Implémentation de commandes numériques en temps réel

A-Objectif de l'unité

L'objectif de cette unité d'enseignement c'est de traiter et implémenter la commande numérique en temps réel des ensembles convertisseurs machines par composants programmables tels que les µContrôleurs, DSP, ARM et FPGA.

B-Processeurs numériques

Ce sont des composants programmables et offrent la possibilité de réaliser des commandes en temps réel.

B.1-Microcontrôleur

Un µContôleurs (PIC) est une unité de traitement et d'exécution de l'information à laquelle on a ajouté des périphériques internes permettant de réaliser des montages sans nécessiter l'ajout de composants annexes. Un microcontrôleur PIC peut donc fonctionner de façon autonome après programmation.

Les PIC intègrent une mémoire programme non volatile (FLASH), une mémoire de données volatile, une mémoire de donnée non volatile (E2PROM), des ports d'entrée-sortie (numériques, analogiques, MLI, UART, bus I2C, Timers, SPI, etc.), et même une horloge, bien que des bases de temps externes puissent être employées. Certains modèles disposent de ports et unités de traitement de l'USB et Ethernet. Les microcontrôleurs sont fréquemment utilisés dans les systèmes embarqués, comme les contrôleurs des moteurs automobiles, les télécommandes, les appareils de bureau, l'électroménager, les jouets, la téléphonie mobile, etc.

Dans cette unité d'enseignement, on va s'intéresser à un microcontrôleur d'architecture Atmel AVR comme l'Atmega328p. Il est intégré dans une plateforme de prototypage appelée Arduino qui est open-source et permet aux utilisateurs de créer des objets électroniques interactifs à partir de cartes électroniques matériellement libres.

B.2-Processeur de signal numérique DSP

Un DSP de l'anglais ''Digital Signal Processor'', qu'on pourrait traduire par ''processeur de signal numérique'' ou ''traitement numérique de signal'' est un microprocesseur optimisé pour exécuter des applications de traitement numérique du signal (filtrage, extraction de signaux, etc.) le plus rapidement possible. Les DSP ont permis d'envisager des commandes numériques de plus en plus complexes. Ils sont utilisés dans la plupart des applications du traitement numérique du signal en temps réel. On les trouve dans les modems (modem RTC, modem ADSL), les téléphones mobiles, les appareils multimédia (lecteur MP3), les récepteurs GPS... Ils sont également utilisés dans des systèmes vidéo, les chaînes de traitement de son, partout où l'on reçoit un signal complexe que l'on doit modifier à l'aide du filtrage.

C-Plate-forme Arduino Uno

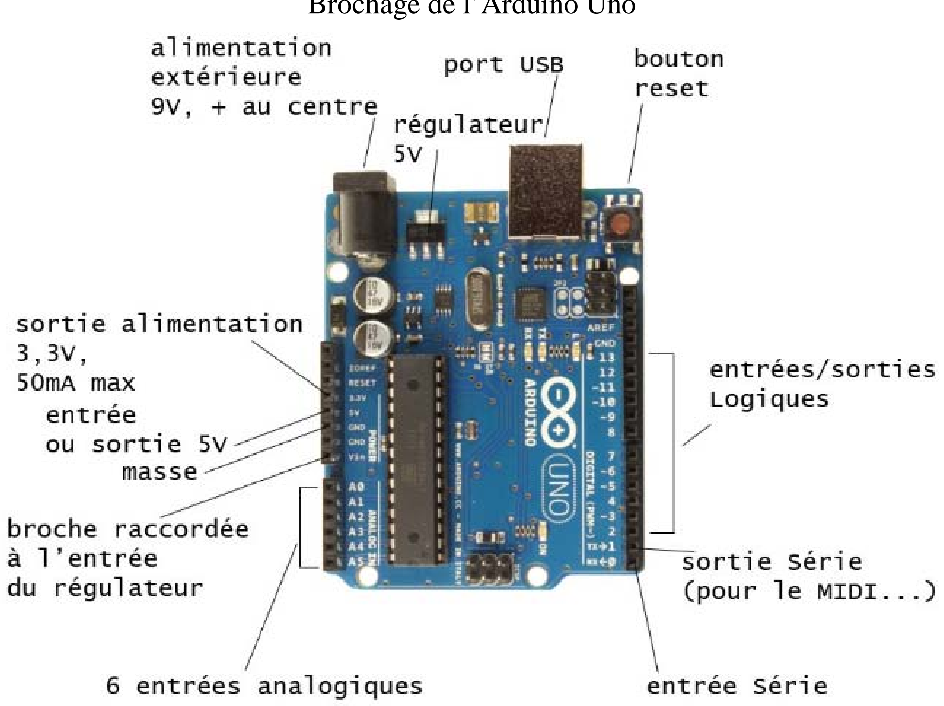
C.1- Présentation de l'Arduino Uno

Arduino est une plate-forme de prototypage d'objets interactifs à usage créatif constituée d'une carte électronique contenant un microcontrôleur et d'un environnement de programmation. Cet environnement matériel et logiciel permet à l'utilisateur de formuler ses projets par l'expérimentation directe avec l'aide de nombreuses ressources disponibles en ligne. Arduino est un projet en source ouverte: une importante communauté d'utilisateurs l'utilise. Il peut analyser et produire des signaux électriques de manière à effectuer des tâches très diverses.

La programmation sur Arduino se fait dans un langage qui s'inspire à la fois du C et du C++. Le C++ intervient surtout pour la création de librairies. Pour programmer cette carte, on utilise logiciel IDE Arduino. Il est utilisé dans plusieurs applications comme l'électrotechnique industrielle et embarquée, la domotique, le domaine de pilotage de robots, la commande des moteurs, les jeux de lumières, la communication avec l'ordinateur et la commande des appareils mobiles.

Parmi cette famille d'Arduino, on rencontre l'Arduino Uno qui contient un microcontrôleur d'architecture Atmel AVR et de type Atmega328p et dont le brochage est donné ci-dessous.

Brochage de l'Arduino Uno



C.2- Constitution de l'Arduino Uno

La carte Arduino UNO possède des «ports» permettant par exemple de se connecter à un ordinateur ou de s'alimenter.

La carte Arduino UNO est la pièce maîtresse de tout circuit électronique pour les débutants Elle est dotée:

- de 14 entrées/sorties (dont 6 fournissent la sortie PWM)
- 6 entrées analogiques
- un cristal à 16MHz
- une connexion USB
- une prise jack d'alimentation
- un en-tête ICSP
- un bouton reset.

C.3- Caractéristiques de la carte Arduino Uno

La carte ArduinoUno a les caractéristiques suivantes :

- Tension d'alimentation interne = 5V
- Tension d'alimentation (recommandée)= 7 à 12V, limites =6 à 20 V
- Entrées/sorties numériques : 14 dont 6 sorties MLI
- Entrées analogiques = 6
- Courant max par broches E/S = 40 mA
- Courant max sur sortie 3,3V = 50mA
- Mémoire Flash 32 kB
- Mémoire SRAM 2 kB
- Mémoire EEPROM 1 KB
- Fréquence horloge = 16 MHz ; Dimensions = 68.6mm x 53.3mm

La carte s'interface au PC par l'intermédiaire de sa prise USB. La carte s'alimente par le jack d'alimentation (utilisation autonome) mais peut être alimentée par l'USB (en phase de développement par exemple).

C.4- Avantages de l'Arduino

Il se caractérise par de nombreux avantages :

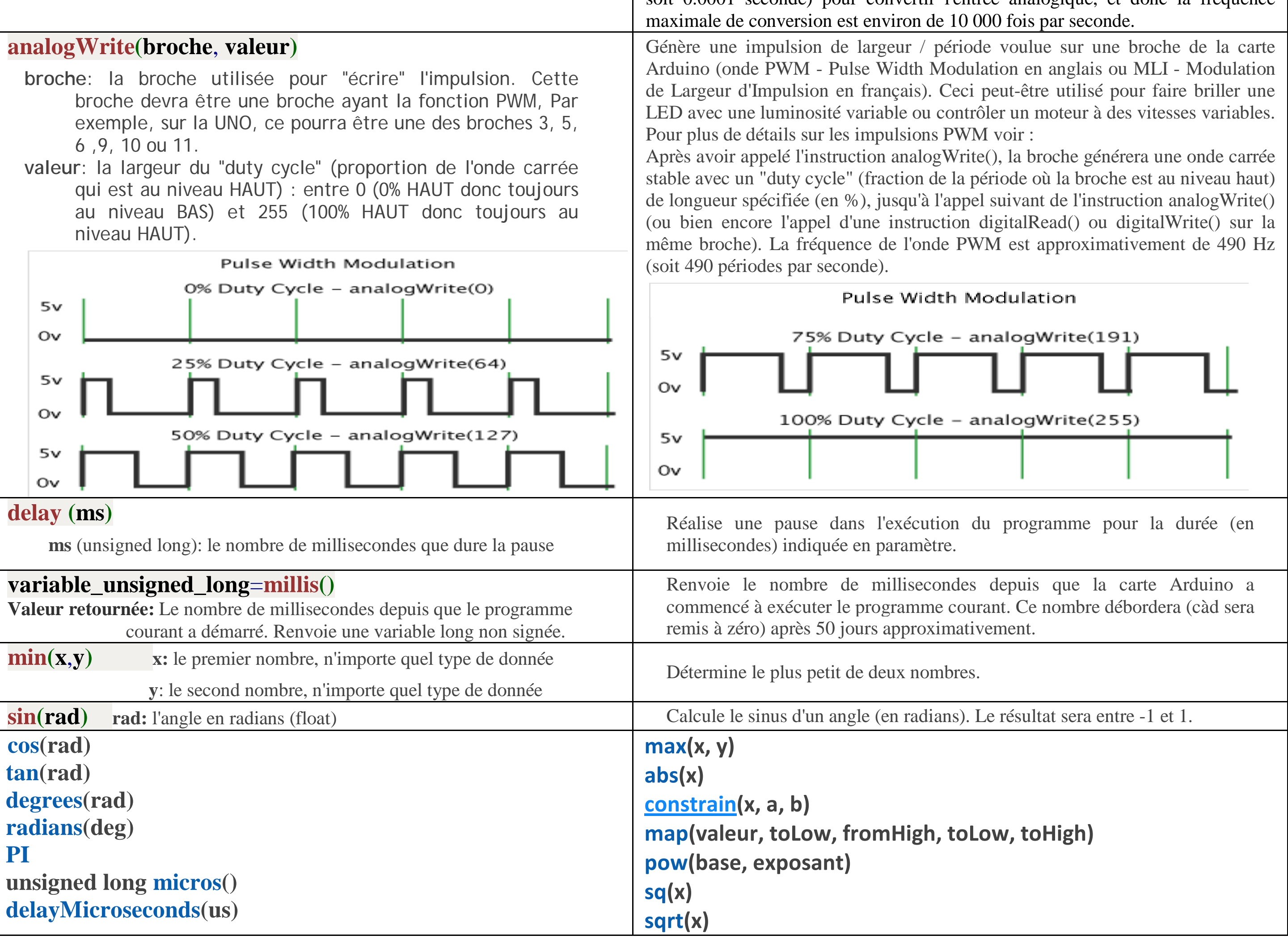
- Environnement de programmation clair, simple et gratuit
- Multiplateforme : tourne sous Windows, Macintosh et Linux,
- Nombreuses librairies disponibles avec diverses fonctions implémentées,
- Logiciel et matériel open source et extensible,
- Nombreux conseils, tutoriaux et exemples en ligne (forums, site perso etc.).
- Pas cher,

C.5- Fonctions de l'IDE de l'Arduino

Les fonctions les plus utilisées de l'IDE de l'Arduino sont décrites comme suit :

http://www.mon-club-elec.fr/pmwiki_reference_arduino/pmwiki.php?n=Main.Reference

Syntaxe de la fonction	Description
pinMode(**broche, mode**) **broche**: le numéro de la broche de la carte Arduino dont le mode de fonctionnement (entrée ou sortie) doit être défini. **mode**: soit INPUT (entrée en anglais) (=0) ou OUTPUT (sortie en anglais) (=1)	Configure la broche spécifiée pour qu'elle se comporte soit en mode entrée, soit en mode sortie.
digitalWrite(broche, valeur) **broche**: le numéro de la broche de la carte Arduino **valeur** : HIGH ou LOW (ou bien 1 ou 0)	Met un niveau logique HIGH (HAUT en anglais) ou LOW (BAS en anglais) sur une broche numérique. Si la broche a été configurée en SORTIE avec l'instruction pinMode(), sa tension est mise à la valeur correspondante : 5V (ou 3.3V sur les cartes Arduino 3.3V) pour le niveau HAUT, 0V (masse) pour le niveau BAS
digitalRead(broche) **broche**: le numéro de la broche de la carte Arduino	Lit l'état (= le niveau logique) d'une broche précise en entrée numérique, et renvoie la valeur HIGH (HAUT) ou LOW.
analogRead(broche_analogique) **broche_analogique** : le numéro de la broche analogique (et non le numéro de la broche numérique) sur laquelle il faut convertir la tension analogique appliquée (0 à 5 sur la plupart des cartes Arduino, 0 à 7 sur la Mini et la Nano)	Lit la valeur de la tension présente sur la broche spécifiée. La carte Arduino comporte 6 voies (8 voies sur la Mini et la Nano) connectées à un convertisseur analogique-numérique 10 bits. Cela signifie qu'il est possible de transformer la tension d'entrée entre 0 et 5V en une valeur numérique entière comprise entre 0 et 1023. Il en résulte une résolution (écart entre 2 mesures) de : 5 volts / 1024 intervalles, autrement dit une précision de 0.0049 volts (4.9 mV) par intervalle !



Fonction	Description
	Une conversion analogique-numérique dure environ 100 µs (100 microsecondes soit 0.0001 seconde) pour convertir l'entrée analogique, et donc la fréquence maximale de conversion est environ de 10 000 fois par seconde.
analogWrite(broche, valeur) broche: la broche utilisée pour "écrire" l'impulsion. Cette broche devra être une broche ayant la fonction PWM, Par exemple, sur la UNO, ce pourra être une des broches 3, 5, 6 ,9, 10 ou 11. valeur: la largeur du "duty cycle" (proportion de l'onde carrée qui est au niveau HAUT) : entre 0 (0% HAUT donc toujours au niveau BAS) et 255 (100% HAUT donc toujours au niveau HAUT). Pulse Width Modulation 0% Duty Cycle – analogWrite(0) 5v 0v 25% Duty Cycle – analogWrite(64) 5v 0v 50% Duty Cycle – analogWrite(127) 5v 0v	Génère une impulsion de largeur / période voulue sur une broche de la carte Arduino (onde PWM - Pulse Width Modulation en anglais ou MLI - Modulation de Largeur d'Impulsion en français). Ceci peut-être utilisé pour faire briller une LED avec une luminosité variable ou contrôler un moteur à des vitesses variables. Pour plus de détails sur les impulsions PWM voir : Après avoir appelé l'instruction analogWrite(), la broche générera une onde carrée stable avec un "duty cycle" (fraction de la période où la broche est au niveau haut) de longueur spécifiée (en %), jusqu'à l'appel suivant de l'instruction analogWrite() (ou bien encore l'appel d'une instruction digitalRead() ou digitalWrite() sur la même broche). La fréquence de l'onde PWM est approximativement de 490 Hz (soit 490 périodes par seconde). Pulse Width Modulation 75% Duty Cycle – analogWrite(191) 5v 0v 100% Duty Cycle – analogWrite(255) 5v 0v
delay (ms) **ms** (unsigned long): le nombre de millisecondes que dure la pause	Réalise une pause dans l'exécution du programme pour la durée (en millisecondes) indiquée en paramètre.
variable_unsigned_long=millis() **Valeur retournée:** Le nombre de millisecondes depuis que le programme courant a démarré. Renvoie une variable long non signée.	Renvoie le nombre de millisecondes depuis que la carte Arduino a commencé à exécuter le programme courant. Ce nombre débordera (càd sera remis à zéro) après 50 jours approximativement.
min(x,y) **x:** le premier nombre, n'importe quel type de donnée **y**: le second nombre, n'importe quel type de donnée	Détermine le plus petit de deux nombres.
sin(rad) **rad:** l'angle en radians (float)	Calcule le sinus d'un angle (en radians). Le résultat sera entre -1 et 1.
cos(rad) **tan(rad)** **degrees(rad)** **radians(deg)** **PI** **unsigned long micros()** **delayMicroseconds(us)**	**max(x, y)** **abs(x)** **constrain(x, a, b)** **map(valeur, toLow, fromHigh, toLow, toHigh)** **pow(base, exposant)** **sq(x)** **sqrt(x)**

D-Exemples d'implémentation de commandes numériques en temps réel à l'aide d'Arduino Uno

Les exemples suivants sont implémentés, en temps réel à l'aide d'Arduino Uno, sous environnement du logiciel TINKERCAD en ligne (site : tinkercad.com) après enregistrement sur ce site web.

Remarque: Le logiciel de simulation est **TINKERCAD** en ligne (site: tinkercad.com) après enregistrement sur ce site web. Si vous trouvez des difficultés, essayez de voir des vidéos soit en arabe ou en français ou en anglais. Un de ces vidéos est: https://www.youtube.com/watch?v=5m2aTJ9gnYE&t=544s

D.1- Commande numérique du clignotement d'une LED (Génération d'un signal carré):

```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an otput.
// LED_BUILTIN ; c'est la broche 13 de l'Arduino
  pinMode(LED_BUILTIN, OUTPUT);
}

void loop() {     // the loop function runs over and over again forever
  digitalWrite(LED_BUILTIN, HIGH);   // on allume la LED
  delay(1000);                       // on attend une seconde
  digitalWrite(LED_BUILTIN, LOW);    // on attend une seconde
  delay(1000);                       // on attend une seconde
}
```

Programme Arduino

```
void setup() {
    pinMode(LED_BUILTIN, OUTPUT);
              }

void loop() {
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
               }
```

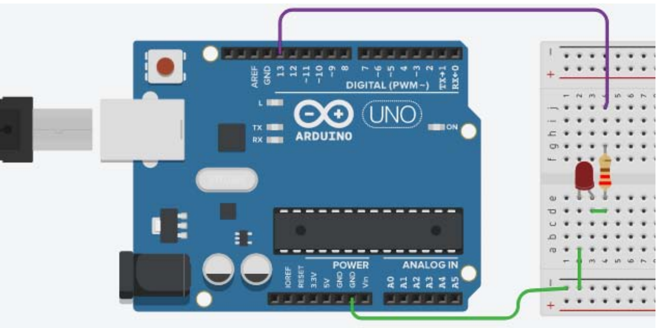
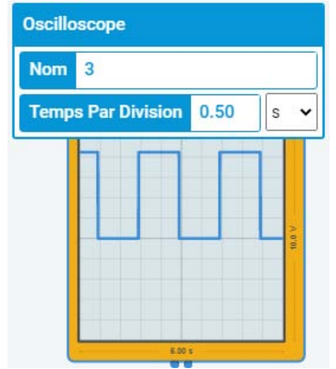


Schéma du montage



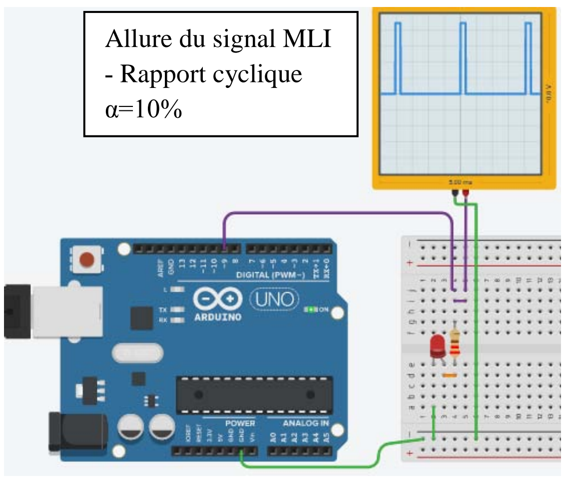
Allure du signal de sortie (500ms/div)

D.2- Commande numérique de la luminosité d'une LED (Génération d'un signal MLI):

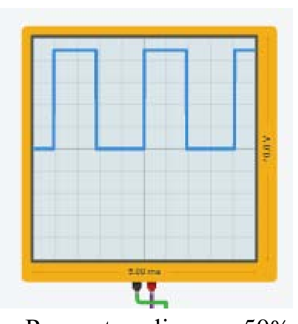
```
int ledPin = 9;    // LED connected to digital pin 9

void setup() {// nothing happens in setup
 }

void loop() {
  // fade in from min to max in increments of 5 points:
  for (int fadeValue = 0 ; fadeValue <= 255; fadeValue += 5) {
    // sets the value (range from 0 to 255):
    analogWrite(ledPin, fadeValue);
    // wait for 30 milliseconds to see the dimming effect
    delay(30);
  }

  // fade out from max to min in increments of 5 points:
  for (int fadeValue = 255 ; fadeValue >= 0; fadeValue -= 5) {
    // sets the value (range from 0 to 255):
    analogWrite(ledPin, fadeValue);
    // wait for 30 milliseconds to see the dimming effect
    delay(30);
  }
}
```

```
int ledPin = 9;

void setup() {
           }

void loop() {
  for (int fadeValue = 0 ; fadeValue <= 255; fadeValue += 5) {
        analogWrite(ledPin, fadeValue);
        delay(30);
            }

   for (int fadeValue = 255 ; fadeValue >= 0; fadeValue -= 5) {
        analogWrite(ledPin, fadeValue);
        delay(30);
                                                             }
              }
```

Programme Arduino



Allure du signal MLI - Rapport cyclique α=10%



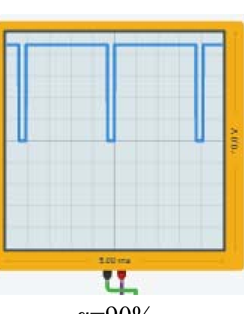


Schéma du montage (500ms/div) et allures du signal MLI - Rapport cyclique α=50% α=90%

D.3- Comptage & Impression du temps d'exécution sur le Moniteur Série:

```
unsigned long time;

void setup(){
 Serial.begin(9600);
}
void loop(){
 Serial.print("Time: ");
 time = millis();
 //prints time since program started
 Serial.println(time);
 // wait a second so as not to send massive amounts of data
 delay(1000);
}
```

```
1  unsigned long time;
2
3  void setup(){
4    Serial.begin(9600);
5  }
6  void loop(){
7    Serial.print("Time: ");
8    time = millis();
9    Serial.println(time);
10   delay(1000);
11 }
12
```

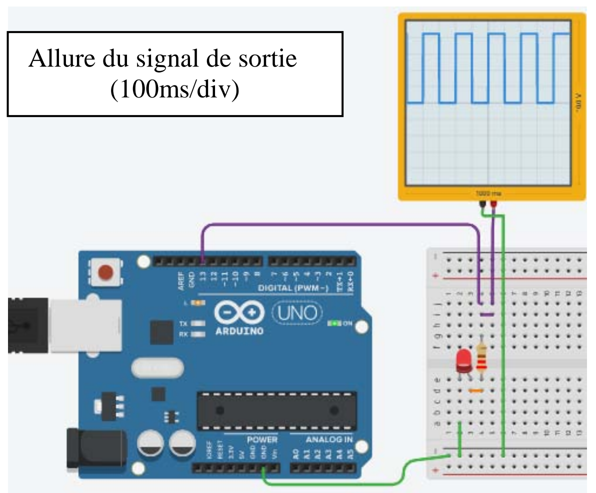
Moniteur série

```
Time: 1
Time: 1001
Time: 2002
Time: 3003
Time: 4004
Time: 5005
Time: 6006
Time: 7007
Time: 8008
Time: 9010
```

Résultat d'exécution du programme

D.4- Commande numérique sans fonction ''delay()''du clignotement d'une LED (Génération d'un signal carré): Blink Without Delay

```
// constants won't change. Used here to set a pin number:
const int ledPin =  LED_BUILTIN;// the number of the LED pin (13)

// Variables will change:
int ledState = LOW;           // ledState used to set the LED

// Generally, you should use "unsigned long" for variables that hold time
// The value will quickly become too large for an int to store
unsigned long previousMillis = 0;       // will store last time LED was updated

// constants won't change:
const long interval = 1000;          // interval at which to blink (milliseconds)

void setup() {
 // set the digital pin as output:
 pinMode(ledPin, OUTPUT);
}

void loop() {
 // here is where you'd put code that needs to be running all the time.

 // check to see if it's time to blink the LED; that is, if the difference
 // between the current time and last time you blinked the LED is bigger than
 // the interval at which you want to blink the LED.
 unsigned long currentMillis = millis();

 if (currentMillis - previousMillis >= interval) {
  // save the last time you blinked the LED
  previousMillis = currentMillis;

  // if the LED is off turn it on and vice-versa:
  if (ledState == LOW) {
   ledState = HIGH;
  } else {
   ledState = LOW;
  }

  // set the LED with the ledState of the variable:
  digitalWrite(ledPin, ledState);
 }
}
```




```
const int ledPin =  LED_BUILTIN;
int ledState = LOW;
unsigned long previousMillis = 0;
unsigned long currentMillis=0;
const long interval = 100;

void setup() {
    pinMode(ledPin, OUTPUT);
            }

void loop() {
    currentMillis = millis();

  if (currentMillis - previousMillis >= interval) {
      previousMillis = currentMillis;

    if (ledState == LOW) { ledState = HIGH; }
    else { ledState = LOW; }

   digitalWrite(ledPin, ledState);
                                                    }
            }
```

Schéma du montage

Programme Arduino

D.5- Commande numérique d'un moteur à courant continu en marche et arrêt

```
      // the setup function runs once when you press reset or power the board
void setup() {
      // initialize digital pin LED_BUILTIN as an otput.
      // LED_BUILTIN ; c'est la broche 13 de l'Arduino
  pinMode(LED_BUILTIN, OUTPUT);
}

      // the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);   // on allume la LED
  delay(3000);                       // on attend une seconde
  digitalWrite(LED_BUILTIN, LOW);    // on attend une seconde
  delay(3000);                       // on attend une seconde
            }
```

Programme Arduino

```
int Dt=3000;

void setup() { pinMode(LED_BUILTIN, OUTPUT); }

void loop() {
  digitalWrite(LED_BUILTIN, HIGH);     delay(Dt);
  digitalWrite(LED_BUILTIN, LOW);      delay(Dt);
            }
```

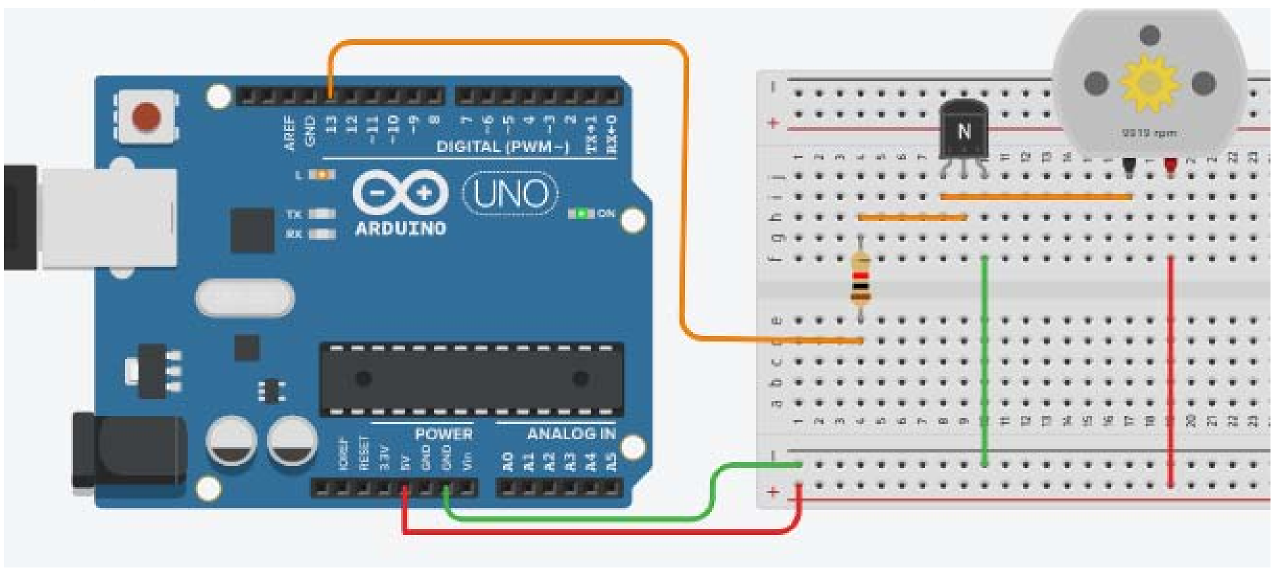


Schéma du montage

E-Instructions du Langage Arduino

E.1 - Les types

Pour programmer, nous allons utiliser plusieurs sortes de données. Par exemple, si nous voulons incrémenter un compteur, nous utiliserons une variable de type entier. Si nous voulons calculer la moyenne d'une série de données fournies par un capteur, nous aurons besoin d'un nombre réel. S'il s'agit de tester une condition, nous pourrions avoir besoin d'un simple booléen. Voici un rappel des principaux types:

- **void**: c'est un mot clef utilisé pour spécifier qu'il n'y a **pas de variable**, on le retrouve généralement dans le prototype d'une fonction pour indiquer que celle-ci ne renvoie pas de paramètre (ou qu'elle n'en prend pas)
- **boolean**: c'est une variable logique qui ne peut prendre que **deux états: false, true**. Elle sert essentiellement à **tester des conditions**, des **états binaires**.
- **char**: traditionnellement, le **char** est utilisé pour les caractères. En fait, il s'agit d'une variable codée sur **un octet**, qui peut donc prendre **256 valeurs** (2^8) différentes. Les caractères ayant longtemps été indexés sur un table à 256 entrées, on a tendance à ne réserver l'usage du type **char** qu'aux seuls caractères. Mais il peut de fait être utilisé pour toute variable qui n'a pas besoin d'excéder l'**intervalle [-128; 127]**.
- **unsigned char**: le mot clef **unsigned** signifie que l'on considère des **valeurs positives** uniquement. Ainsi, si un **char** permet de représenter des valeurs allant de -128 à +127, le type **unsigned char** utilise quant à lui l'intervalle **[0, 255]**.
- **byte**: le **byte** est un synonyme du **unsigned char**, c'est-à-dire qu'il se code sur **un seul octet** et prend des valeurs pouvant aller **de 0 à 255**. Il est particulièrement **adapté pour les sorties PWM**.
- **int**: un **int** (pour **integer**) est un **entier relatif** codé sur **deux octets**, ce qui signifie qu'il possède **2^16 valeurs différentes**. Il peut prendre des valeurs comprises dans l'**intervalle [-32,768; 32,767]**.
- **unsigned int**: c'est un **entier positif** codé sur **deux octets**, prenant ses valeurs dans l'**intervalle [0; 65,535]**.
- **word**: synonyme de **unsigned int**.
- **long**: **entier relatif** codé sur **4 octets**, pouvant donc prendre des valeurs allant de **-2,147,483,648 à 2,147,483,647**.
- **unsigned long**: **entier positif** codé sur **4 octets**, prenant dès lors ses valeurs dans l'**intervalle [0; 4,294,967,295]**.
- **float**: il s'agit d'une variable de **type réel**, codée sur **4 octets**. Sa précision n'excède pas **6 ou 7 décimales**.
- **double**: traditionnellement, le type **double** offre une précision deux fois supérieure à celle du **float**. Dans le cas du microcontrôleur Arduino, les deux types sont synonymes: **double = float**.
- **String**: le type **String** est le premier type complexe que nous rencontrons. Ce n'est d'ailleurs plus vraiment un type, mais un **Objet**, au sens de la **POO**(programmation object oriented). Pour faire simple dans un premier temps, considérons qu'il s'agit d'une **chaîne de caractères**, que l'on peut donc y stocker des mots, des phrases, ... Le type **String** vient avec plusieurs **méthodes** qui permettent de **traiter, comparer, manipuler**, ... les chaînes de caractères.
- **array**: ce n'est pas un type à proprement parler. Il est possible de déclarer **pour chaque type** un **tableau de données** de ce type, un tableau d'entiers par exemple pour stocker des états, ou un tableau de booléens, pour caractériser une série de données. On utilisera pour cela les **crochets [ ]** placés après le nom de la variable.

Ainsi, lorsque l'on souhaitera créer une variable, on commencera par spécifier son type, puis son nom, et éventuellement on lui assignera une valeur qu'il sera ensuite possible de modifier.

Exemples:

int age = 13; // on créé un entier contenant la valeur 13

String message = "hello world"; // on créé une chaîne de caractères contenant la valeur "hello world"

char key = 'c'; // on créé une variable de type char contenant le caractère 'c', ou plus exactement la position du caractère 'c' dans le tableau indexant les principaux caractères

int tabValeurs [1024]; // on créé un tableau pouvant contenir 1024 entiers. Les valeurs ne sont pas initialisées, elles peuvent donc contenir absolument n'importe quoi dans un premier temps.

Pour **accéder à une valeur stockée dans un tableau**, on utilisera l'**index de sa position** que l'on indiquera entre les crochets, tout en faisant attention au fait que **l'indexation commence à 0**. Ainsi, pour un tableau de 1024 éléments, le premier éléments sera accessible en appelant tabValeurs[0] et le dernier en invoquant tabValeurs[1023].

Pourquoi tous ces types ? N'est-il pas plus simple de ne déclarer que des long, voire uniquement des floats ? L'intérêt réside dans l'**utilisation de la mémoire**. La mémoire d'un microcontrôleur est fortement limitée (32 ko pour l'Arduino). Créer un tableau de 1024 entiers prend 2048 octets, soit 2 ko, soit 1/16 de la mémoire disponible. Si l'on sait par avance que les valeurs ne seront jamais supérieures à 255 (et toujours positives), par exemple dans le stockage de sorties analogiques, on pourrait utiliser à la place des **bytes** et gagner ainsi la moitié de l'espace.

Supposons par exemple que l'on veuille stocker **une image couleur de 80x60 pixels**. On aura un tableau contenant 200x200x3 = **14400 entrées**. Si on les déclare **comme des int, l'espace mémoire utilisé est 28800 ko**, soit la quasi-totalité des ressources du micro-contrôleur. Autant dire que l'on ne pourra pas faire de traitement vraiment intéressant de

cette image. Maintenant, comme on sait que les **codes couleurs sont codés sur un seul octet**, on peut décider d'utiliser des **bytes** plutôt que des **int**. On n'utilise alors que **14 ko**, soit moins de la moitié de la mémoire totale, ce qui laisse de la place pour un traitement de l'image.

E.2 - Les fonctions

Une fonction possède un nom, des paramètres d'entrée et des paramètres de sortie. On appelle prototype d'une fonction la spécification de ces trois données. Par exemple, on peut vouloir créer une fonction qui prend deux entiers en paramètre, puis renvoie leur produit. Son prototype sera:

```
int multiplication(int, int)
```

Si la fonction **ne renvoie aucune donnée**, son type de retour sera **void**. Si elle ne prend **aucun paramètre**, on pourra soit ne rien mettre entre les parenthèses, soit écrire également **void**.
Si la fonction renvoie une donnée, elle devra le faire avec le mot-clef **return**.
Enfin, tout ce que fait la fonction doit être placé entre deux accolades.
Exemples:

```
int multiplication (int a, int b)
{
    int c;
    c = a * b;
    return c;
}
```

que l'on pourra écrire plus succinctement:

```
int multiplication (int a, int b)
{
    return a * b;
}
```

Ainsi, à chaque fois que nous aurons à exécuter une multiplication, il suffira d'appeler la fonction que nous venons de définir:

```
void quelques_calculs()
{
    int a = 7;
    int b = 8;
    int c = muliplication(a, b);
}
```

E.3 - Les structures de condition et d'itération

Les structures principales sont grosse merdo de deux sortes: structure conditionnelles et structures itératives. Les principales sont les suivantes:

a - Structure conditionnelle IF ... ELSE

La structure conditionnelle **IF ... ELSE** permet de vérifier une condition logique. Elle repose donc sur l'évaluation d'une expression simple ou complexe, à laquelle elle donne la valeur Vrai (**TRUE** ou **1**) ou la valeur Faux (**FALS**E, ou **0**).
Par exemple, on veut vérifier sur la valeur donnée par un capteur de distance est supérieure à un certain seuil. On écrira:

```
int seuil = 80;
int valeur = analogRead(capteur_distance);
if (valeur > seuil)
{
// traitement si la valeur est strictement supérieure
}
else
{
// traitement si la valeur est inférieure ou égale
}
```

Ici, si la valeur lue est supérieure au seuil, le code situé entre les deux premières accolades sera exécuté, mais pas celui entre les deux accolades suivantes.
Si la valeur est inférieure - ou égale -, alors c'est l'inverse. Le code situé entre les deux premières accolades ne sera pas exécuté, mais celui placé entre les deux suivantes le sera.

Il est possible d'utiliser les opérateurs logiques pour évaluer plusieurs conditions en même temps. On utilisera l'opérateur de négation **!** pour avoir l'inverse d'une valeur (si **P** est Vrai, **!P** est faux), l'opérateur de conjonction **&&** qui renverra Vrai uniquement si les deux conditions qu'il rassemble sont vraies ensembles, l'opérateur de disjonction ||qui renverra Vrai si et seulement si une des valeurs est Vraie, et enfin l'opérateur de disjonction exclusive **^** qui renverra Vrai si une seule des conditions est vrai et l'autre fausse.

Exemple:

```
int a = 80;  int b = 30;  int c = 2;
( ( (b == c) || ! (b > a) ) ^ (a < c) )
```

renverra Vrai.

En effet;

- l'expression **(b == c)** est fausse, puisque b est différent de c
- l'expression **(b > a)** est fausse
- donc l'expression **!(b > a)** est vraie
- par conséquent, **((b == c) || ! (b > a))** est vraie, puisque l'une des deux expressions est vraie
- Ensuite, **(a < c)** est fausse.
- En conclusion, comme **((b == c) || ! (b > a))** est vraie et **(a < c)** est fausse, leur disjonction exclusive est vraie. L'expression est donc vraie.

On peut bien-sûr enchaîner séquentiellement les évaluations conditionnelles:

```
int a = 80;  int b = 30;  int c = 2;
if ( a > b )
{
// traitement
}
else
if ( a > c)
{
// traitement
}
else
{
// traitement
}
```

Notons que le **ELSE** est facultatif.

Enfin, la condition du **IF ... ELSE** peut-être un simple booléen, comme dans l'exemple suivant:

```
boolean estActif = true;
... // traitements qui peuvent modifier la valeur de estActif
if ( estActif )
{
// traitement
}
```

b - Structure de branchement SWITCH ... CASE ... DEFAULT

Le **SWITCH** peut être imaginé comme un carrefour. On arrive à un moment du code où l'on peut prendre plusieurs directions. On suppose par exemple que l'on cherche à savoir parmi quatre boutons-poussoirs lequel a été actionné. On a pour cela une variable qui contient le numéro du bouton-poussoir que l'on recherche, valeur que l'on examine et en fonction de laquelle on exécute différents traitements.

Exemple:

```
byte boutonActif = 0;
... // traitement qui permet de savoir sur quel bouton on a appuyé, contient 0 si aucun bouton n'a été utilisé
switch ( boutonActif )
{
case 1:
// traitement si c'est le bouton 1
break;
case 2:
```

```
// traitement si c'est le bouton 2
break;
case 3:
// traitement si c'est le bouton 3
break;

case 4:
// traitement si c'est le bouton 4
break;

default:
// traitement si aucun bouton n'a été utilisé
break;

}
```

La limitation principale du **SWITCH ... CASE ... DEFAULT** est qu'il opère de façon discrète (par opposition à continue) et sur des expressions simples. On ne peut par exemple pas avoir un **case a < b**, ou un **case a && b**.
Le mot-clef **BREAK** signifie que l'on sort du **SWITCH** à la fin du traitement. Il faut en effet savoir que chaque case correspond à une entrée, mais qu'une fois dans le **SWITCH**, on est censé parcourir toutes les valeurs qui suivent jusqu'à la dernière. On peut le voir comme un toboggan à plusieurs entrées. On choisit à quelle hauteur on arrive, mais une fois dans le toboggan, on doit parcourir toute la distance, toutes les hauteurs, et cela jusqu'en bas. Le **BREAK** permet d'avoir des points de sortie. Traditionnellement, il vient conclure chaque **CASE**, mais on peut imaginer évidemment des situations plus compliquées. Par exemple, on veut savoir quel actionneur d'un robot doit être activé. On veut que si les valeurs des actionneurs **1**, **3**, ou **5** ont a être modifiées, les actionneurs qui leur succèdent doivent s'adapter également (respectivement (**2**, **4** et **6**). En revanche, si on décide de modifier les valeurs de **2**, **4** ou **6** (pour une correction suite à une information donnée par le retour d'un capteur), on ne veut pas modifier les actionneurs **1**, **3** et **5**. On aura alors:

```
switch ( valeur_indiquant_l_actionneur_a_modifier )
{
case 1:
// traitement si c'est l'actionneur 1
case 2:
// traitement si c'est l'actionneur 2
break;
case 3:
// traitement si c'est l'actionneur 3
case 4:
// traitement si c'est l'actionneur 4
break;
case 5:
// traitement si c'est l'actionneur 5
case 6:
// traitement si c'est l'actionneur 6
break;
default:
break;
}
```

Si on modifie l'actionneur **1**, on passe ensuite au second qui est lui aussi modifié, puis on sort du **SWITCH**. Si par contre on commence par modifier l'actionneur **2**, on sort immédiatement après et rien d'autre n'est modifié.
Notons enfin que le mot-clef **DEFAULT** qui vient conclure un **SWITCH** est utilisé comme traitement par défaut. Dans le cas par exemple où la valeur considérée correspondrait à une touche de clavier, dont seules les pressions sur les flèches nous intéresseraient, toute autre pression de touche sera renvoyée sur **DEFAULT**. On peut imaginer que ce n'est pas obligatoire, mais certains compilateurs vous hurleront aux oreilles si vous n'envisagez pas tous les cas. Il est donc toujours préférable de le mettre, même s'il reste vide. Le **BREAK** qui le suit est évidemment inutile, mais on peut le

laisser par souci d'homogénéité du code (et rien n'oblige à finir par le traitement par default. Il peut très bien figurer en début de branchement).

c - Structure d'itération FOR (initialisation; condition de traitement; incrémentation)

Il est parfois utile d'exécuter un traitement un grand nombre de fois. On veut par exemple parcourir un tableau - pour en faire la moyenne -, ou faire clignoter un nombre défini de fois une LED, ou encore faire avancer tourner une tourelle sur laquelle est fixé un capteur. On utilisera pour cela une boucle FOR. Dans sa forme la plus simple, un FOR utilise un compteur que l'on incrémente.

Exemple:

```
int tab[128]; // définition d'un tableau d'entiers contenant 128 cases, indexées de 0 à 127
... // traitement pour remplir le tableau avec des valeurs provenant d'un capteur
int somme = 0;
for (int i = 0; i < 128; i++)
{
somme += tab[i];
}
float moyenne = somme / 128.0;
```

On commence donc par définir un tableau de 128 cases. On remplit ensuite ce tableau avec des valeurs issues par exemple d'un capteur. On initialise ensuite une variable entière à 0, la variable **somme**.
La boucle **FOR** nous permet ensuite de parcourir le tableau. A chaque tour de boucle, la valeur contenue dans la case du tableau correspond à l'index donné par la variable i est ajoutée à la variable somme.
Une fois le tableau parcouru, on créé une variable de type **DOUBLE**, et l'on divise la somme par le nombre d'éléments pour obtenir la moyenne.
Plusieurs remarques déjà d'ordre syntaxique:

- l'écriture **i++** est un raccourci pour **i = i +1**. C'est-à-dire qu'à chaque tour de boucle, on augmente la valeur de i d'une unité
- l'écriture **somme += tab[i]** est un raccourci pour **somme = somme + tab[i]**. A chaque tour de boucle, somme ajoute à sa valeur précédente la valeur contenue dans la case du tableau sur laquelle on pointe.
- enfin, on écrit **128.0** pour en faire un **float** et non un **entier**. En effet, si la division est effectuée entre deux entiers, elle agira comme une division entière. Ainsi, **5/2 = 2**, **3/8 = 0**, et **1000/128 = 7**. En forçant l'une des deux valeurs à être de type **float**, on a alors une division au sens usuel, et son résultat sera un nombre réel, de type **float**.

Au sein de la boucle FOR, on commence donc par initialiser un compteur à 0. On spécifie que l'on continue à rentrer dans la boucle tant que sa valeur est inférieure à 128 (on va donc jusqu'à la case 127). Puis, à chaque tour, on incrémente la valeur de 1.
Supposons que l'on souhaite faire exécuter à un servo-moteur une rotation de 180 degrés, avec un relevé d'un capteur tous les 5 degrés. On écrira alors:

```
Servo s; // on créé une variable de type Servo (ce que l'on verra plus tard)
s.attach(9); // on spécifie que la communication avec le Servo se fair sur le connecteur 9 de l'Arduino
... // quelques traitements
for (byte degre = 0; degre <= 180; degre += 5)
{
s.write(degre); // on indique en degre la position que doit atteindre le Servo
lireCapteur(); // on exécute une fonction qui lit un capteur
... // traitement, par exemple un envoi de données en série si la valeur du capteur dépasse un seuil
}
```

Ici, dans la boucle, on initialise la valeur de **degre** à 0. Dans le premier tour de boucle, le Servo se dirige vers la position 0 qui se trouve être le point le plus à gauche. A la fin de la boucle, on augmente la valeur de **degre** de 5. Puis on augmente à nouveau de 5 au tour suivant, etc etc etc ... A chaque tour, le Servo se déplace donc de 5 degrés, jusqu'à ce qu'il atteigne 180 degrés. A la fin de ce dernier tour, la variable est incrémentée de 5, elle vaut donc 185, et la condition de traitement n'est plus respectée. On sort donc de la boucle.

d - Structure d'itération WHILE

Si le **WHILE** et le **FOR** sont interchangeables, ils sont utilisés dans des cadres différents. Lorsqu'on utilise un **FOR**, on suppose que l'on connaît à l'avance le nombre d'itérations. Celui-ci est fixé en dehors de la boucle **FOR**. Pour un

WHILE, la condition d'arrêt dépend d'un événement qui inervient durant le processus de bouclage. On attend par exemple une entrée au clavier, ou un signal Bluetooth, ou par exemple une valeur située dans un intervalle. Quand un **WHILE** démarre, **on ne sait pas à l'avance quand - et si - il va s'arrêter**.
Pour rappel, on a dans la structure du **FO**R une initialisation, une condition de déroulement, et une incrémentation. Dans le cas du **WHILE**, l'initialisation se fait avant la boucle. La condition de déroulement se fait à l'appel du **WHILE**, et la modification des variables impliquées dans la condition se fait au sein de la boucle. Elle dépend de l'environnement (environnement physique, utilisateur, ...), et n'est pas fixée par le programme. On peut dire que le **FOR** est une boucle aux conditions immanentes, tandis que le **WHILE** est une boucle aux conditions transcendantes.
D'un point de vue syntaxique, la WHILE est défini ainsi:

```
... // initialisation des variables de condition de traitement
while ( condition )
{
// différents traitements
// réactualisation de la condition
// différents traitements
}
```

On a donc pour le **FOR**:

```
FOR (initialisation, condition, actualisation)
{
// traitement
}
```

et pour le **WHILE**

```
// initialisation
WHILE ( condition )
{
// traitement
// actualisation
// traitement
}
```

En résumé, s'il est possible de transformer un **FOR** en **WHILE** et inversement, on préfèrera un **FOR** pour une série d'instruction dont on maîtrise les conditions de sortie, et un **WHILE** lorsqu'il s'agit d'une intervention contingente au programme.

E.4 - Les directives de préprocesseur

Les directives de préprocesseur sont généralement placées en début de programme. Les plus utilisées sont de deux sortes: les inclusions et les définitions.
Une inclusion sert à utiliser des bouts de programmes contenus dans un autre fichier. C'est le cas par exemple lorsque l'on appelle une **bibliothèque** (ou **librairie**). Lorsque l'on a utilisé plus haut une variable de type **Servo**, c'est parce que nous avons inclus une bibliothèque qui définit ce type d'objets. Si l'on veut utiliser des fonctions de mathématiques avancées, on utilisera une bibliothèque dédiée.
L'inclusion se fait à l'aide de la directive **#include < nom_du_fichier >** ou **#include " nom_du_fichier "**. Dans le premier cas (chevrons), la bibliothèque se situe à un endroit connu par le programme, défini pour contenir l'ensemble des bibliothèques. Dans le second cas (guillements), le fichier importé se situe dans le même répertoire que le fichier qui l'appelle, ou à une position définie à partir de lui.
Ainsi, si l'on écrit:

```
#include < mes_librairies/ma_librairie.h>
```

on ira chercher les données contenues dans un fichier **ma_librairie.h** se situant dans le répertoire **mes_librairies** qui lui-même se trouve dans le répertoire dans lequel sont usuellement placées les librairies.
En écrivant:

```
#include "mes_librairies/ma_librairie.h"
```

on se place au niveau du répertoire contenant le fichier dans lequel on écrit, on va dans un de ses sous-répertoires nommé **mes_librairies**, et on importe le fichier **ma_librairie.h** qui s'y trouve.
La seconde directive la plus utilisée est **#define**. Elle sert principalement à remplacer une valeur par un mot. Lors de la compilation, partout où ce mot aura été utilisé, le compilateur placera la valeur. C'est utile par exemple pour définir la hauteur et la largeur d'une image. On écrira:

```
#define HAUTEUR 600
#define LARGEUR 800
```

Dans le code, on pourra utiliser **HAUTEUR** et **LARGEUR**, ce qui permettra de le rendre plus clair. A la compilation, ces valeurs seront remplacées par **600** et **800** respectivement. Elles ne peuvent être modifiées durant l'écriture ou l'exécution du programme.

E.5 - Les variables globales

Les variables globales sont placées également en début de programme, et sont connues par toutes les fonctions. De manière générale, une variable définie dans une fonction n'existe qu'au sein de celle-ci. Aucune autre fonction ne la connaît, à moins qu'elle lui soit passée en paramètre.
Avec les variables globales, on peut partager des données. Ainsi, si plusieurs fonctions lisent un capteur, ou l'actualisent, on définira ce capteur en dehors des fonctions, ce qui évitera d'avoir à en recréer une instance à chaque nouvelle fonction appelée, et de le transmettre à chaque fois en paramètre.
Un bon code trouve l'équilibre entre variables globales et variables locales. Les variables globales seront généralement des variables du système physique (la valeur d'un potentiomètre par exemple), tandis que les variables locales pourront être des compteurs de parcours, des valeurs de stockage pour des calculs intermédiaires, ...

E.6- Les commentaires

Dans tout code, il est fondamental de glisser des commentaires qui peuvent être utiles soit lorsqu'on le reprend après un long moment, soit lorsque l'on diffuse notre code.
Les commentaires ne sont pas évalués par le compilateur, ils ne s'adressent qu'au développeur.
En C et en C++, les commentaires sur une seule ligne commencent par **//** et ceux placés sur plusieurs lignes sont encadrés par les signes **/*** et ***/**.