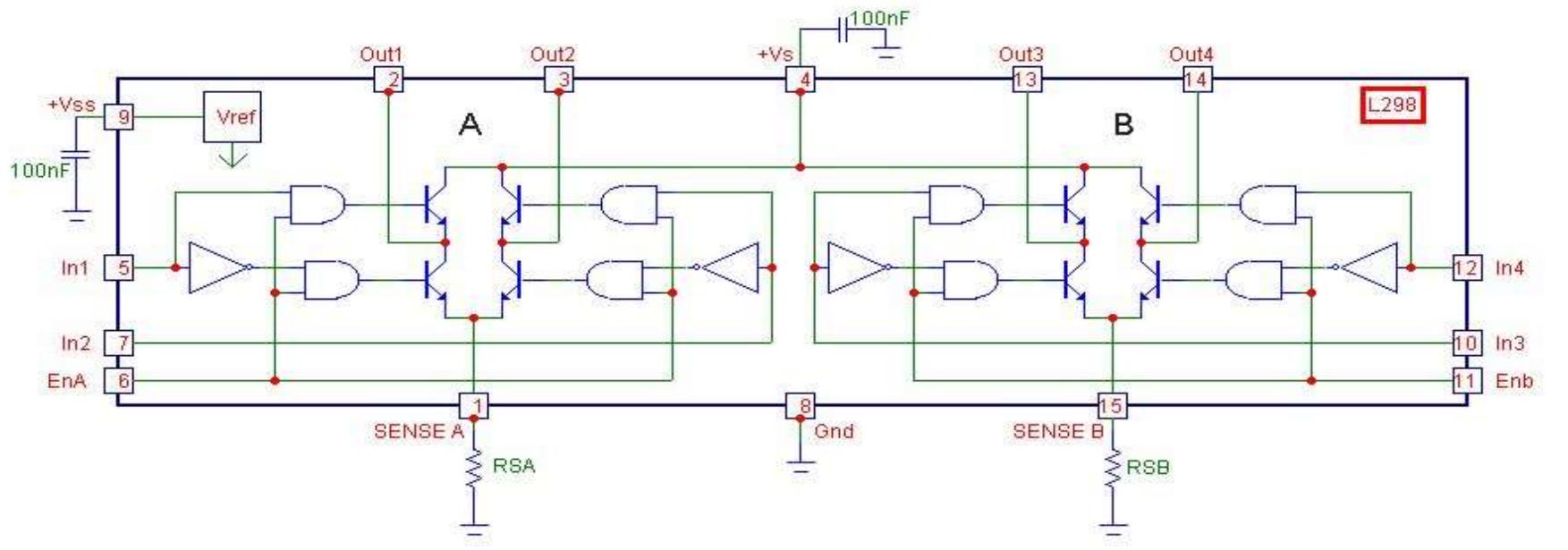


Le pont en H intégré L298

- Le L298 contient **deux** ponts en H avec trois entrées de commande pour chaque pont
- Intensité maximale : 2A par pont ;
- Alimentation Logique: $V_{ss}=5V$
- Alimentation Moteur: de 5.5V à 50V ;
- Dissipation puissance total : 25w ;



Le pont en H intégré L298 (2)

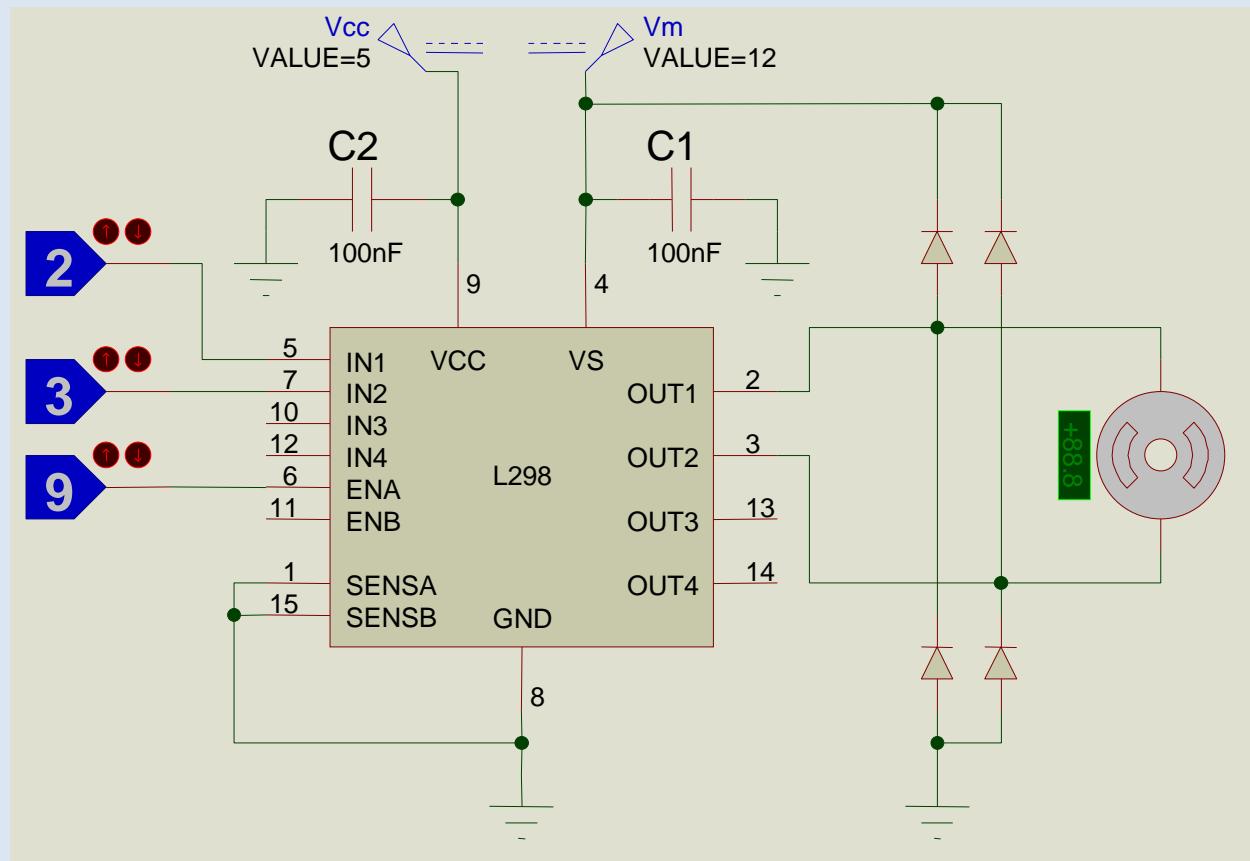


Entrées		Fonction
ENA = 1 ou ENA=PWM	In1=1 ; In2=0	marche avant (droite)
	In1=0 ; In2=1	marche arrière (gauche)
ENA=PWM	In1=In2	frein
ENA = 0	In1=X ; In2=X	Roue libre

Exemple Arduino - L298

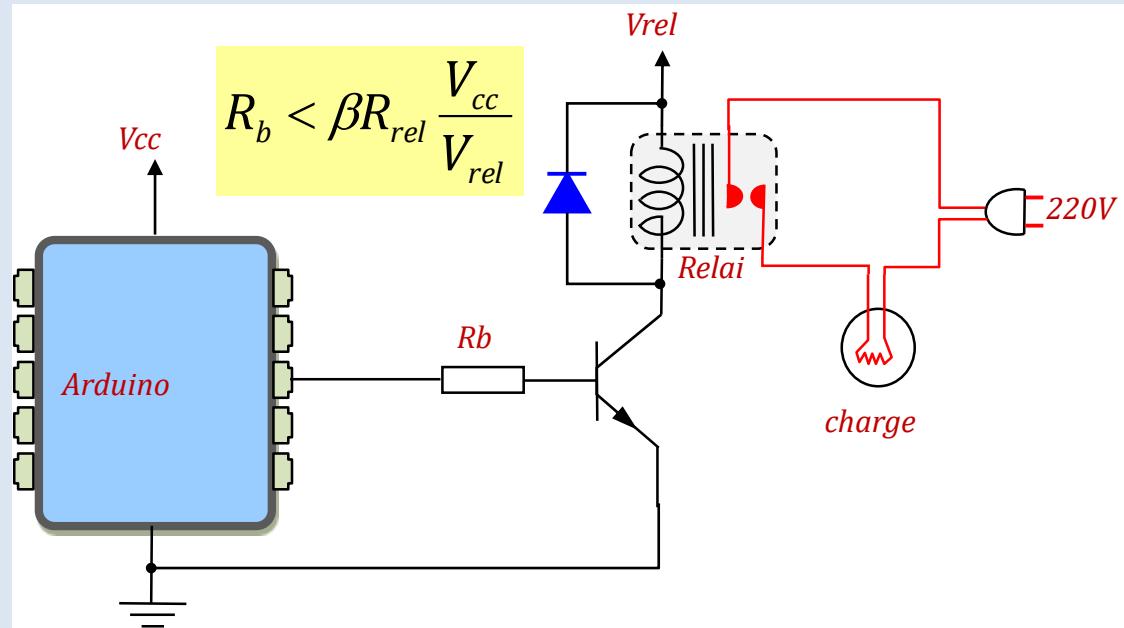
- Arduino branché à un terminal sur le port série et contrôle le L298 par les pates 2, 3 et 9.
 - On écoute le port série. Si on reçoit:

- ❑ G → gauche
 - ❑ D → droite
 - ❑ S → Arrêt
 - ❑ L → Roue libre
 - ❑ + → Accélérer
 - ❑ - → Ralentir



Au premier démarrage, on commence avec un rapport pwm de $\frac{1}{2}$

Commande à l'aide d'un relai

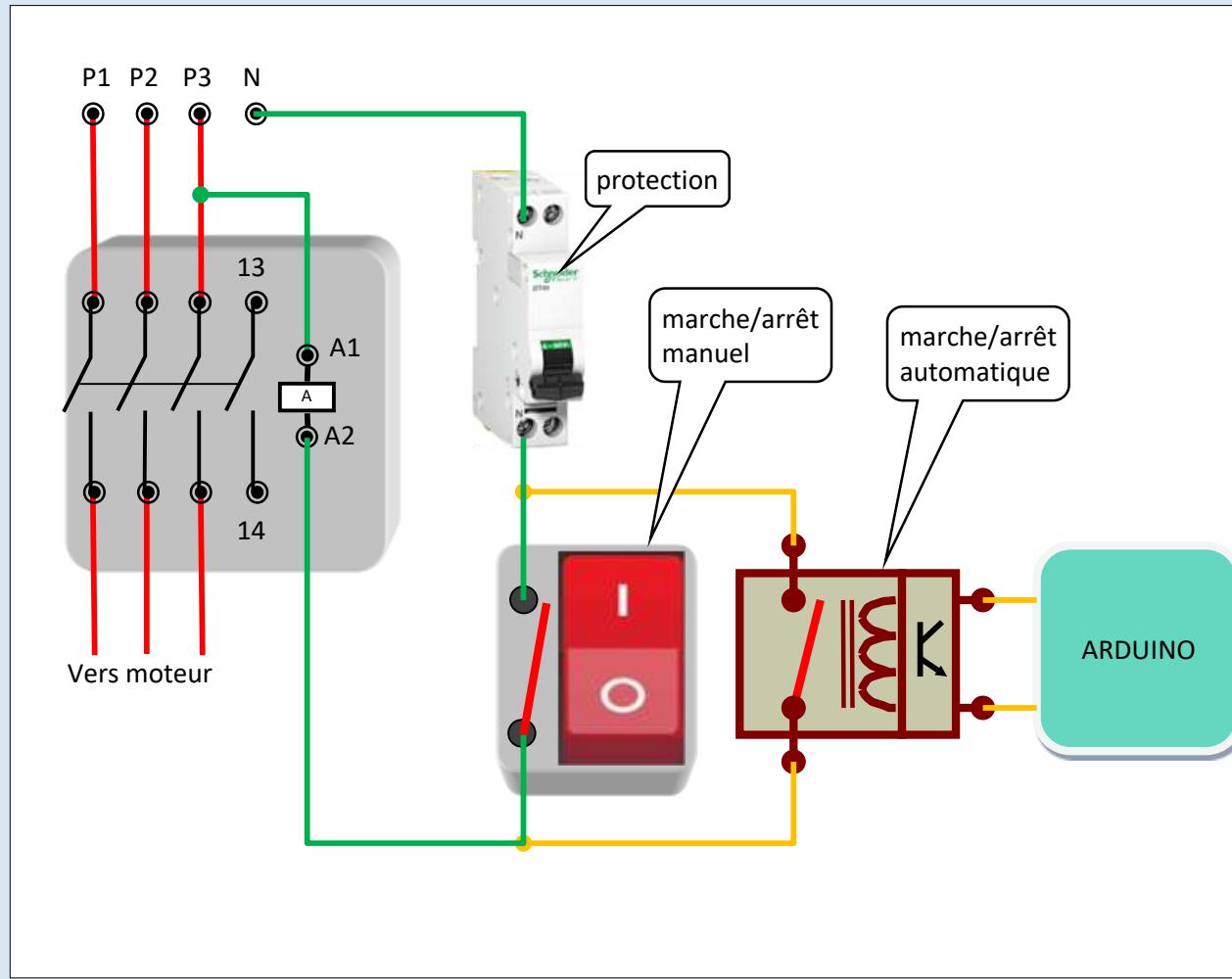


- Isolation galvanique entre le circuit de commande et le circuit de puissance. On peut commander des charges alimentées par le secteur
- Un petit transistor bipolaire fait l'affaire car il alimente juste la bobine du relai. On peut utiliser un transistor MOS ou un coupleur optique

Commande par interrupteur marche/arrêt

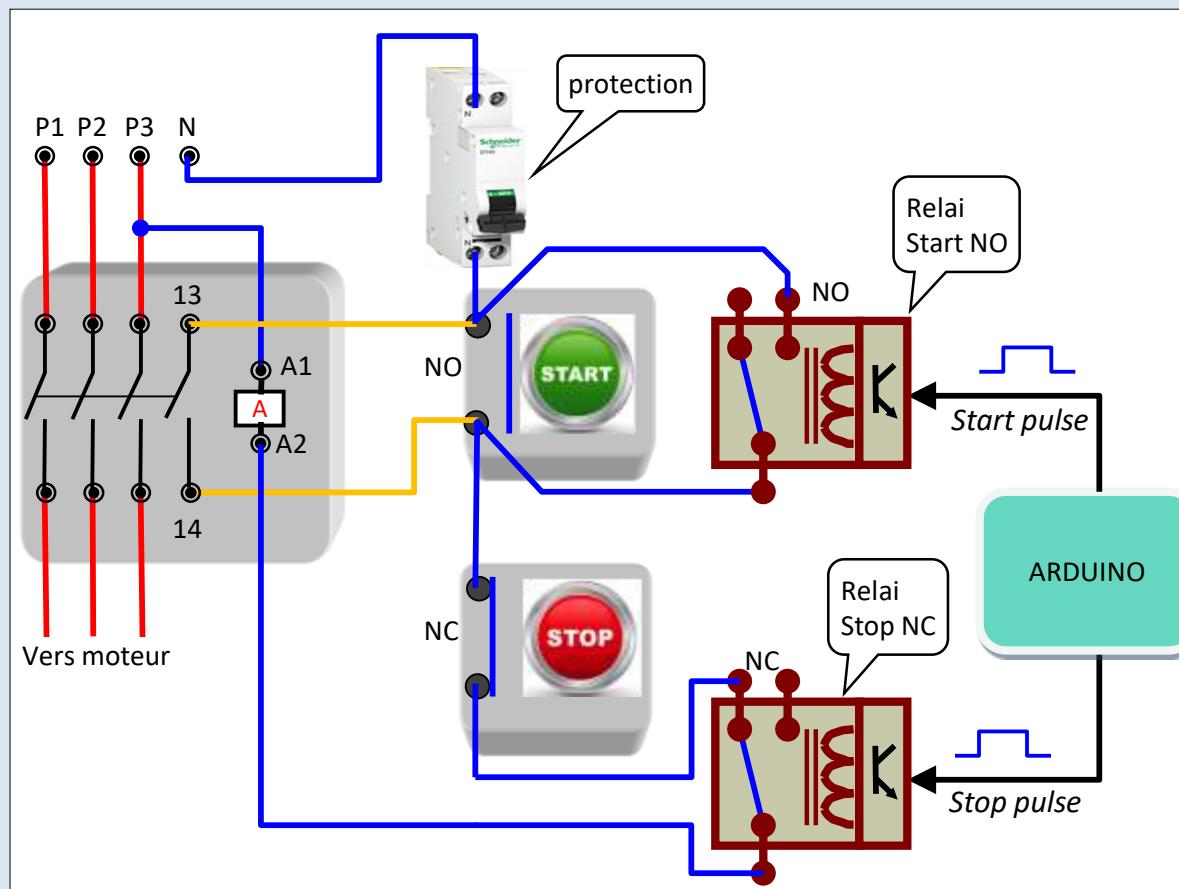
Arduino doit envoyer une commande par niveau

Haut → Relais fermé → marche , Bas → Relai ouvert → Arrêt

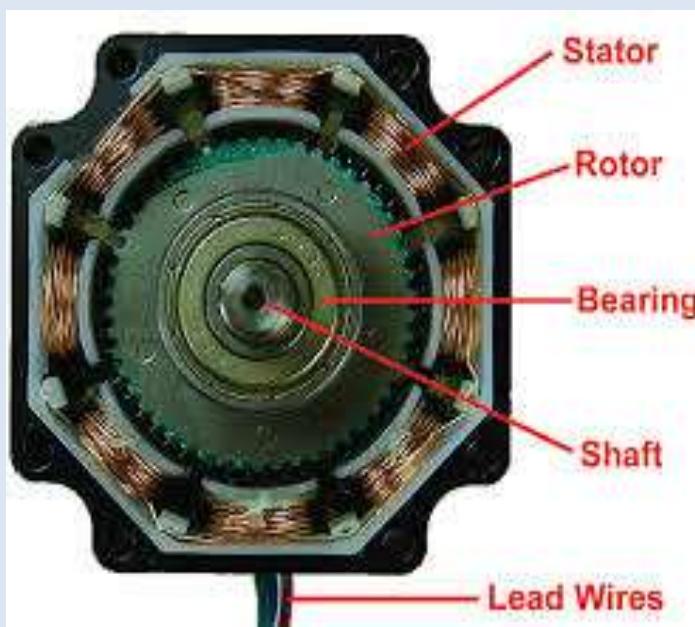
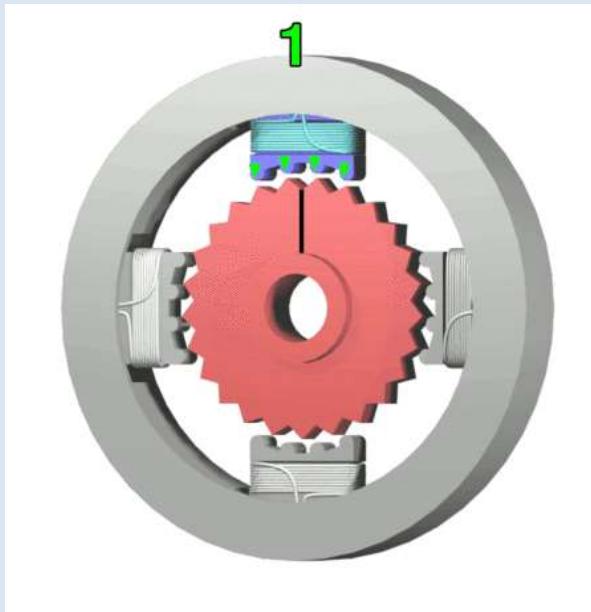
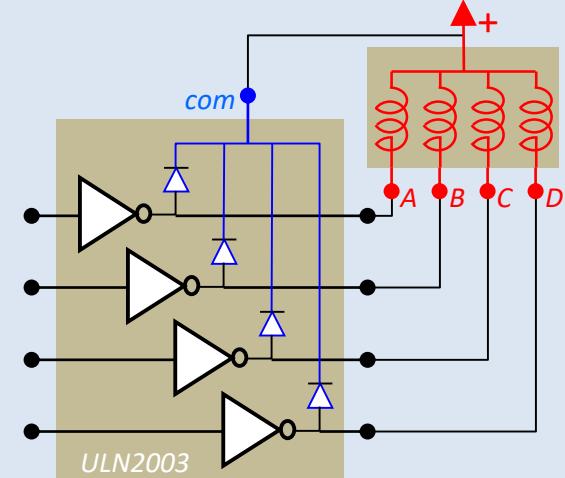
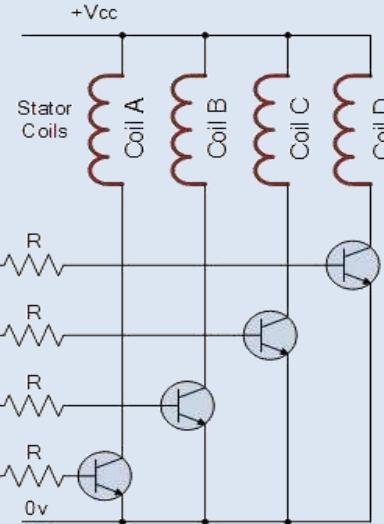
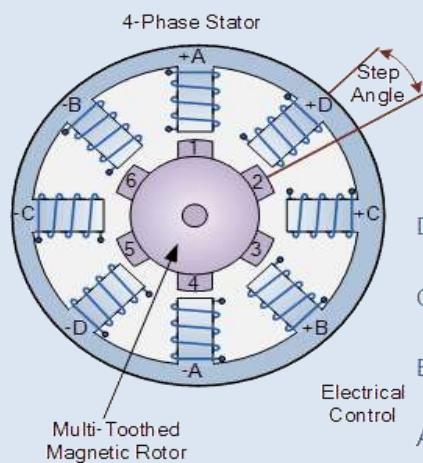
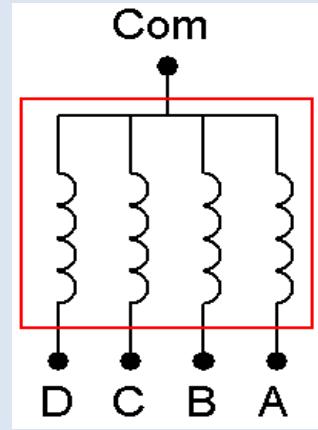


Commande par bouton Start et bouton Stop

- Arduino doit envoyer une commande sous forme d'impulsion
- Impulsion Start → marche, le contact de maintien (13,14) maintient l'alimentation de la bobine A du contacteur
- Impulsion Stop → Ouverture du circuit de commande, le contacteur lâche,



Commande d'un moteur pas à pas



D	C	B	A
0	0	0	1
0	0	1	0
0	1	0	0
1	0	0	0

$$N_{pas} = N_{droitor} \times N_{phases}$$

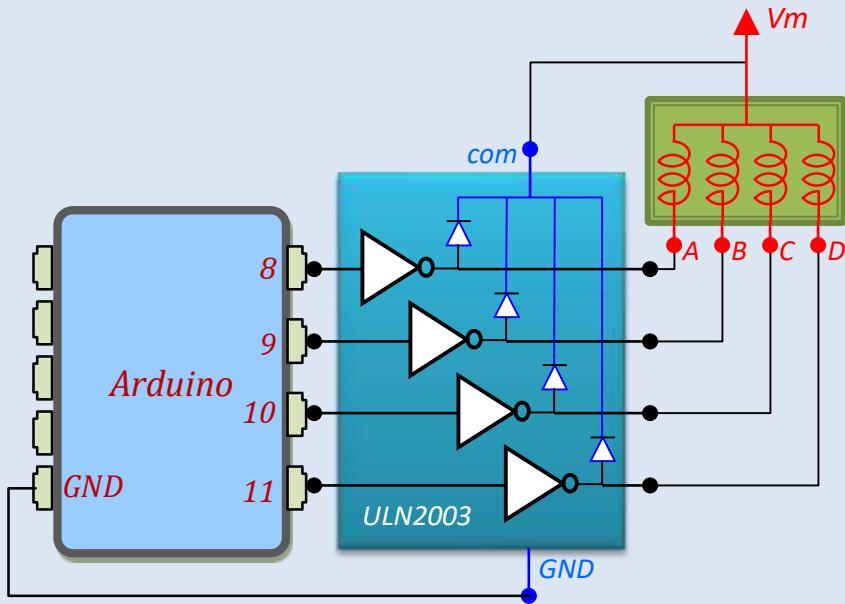
$$N_{M1} = 6 \times 4 = 24 \rightarrow 15^\circ$$

$$N_{M2} = 25 \times 4 = 100 \rightarrow 3.6^\circ$$

$$N_{M3} = 50 \times 4 = 200 \rightarrow 1.8^\circ$$

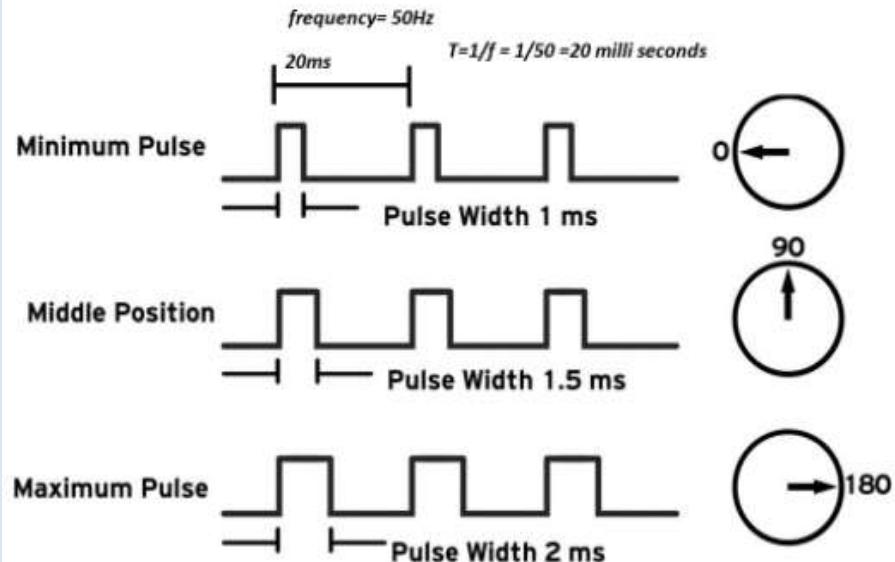
Exemple commande moteur pas à pas

- ① 10 pas en avant, 5 pas en arrière
- ② $\frac{1}{2}$ seconde par pas

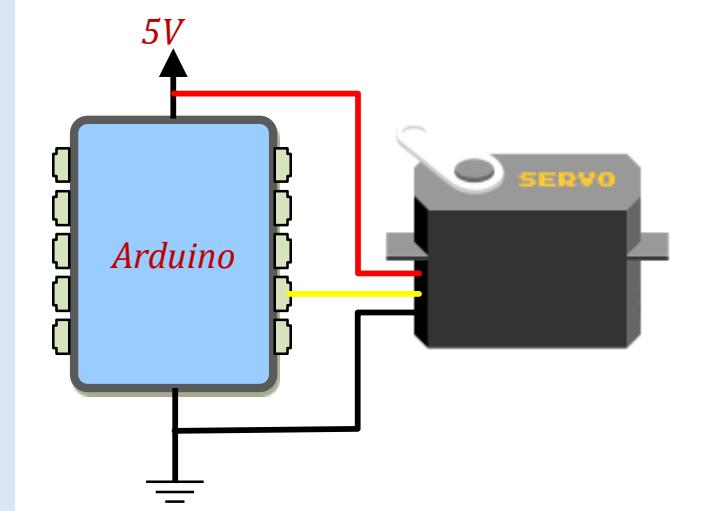


```
void setup() {
    DDRB = B1111;
    PORTB = B0000001;
}
void loop() {
    gauche(10, 500);
    droite(5,500);
}
void gauche(int N, int T){
    for(int i =0; i<N; i++){
        PORTB <<= 1;
        if( PORTB == B00010000) PORTB = B00000001;
        delay(T);
    }
}
void droite(int N, int T){
    for(int i =0; i<N; i++){
        PORTB >>= 1;
        if(PORTB == B00000000) PORTB = B0001000;
        delay(T);
    }
}
```

Commande d'un servomoteur



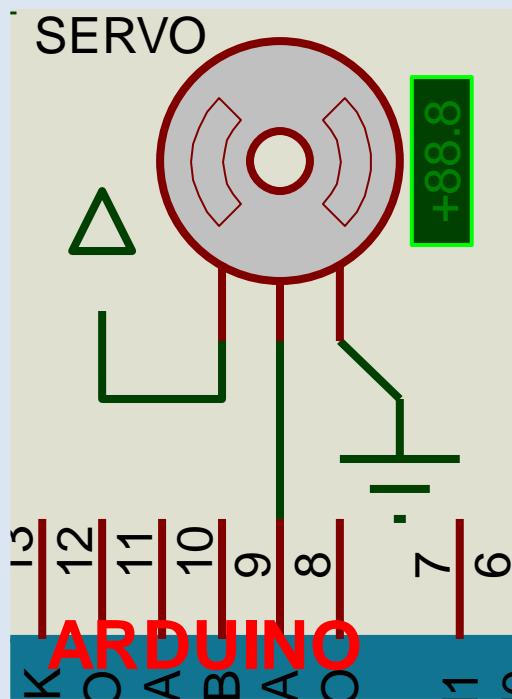
- positionnement angulaire ou rotation continue
- position angulaire de 0 à 180°
- Commande par un signal PWM 50Hz (20 ms). Timing sur la figure. (peut changer selon les servos, faire des tests)



La librairie servo

- Générer des signaux PWM 50Hz ($T=20\text{ms}$) pour commander des servomoteurs
- **attach(pin, [min, max]);** Définit la pate Arduino connectée au servomoteur. Les paramètre optionnels *min* et *max* définissent la largeur minimale (pour 0°) et maximale (pour 180°) de l'impulsion PWM (en μs). Par défaut [544, 2400]
- **write(A);** Génère un signal PWM avec une largeur d'impulsion correspondant à l'angle *A*
- **writeMicroseconds (U);** Génère un signal PWM (50Hz) avec une impulsion de largeur *U* μs
- **read();** retourne la position actuelle du moteur (dernier paramètre envoyé par la fonction write()),
- **detach();** Libère la pate attachée par attach()

Programme pour tester un Servomoteur



Minimum Angle:	0
Maximum Angle:	180
Rotational Speed:	60
Minimum Control Pulse:	1m
Maximum Control Pulse:	2m

```
#include <Servo.h>
Servo servo1;

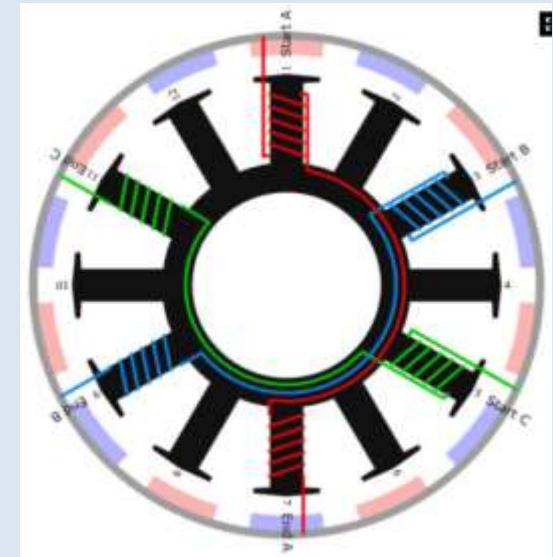
void setup() {
    servo1.attach(9,1000,2000);
    for(int A=0; A <= 180; A+=5){
        servo1.write(A);
        delay(100);
    }
    delay(2000);
    for(int A=180; A >= 0; A-=5){
        servo1.write(A);
        delay(100);
    }
}

void loop() {
```

- Petit programme qui fait un aller retour

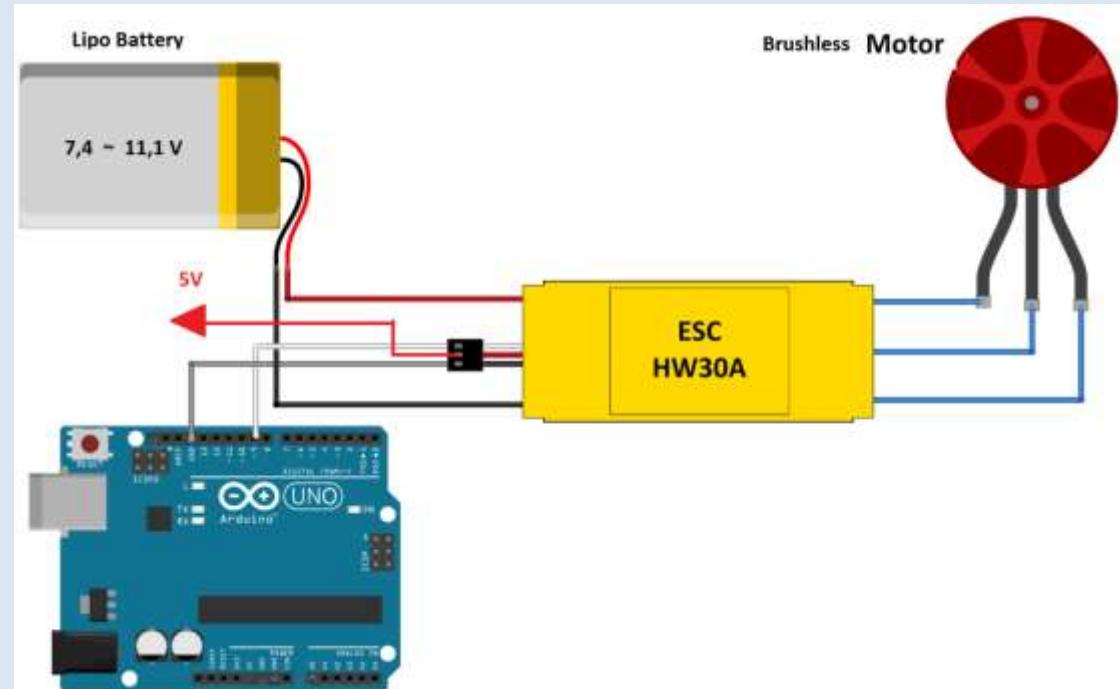
Le moteur Brushless

- Ressemble au moteur CC mais sans collecteurs
- Le stator est constitué de 3 bobines A, B, C et le rotor est constitué d'aimants permanents. Le rotor peut être autour du stator ou l'inverse (outrunner, inrunner)



- Il se commande un peu comme un moteur pas à pas: On alimente les bobines séquentiellement ce qui crée un champ magnétique tournant. A chaque fois, Le rotor se déplace pour aligner le champs avec les aimants permanant.
- L'opération es délicate car le champ magnétique doit toujours être en avance sur le rotor pour créer un couple toujours dans le même sens. Pour cette raison on utilise un contrôleur électronique pour moteur brushless

Le contrôleur ESC-30A



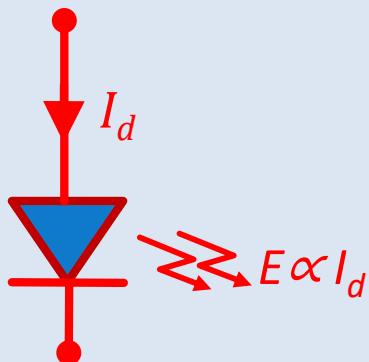
- Le contrôleur ESC-30A est un des contrôleurs du commerce permettant de piloter un moteur brushless (30A). La mention BEC 5V/2A précise que le contrôleur fournit une sortie 5V qui peut servir à alimenter l'Arduino (sur la puce 5V) à partir de la batterie. Avec un BEC > 5V, on peut toujours alimenter l'Arduino mais sur la puce V_{in}
- L'ESC-30 se contrôle à l'aide d'un signal PWM exactement comme un servomoteur. On peut donc utiliser la librairie **servo** pour le commander.
- Pour tester un moteur brushless, on peut utiliser le petit programme que nous avons utilisé pour tester le servomoteur

Exemple PWM

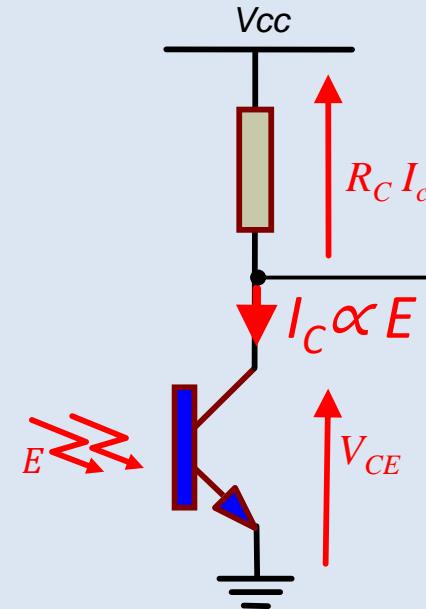
- brancher deux boutons poussoir BP1 et BP2 sur l'Arduino
- Faire le programme qui génère un signal PWM sur la pate 9,
- Si on clique sur P1 => le rapport cyclique augmente de 5%,
- Si on clique sur P2 => le rapport cyclique diminue de 5%,
- Vérifier sur ISIS en visualisant le signal sur l'oscilloscope

Emetteur Récepteur Infrarouge

Emetteur Photodiode



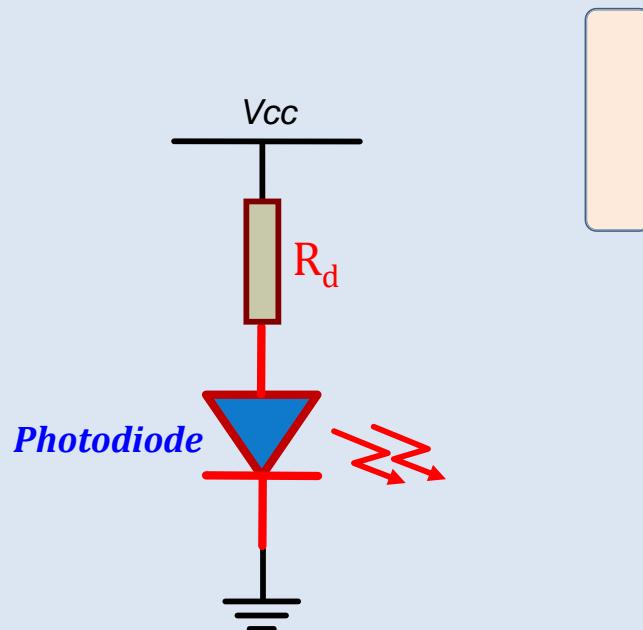
Récepteur Phototransistor



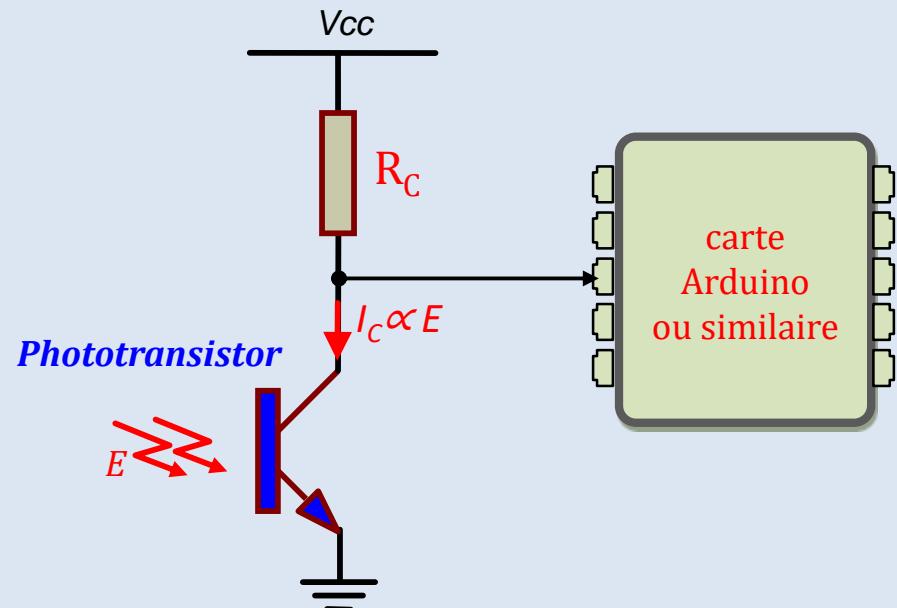
- $E=0 \rightarrow I_c=0 \rightarrow T \text{ bloqué} \rightarrow V_s = V_{cc}$
- $E >> 0 \rightarrow I_c = I_{csat} \rightarrow T \text{ saturé} \rightarrow V_s \leq 0.2V$

Détecteur de passage

Emetteur

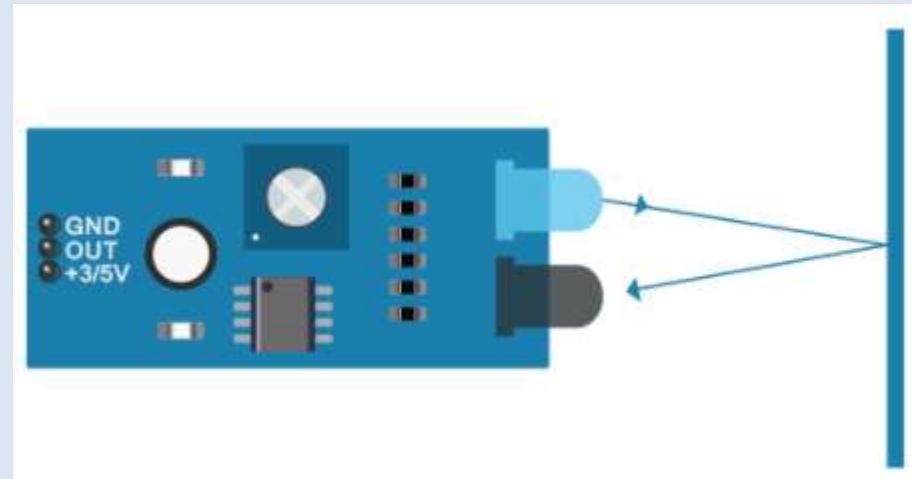


Récepteur

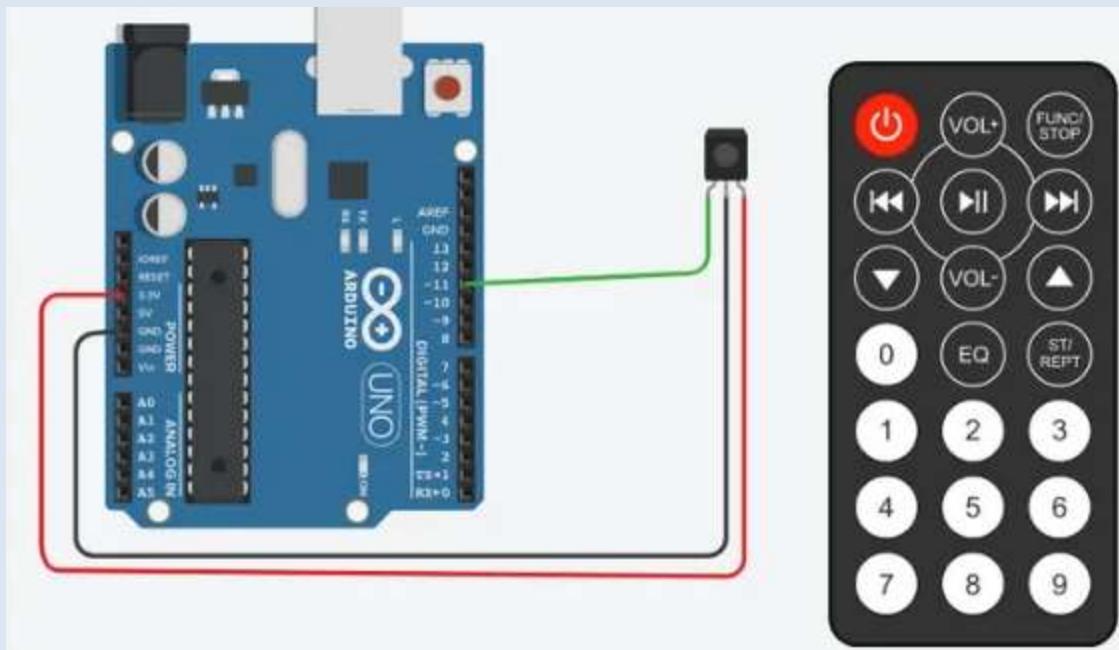


- Pas d'obstacle $\rightarrow T$ éclairé \rightarrow Saturé $\rightarrow V_{out} \approx 0$
- Obstacle $\rightarrow T$ non éclairé \rightarrow Bloqué $\rightarrow V_{out} = V_{cc}$

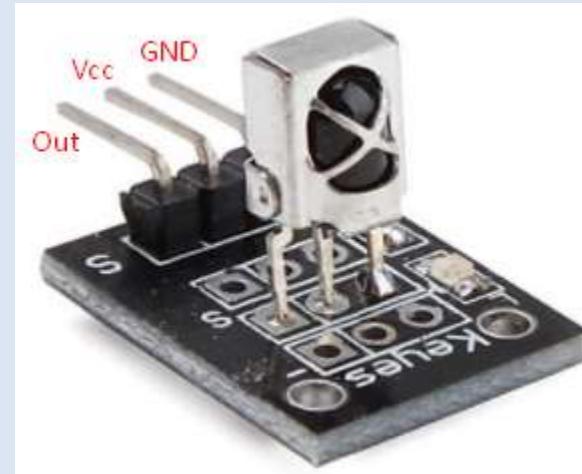
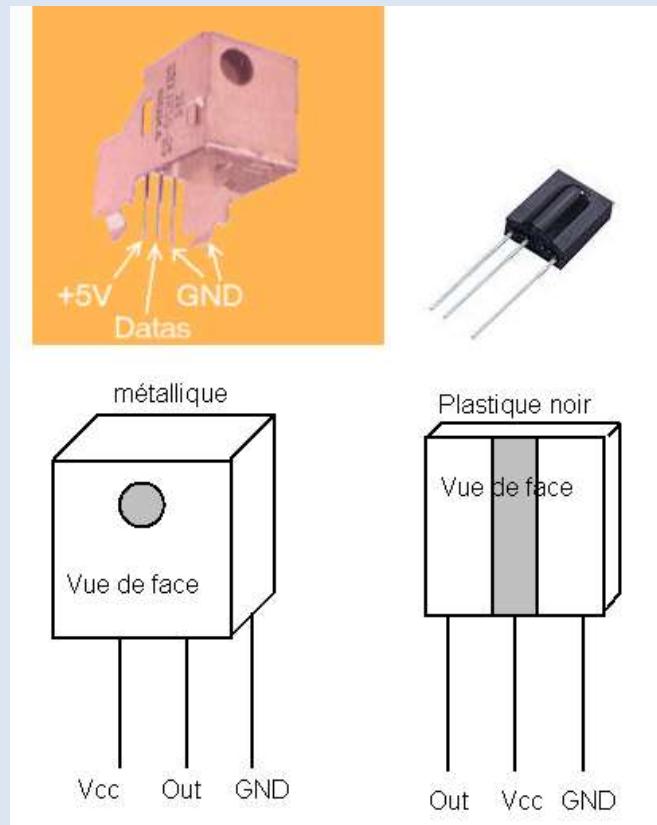
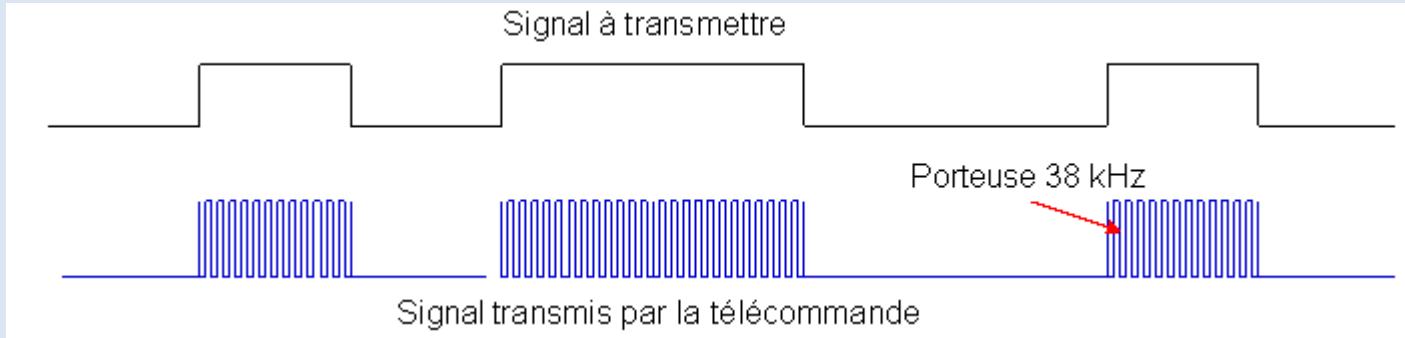
Détecteur de proximité



- Obstacle → T éclairé → Saturé → $V_{out} \approx 0$
- Pas d'Obstacle → T non éclairé → Bloqué → $V_{out} = V_{cc}$



Modulation

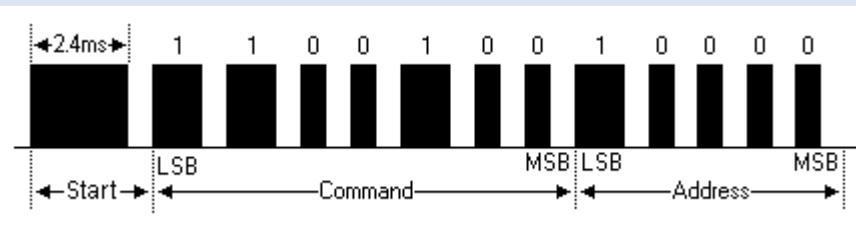


Le récepteur supprime la porteuse et restitue le signal utile

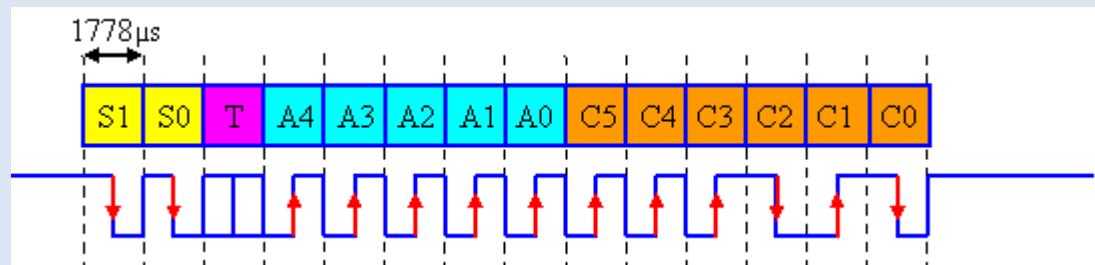
Codage et protocole

Pour minimiser les problèmes de synchronisation et les erreurs de communication, des codages de lignes plus au moins sophistiqués sont utilisés

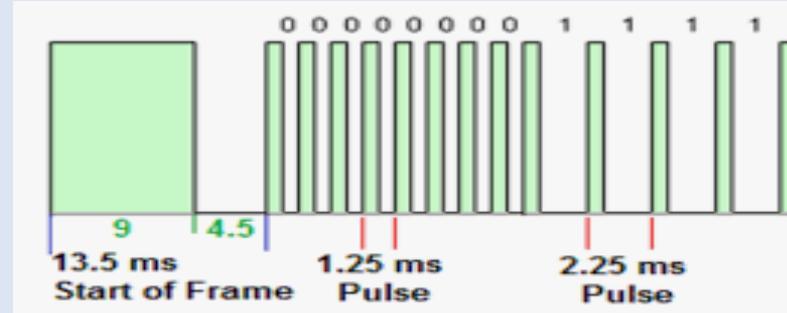
SONY SIRC



Philips RC5



NEC



La librairie IRremote

Décoder et transmettre des signaux IR des différents codes populaires, RC5, RC6, SONY, NEC, SAMSUNG

Comme toujours avec ce genre de librairie, il n'ya pas de documentation détaillée, il faut s'inspirer des exemples

```
#include <IRremote.h> // On inclut le fichier des headers
IRrecv myreceiver(PIN); // on crée un objet (avec le nom de notre choix) en
                        précisant la pate connectée au récepteur
decode_results RES;    // On déclare une variable pour stocker le code reçu

void setup(){
    myreceiver.enableIRIn(); // valider la réception
    myreceiver.blink13(true); // feedback réception sur la LED intégrée
    .....
}

• myreceiver.decode(&RES)) → vrai si on a reçu quelque chose de valide
• RES.decode_type: → NEC, SAMSUNG, SONY, RC5, RC6, or UNKNOWN.
• RES.value: → Le code reçu
• RES.bits: → Le nombre de bits utilisés par le code
```

Test Télécommande 1

```
/* afficher sur le moniteur série le nom du protocole suivi du
code de la touche actionnée */

#include <IRremote.h>
IRrecv myreceiver(2); // on crée l'objet myreceiver sur la pate 2
decode_results RES; // variable pour stocker le code reçu

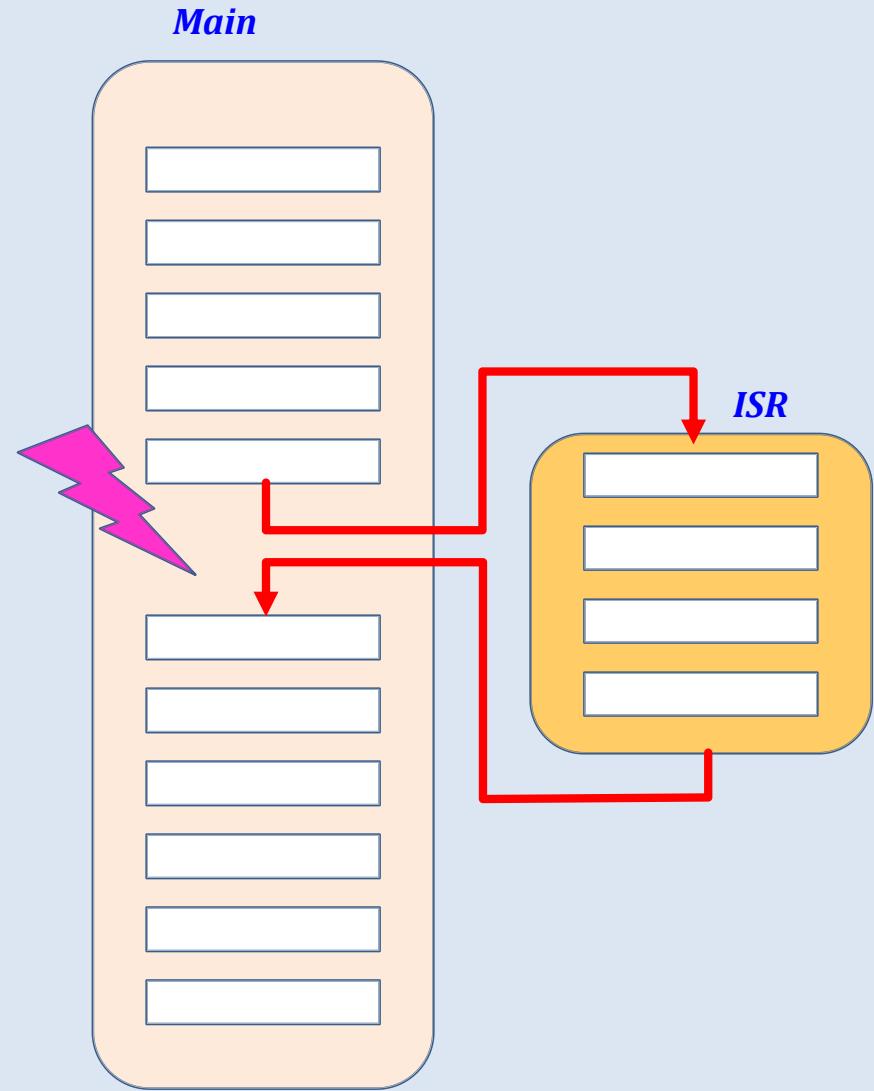
void setup(){
    myreceiver.enableIRIn(); // valider la réception
    myreceiver.blink13(true); // feedback sur la LED intégrée
    Serial.begin(115200); // pour afficher le code reçu sur le
moniteur série
}

void loop() {
    if (myreceiver.decode(&RES)) { // si on reçoit quelque chose
valide
        switch (RES.decode_type) {
            case NEC:
                Serial.print("NEC: ");
                break;
            case SONY:
                Serial.print("SONY: ");
                break;
            case RC5:
                Serial.print("RC5: ");
                break;
            case RC6:
                Serial.print("RC6: ");
                break;
            case SAMSUNG:
                Serial.print("SAMSUNG: ");
                break;
            case UNKNOWN:
                Serial.print("UNKNOWN: ");
                break;
        }
        Serial.println(RES.value, HEX);
        delay(1000);
        myreceiver.resume();
    }
}
```

```
case SONY:
    Serial.print("SONY: ");
    break;
case RC5:
    Serial.print("RC5: ");
    break;
case RC6:
    Serial.print("RC6: ");
    break;
case SAMSUNG:
    Serial.print("SAMSUNG: ");
    break;
case UNKNOWN:
    Serial.print("UNKNOWN: ");
    break;
}
```

Les interruptions

Une interruption est un **événement** qui provoque la suspension temporaire du programme en cours pour exécuter une **procédure** d'interruption (*ISR: Interrupt Service Routine*) liée à l'événement. A la fin de cette procédure, le microcontrôleur reprend le programme à l'endroit où il l'a laissé



Les interruptions sur Arduino

- Les cartes type Arduino n'ayant pas tous le même processeur, Elles n'ont pas les mêmes sources interruption.
- Sur Arduino Uno (AVR328), il y a 26 sources d'interruptions :
 - Les interruptions **externes** provoquée par le changement d'état sur certaines entrées numériques,
 - Les interruptions liés aux **timers** qui sont des compteurs internes qui permettent de gérer le temps,
 - D'autres modules internes peuvent déclencher des interruptions : USART, I2C, SPI, ADC, EEPROM, flash, ...
- Les interruptions sont validés/interdites soit globalement soit individuellement,
- Chaque interruption a au moins un bit de validation et un drapeau (flag)
- Quand l'événement d'une interruption intervient, le flag se lève, si l'interruption ne peut être traitée immédiatement pour cause de priorité ou de validation, le flag reste levé, et l'interruption reste en attente

General Interrupt Enable bit

7	6	5	4	3	2	1	0	SREG
I	T	H	S	V	N	Z	C	
R/W								

- Le bit **I** du registre d'état SREG permet de valider/interdire toutes les interruptions
- Pour valider en langage C, on a plusieurs façon:
 - *interrupts();*
 - *sei();*
 - *bitSet(SREG, 7);*
 - *SREG |= 0b10000000;*
- Pour désactiver en langage C , on a plusieurs façon:
 - *noInterrupts ();*
 - *cli();*
 - *bitClear(SREG, 7);*
 - *SREG &= 0b01111111;*

La fonction d'interruption ISR

- Sur Arduino, la fonction ISR a certaines limitations:
- Elle ne peut pas recevoir de paramètre et ne retourne rien: `void xxxxxxxx() {...}`
- Elle doit être courte avec un temps d'exécution le plus court possible,
- Les fonctions *delay()* et *millis()* ne fonctionnent pas.
micro() fonctionne correctement pendant $\approx 1\text{ms}$.
delayMicroseconds() fonctionne correctement ,
- L'accès au port série ne fonctionnent pas
- Pour partager des variables avec l'ISR, il faut les déclarer en global ***volatile***,

Les Interruptions Externes

- Sur Arduino UNO, les entrées numériques 2 et 3 peuvent déclencher les interruptions externes INT0 et INT1. Sur les autres cartes, il faut consulter la doc:

	<i>int.0</i>	<i>int.1</i>	<i>int.2</i>	<i>int.3</i>	<i>int.4</i>	<i>int.5</i>
<i>Uno, Ethernet</i>	2	3				
<i>Mega2560</i>	2	3	21	20	19	18
<i>32u4 based</i>	3	2	0	1	7	

- Pour valider l'interruption il faut utiliser la fonction ***attachInterrupt(int_number, ISR_fonc, mode);***
 - ***int_number:*** numéro de l'interruption: 0 → pin2, 1 → pin3. On peut utiliser la fonction ***digitalPinToInterrupt(pin)*** qui retourne le numéro d'interruption associé à une pate donnée
 - ***ISR_fonc:*** nom de la fonction ISR qui doit être exécutée quand l'interruption se déclenche
 - ***mode:*** Evénement déclencheur: LOW, FALLING, RISING, CHANGE
- Pour désactiver l'interruption, il faut utiliser la fonction ***detachInterrupt(int_number);***

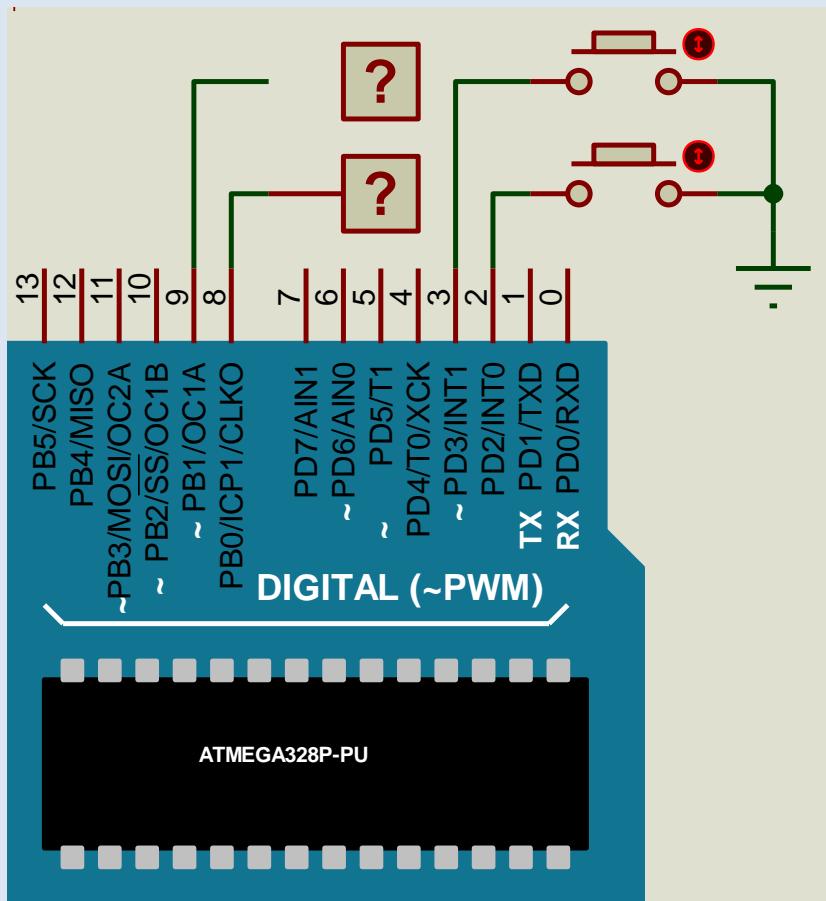
Vecteurs d'interruption de Atmega328P

Vector Number	Interrupt definition	Vector name
2	External Interrupt Request 0	INT0_vect
3	External Interrupt Request 1	INT1_vect
4	Pin Change Interrupt Request 0	PCINT0_vect
5	Pin Change Interrupt Request 1	PCINT1_vect
6	Pin Change Interrupt Request 2	PCINT2_vect
7	Watchdog Time-out Interrupt	WDT_vect
8	Timer/Counter2 Compare Match A	TIMER2_COMPA_vect
9	Timer/Counter2 Compare Match B	TIMER2_COMPB_vect
10	Timer/Counter2 Overflow	TIMER2_OVF_vect
11	Timer/Counter1 Capture Event	TIMER1_CAPT_vect
12	Timer/Counter1 Compare Match A	TIMER1_COMPA_vect
13	Timer/Counter1 Compare Match B	TIMER1_COMPB_vect
14	Timer/Counter1 Overflow	TIMER1_OVF_vect
15	Timer/Counter0 Compare Match A	TIMERO_COMPA_vect
16	Timer/Counter0 Compare Match B	TIMERO_COMPB_vect
17	Timer/Counter0 Overflow	TIMERO_OVF_vect
18	SPI Serial Transfer Complete	SPI_STC_vect
19	USART Rx Complete	USART_RX_vect
20	USART Data Register Empty	USART_UDRE_vect
21	USART Tx Complete	USART_TX_vect
22	ADC Conversion Complete	ADC_vect
23	EEPROM Ready	EE_READY_vect
24	Analog Comparator	ANALOG_COMP_vect
25	Two-wire Serial Interface	TWI_vect
26	Store Program Memory Read	SPM_READY_vect

```
ISR(vector name) {
    // ISR code
}
```

Exemple

Deux boutons pour contrôler deux LEDs



```
#define BP1 2
#define BP2 3
#define LED1 8
#define LED2 9
volatile boolean LED1_STATE=false, LED2_STATE=false;
void setup() {
    pinMode(BP1,INPUT_PULLUP);
    pinMode(BP2,INPUT_PULLUP);
    pinMode(LED1,OUTPUT);
    pinMode(LED2,OUTPUT);
    digitalWrite(LED1,LED1_STATE);
    digitalWrite(LED2,LED2_STATE);
    bitSet(EIFR,INTF0); // baisser le drapeau de INT0
    bitSet(EIFR,INTF1); // baisser le drapeau de INT1
    attachInterrupt(0,led1_isr, FALLING);
    attachInterrupt(1,led2_isr, FALLING);
}
void loop() { }
void led1_isr(){
    LED1_STATE = !LED1_STATE;
    digitalWrite(LED1,LED1_STATE);
}
void led2_isr(){
    LED2_STATE = !LED2_STATE;
    digitalWrite(LED2,LED2_STATE);
}
```

- Le programme ne commence à surveiller l'entrée d'interruption qu'au bout de 65ms
- Entre le moment où l'événement se produit et le début de l'exécution de l'ISR il se passe à peu près 1,5 µs
- Les fonctions `digitalRead()` et `digitalWrite()` mettent à peu près 3,5 µs pour s'exécuter

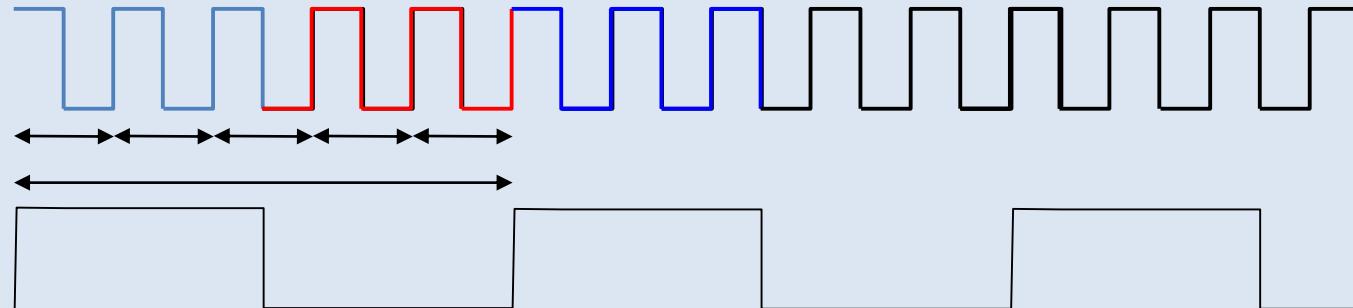
Exercice

- 1) Programme qui réalise un diviseur de fréquence par 5

Algorythme:

chaque 5 transitions du signal d'entrée, on change l'état du signal de sortie

- 2) Programme qui réalise un diviseur de fréquence programmable. Le rapport de division sera lu sur PORTB



Les Timers de l'ARDUINO



Un **Timer** est un compteur incrémenté par une horloge dont la période est connue → compteur de temps. Mais on peut lui appliquer une horloge externe, il devient un simple compteur d'événement

Arduino UNO (AVR328) contient 3 Timers:

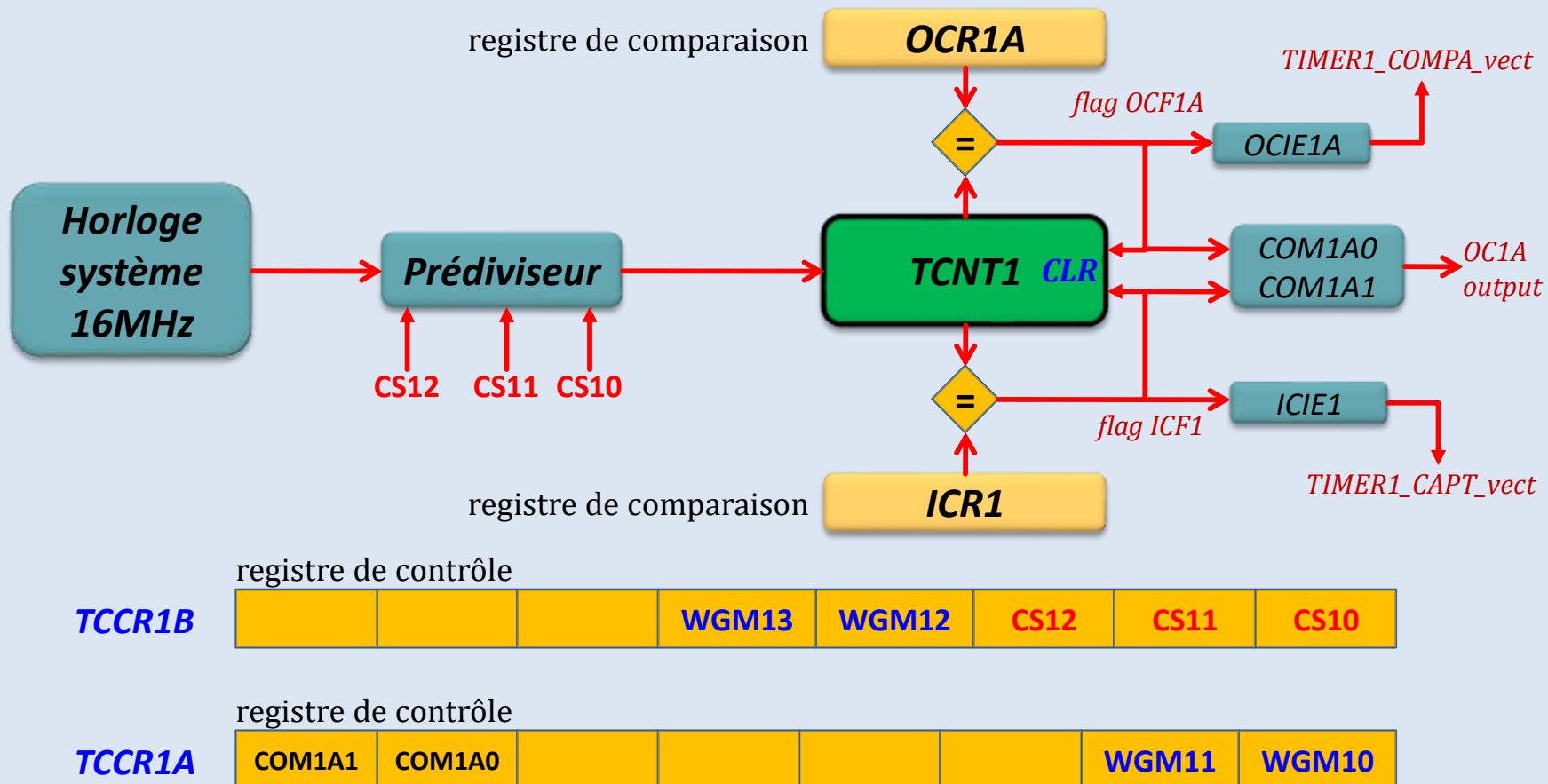
- TIMER 0 (**TC0**) → 8 bits (compte de 0 à 255)
- TIMER 1 (**TC1**) → 16 bits (compte de 0 à 65535)
- TIMER 2 (**TC2**) → 8 bits (compte de 0 à 255)

Les 3 Timers ont un fonctionnement assez similaire. Ils ont de multiples fonctions +/- complexes qui ne peuvent tous être traitées ici. Nous allons nous contenter de quelques exemples avec le TIMER1

Chaque Timer est constitué:

- ❑ Du compteur
- ❑ Du diviseur de fréquence programmable
- ❑ D'un ensemble de registres du control

Le Timer TMR1



Le Timer peut fonctionner en mode **Normal**, **Comparaison (CTC)** ou **PWM**. Seuls les bits de control nécessaires pour les modes Normal et CTC sont représentés.

Mode de fonctionnement

WGM13	WGM12	WGM11	WGM10	mode	Cycle	Int flag	Validation	ISR
0	0	0	0	NORMAL	0→FFFF	TOV1	TOIE1	ISR(TIMER1_OVF_vect)
0	1	0	0	CTC(1)	0→ OCR1A	OCF1A	OCIE1A	ISR(TIMER1_COMPA_vect)
1	1	0	0	CTC(2)	0→ ICR1	ICF1 OCF1A	ICIE1 OCIE1A	ISR(TIMER1_CAPT_vect) ISR(TIMER1_COMPA_vect)
x	x	x	x	PWM				

7	6	5	4	3	2	1	0	TIFR1
-	-	ICF1	-	-	OCF1B	OCF1A	TOV1	
R	R	R/W	R	R	R/W	R/W	R/W	
7	6	5	4	3	2	1	0	TIMSK1
-	-	ICIE1	-	-	OCIE1B	OCIE1A	TOIE1	
R	R	R/W	R	R	R/W	R/W	R/W	

Dans les mode CTC, on peut générer un signal sur la sortie OC1A (pin 9 sur Arduino) conformément aux bits COM1A1, COM1A0

COM1A1	COM1A0	Sortie OC1A
0	0	Non utilisée
0	1	Change à chaque égalité
1	0	Forcée à LOW à chaque égalité
1	1	Forcée à HIGH à chaque égalité

Le Prédiviseur (Prescaler)

La valeur de la période d'horloge dépend du Prédiviseur qui est programmé à l'aide des bits ***CS1i*** du registre de contrôle TCCR1B

<i>CS12</i>	<i>CS11</i>	<i>CS10</i>	DIV	f	T	Cycle
0	0	0	Stoppé	0	∞	
0	0	1	1	16 MHz	62.5 ns	4.096 ms
0	1	0	8	2 MHz	0.5 μ s	32.768ms
0	1	1	64	250 kHz	4 μ s	262.144ms
1	0	0	256	62.5 kHz	16 μ s	1.048576 s
1	0	1	1024	15.625 kHz	64 μ s	4.194304 s

TMR1 mode normal

- Dans le mode normal (les 4 bits **WGM1i** = 0), le Timer **TCNT1** compte de 0 à 65535 (0xFFFF) et revient à 0,
- On peut à tout moment consulter ou modifier la valeur de **TCNT1**
- Quand **TCNT1** déborde, le flag d'interruption TOV1 est positionné
- Si le bit de validation TOIE a été positionné auparavant, l'interruption est déclenchée et la routine *ISR(TIMER1_OVF_vect)* est exécutée

7	6	5	4	3	2	1	0	
-	-	ICF1	-	-	OCF1B	OCF1A	TOV1	TIFR1
R	R	R/W	R	R	R/W	R/W	R/W	

7	6	5	4	3	2	1	0	
-	-	ICIE1	-	-	OCIE1B	OCIE1A	TOIE1	TIMSK1
R	R	R/W	R	R	R/W	R/W	R/W	

TMR1 mode Comparaison

- Il y a deux modes de comparaison. On va les appeler CTC1 et CTC2
- Dans le mode CTC1 (**WGM13:0 = 0100**), la valeur de **TCNT1** est comparée en permanence avec le registre OCR1A. Quand il y a égalité, le drapeau OCF1A passe à 1 (demande d'interruption *Timer/Counter1 compare match A*). Si cette interruption a été validée par le bit correspondant OCIE1A, elle se déclenche: La routine d'interruption ISR(TIMER1_COMPA_vect) est exécutée.
- Dans le mode CTC2 (**WGM13:0 = 1100**), la valeur de **TCNT1** est comparée en permanence avec le registre ICR1. Quand il y a égalité, le drapeau ICF1 passe à 1 (demande d'interruption *Timer/Counter1 capture event*). Si cette interruption a été validée par le bit correspondant OCIE1, elle se déclenche: La routine d'interruption ISR(TIMER1_CAPT_vect) est exécutée.

Les interruptions liées au Timer 1

7	6	5	4	3	2	1	0	
-	-	ICF1	-	-	OCF1B	OCF1A	TOV1	TIFR1
R	R	R/W	R	R	R/W	R/W	R/W	

7	6	5	4	3	2	1	0	
-	-	ICIE1	-	-	OCIE1B	OCIE1A	TOIE1	TIMSK1
R	R	R/W	R	R	R/W	R/W	R/W	

- **TIMSK1**: registre contenant les bits de validations des interruptions liés au Timer1
 - **TOIE1**: Validation de l'interruption de débordement de Timer1
 - **OCIE1A**: Validation de comparaison de Timer1 avec le registre OCR1A
 - **OCIE1B**: Validation de comparaison de Timer1 avec le registre OCR1B
 - **ICIE1**: Validation de comparaison de Timer1 avec le registre ICR1
- **TIFR1**: registre contenant les flags des interruptions liés au Timer1
 - **TOV1**: Se positionne quand le timer TCNT1 repasse à 0
 - **OCF1A**: Se positionne quand le timer TCNT1 devient égal au registre OCR1A
 - **OCF1B**: Se positionne quand le timer TCNT1 devient égal au registre OCR1B
 - **ICF1**: Se positionne quand le timer TCNT1 devient égal au registre ICR1

Exemples basiques en mode normal

```
/* Afficher l'avancement du timer1
```

NDIV	DIV	f	T (us)
1	1	16 MHz	0.0625
2	8	2 MHz	0.50
3	64	250 kHz	4.00
4	256	62.5 kHz	16.00
5	1024	15.625 kHz	64.00 */

```
unsigned long N,T;  
void setup() {  
    Serial.begin(57600);  
    TCCR1A = 0;  
    TCCR1B = 3; //start timer avec div=64, Th=4µs  
}  
void loop() {  
    N = TCNT1;  
    T = N * 4; // temps en µs  
    Serial.print(N);  
    Serial.write(' ');  
    Serial.println(T);  
}
```

```
/*générer un signal carré en utilisant l'interruption TOV1  
à chaque débordement, on change l'état de la pate 4 */
```

```
const byte OUTPIN = 4;  
  
void setup () {  
    pinMode (OUTPIN, OUTPUT);  
    cli(); // Désactiver les interruption lors la config du timer  
    TCCR1A = 0; // mode normal  
    TCCR1B = 1; // DIV=1, Th=62.5ns, Cycle=4.096ms  
    TIMSK1 = 0b00000001; // valider Interr. OV de TMR1  
    sei(); // Valider l'interruption globale  
}  
  
// Routine d'interruption  
ISR(TIMER1_OVF_vect) {  
    digitalWrite(OUTPIN, !digitalRead(OUTPIN));  
}  
  
void loop () {  
}
```

Autre exemple mode normal



- ❑ Déterminer les instants de transition d'un signal quelconque
- ❑ On démarre le Timer1 et on valide l'interruption INTO (CHANGE) pour détecter les transitions du signal.
- ❑ à chaque déclenchement, on relève la valeur du Timer 1

```
volatile uint8_t i=0, S[20];
volatile uint16_t N[20]; // valeurs du timer 1
uint32_t TA,TR; // temps en µs
void setup() {
  Serial.begin(57600);
  TCCR1A = 0; // mode normal
  TCCR1B = 3; // DIV=64, Th=4µs
  bitSet(EIFR,INTFO); // clear INTO interrupt flag
  attachInterrupt(0, toctoc, CHANGE);
}
```

```
void loop() {
  if(i == 7){
    detachInterrupt(0);
    for(int j=0; j < i; j++){
      TA = N[j] * 4;
      TR = (N[j]-N[0]) * 4;
      Serial.print(N[j]);
      Serial.print(" ");
      Serial.print(TA);
      Serial.print("us ");
      Serial.print(TR);
      Serial.print("us ");
      Serial.println(S[j]);
    }
    i=0;
  }
}
```

Virtual Terminal

1221	4884us	0us	1	
1471	5884us	1000us	0	
1721	6884us	2000us	1	
2221	8884us	4000us	0	
2471	9884us	5000us	1	
3221	12884us	8000us	0	
3971	15884us	11000us	1	

```
void toctoc(){
  N[i]=TCNT1;
  S[i]=digitalRead(2);
  i++;
}
```

Exemples basiques en mode CTC

/* Clignoter une LED

NDIV	DIV	f	T (us)
------	-----	---	--------

1	1	16 MHz	0.0625
2	8	2 MHz	0.50
3	64	250 kHz	4.00
4	256	62.5 kHz	16.00
5	1024	15.625 kHz	64.00 */

```
void setup(){
```

```
  pinMode(9,OUTPUT);
```

```
  cli();
```

```
  OCR1A = 31249;
```

```
  TCCR1A = B01000000;// CTCmode, OC1A=toggle
```

```
  TCCR1B = B00001100; //CTCmode, Th=16us
```

```
}
```

```
void loop() {
```

```
}
```

/*

Signal carré de période 2 x 40µs

*/

```
void setup(){
```

```
  pinMode(9,OUTPUT);
```

```
  cli();
```

```
  OCR1A = 9; // cycle=(OCR1A+1)*Th
```

```
  TCCR1A = B01000000;// CTCmode,OC1A=toggle
```

```
  TCCR1B = B00001011; //CTCmode, Th=4us
```

```
}
```

```
void loop() {
```

```
}
```

Encore un exemple

```
/* Acquisition de deux signaux analogiques
avec des rythmes d'échantillonnage différents
A0->0.5s, A1-> 2s
*/
volatile byte TOC=LOW;
int N,M;
unsigned long NEW, OLD=0;
void setup(){
    Serial.begin(57600);
    cli();
    TCCR1A = B00000000;
    TCCR1B = B00011100; //CTC2mode,Th=16us
    TIMSK1 = B00000010;
    = 31249;// cycle=31250*16us
    sei();
}
void loop() {
    if(TOC){
        N = analogRead(A0);
        Serial.print("A0----->");
        Serial.println(N*5.0/1023.0);
        TOC=LOW;
    }
}
```

```
NEW = millis();
if((NEW-OLD) > 2000){
    OLD = NEW;
    M = analogRead(A1);
    Serial.print("A1=====>");
    Serial.println(M*5.0/1023.0);
}
}

ISR(TIMER1_COMPA_vect) {
    TOC=HIGH;
}
```

