

## Minimum vital Arduino™

★ ★ ★

## Programmer la carte via l'IDE

## Table des matières

<b>1 Généralités</b>	<b>1</b>
1.1 Parties essentielles du programme . . . . .	1
1.2 Écrire des commentaires . . . . .	2
1.3 Structure . . . . .	2
1.4 Type des données . . . . .	2
<b>2 Entrées / Sorties Numériques</b>	<b>3</b>
2.1 la fonction <code>pinMode()</code> . . . . .	3
2.2 La fonction <code>digitalWrite()</code> . . . . .	3
2.3 La fonction <code>digitalRead()</code> . . . . .	3
<b>3 Communiquer via le port série</b>	<b>4</b>
3.1 Vitesse de communication : <code>Serial.begin()</code> . . . . .	4
3.2 La carte envoie des informations via le port série : <code>Serial.print()</code> . . . . .	4
<b>4 Entrées analogiques : La fonction <code>analogRead()</code></b>	<b>5</b>
<b>5 Gestion du temps</b>	<b>5</b>
5.1 Temps de pause : fonctions <code>delay()</code> et <code>delayMicroseconds()</code> . . . . .	5
5.2 Temps écoulé depuis le début : fonctions <code>millis()</code> et <code>micros()</code> . . . . .	5

Il est nécessaire de connaître les principales fonctionnalités de l'IDE Arduino, consultables ici, à savoir le bouton pour vérifier le code, le bouton pour téléverser sur la carte, le bouton pour accéder au moniteur série.

## 1 Généralités

## 1.1 Parties essentielles du programme

La phase "setup" (en fait une fonction) est une phase de réglages :

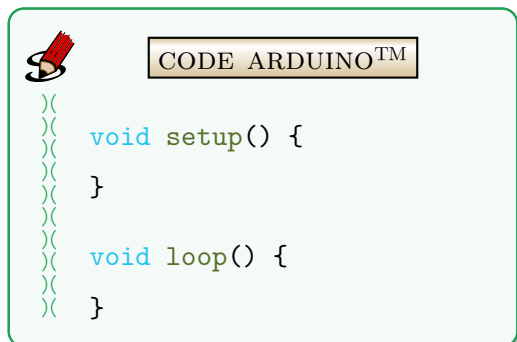
- Elle n'est lue qu'une fois par le compilateur
- Elle permet de définir entrées et sorties
- Elle permet de définir la vitesse de communication du port série

La phase "loop" est une boucle infinie (équivalente à `while True` : en Python) :

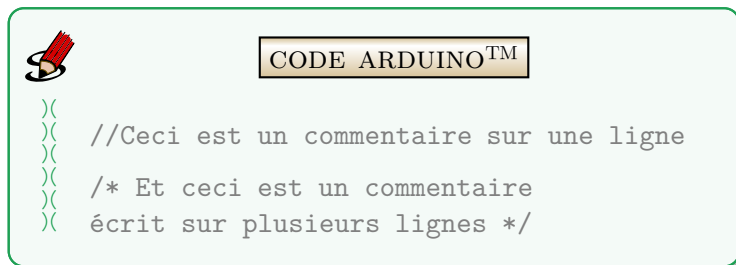
- Elle est lue jusqu'à arrêt de l'alimentation du microcontrôleur
- C'est généralement dans cette phase qu'on place l'essentiel du code

Ces deux phases sont en fait des fonctions. Les variables définies à l'intérieur ne sont valables que pour cette fonction (on parle de variables locales).

On définit des variables pour tout le programme (on parle de variable globale), en les plaçant généralement au début du programme (voir plus loin).

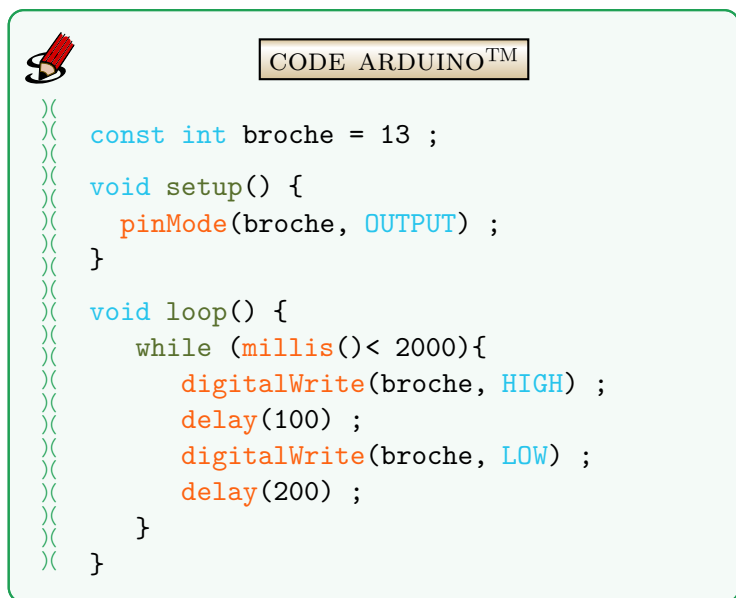


## 1.2 Écrire des commentaires



Comme en Python, on peut écrire des commentaires pour détailler le code sur une ligne ou sur plusieurs lignes.

## 1.3 Structure



Trois choses à retenir :

- On définit d'abord les variables globales, valables pour tout le programme. Il faudra définir le **type** de nos variables (voir plus loin) : **const** pour constante, **int** pour entier.  
Ici, la variable a pour nom **broche**, c'est une constante (**const**) de type entier (**int**) qui a pour valeur 13 ; elle sert à identifier la broche 13 sur la carte.
- Les instructions se terminent par des points-virgules ";" (il n'y en n'avait pas en Python)
- Chaque bloc (fonction, boucles, conditions...) commence par "{" et termine par "}" (en Python, c'étaient les tabulations qui jouaient ce rôle)
- Donc il est inutile de mettre des tabulations, mais on peut tout de même pour clarifier le programme

*Les fonctions et commandes ci-dessus seront vues en détail dans la suite*

## 1.4 Type des données

Pour Arduino, contrairement à Python, il faut définir dès leur création le **type** des variables.

Le type influe sur la valeur maximale de la variable et sur la place occupée en mémoire ; il faudra donc faire un arbitrage sur des programmes lourds.

Les principaux types qu'on pourra utiliser sont :

Type	Valeurs max et min	Taille en mémoire
boolean	0 ou 1	1 octet
byte	0 à 255 ( $2^8 - 1$ )	1 octet
int	-32 768 à 32 767	2 octets
unsigned int	0 à 65 535 ( $2^{2 \times 8} - 1$ )	2 octets
long	-2 147 483 648 à 2 147 483 647	4 octets
unsigned long	0 à 4 294 967 295 ( $2^{4 \times 8} - 1$ )	4 octets
float	$-3,4028235 \cdot 10^{+38}$ à $3,4028235 \cdot 10^{+38}$	4 octets

De plus le type **const** permet de définir des constantes dans le code, comme le numéro des broches.

## 2 Entrées / Sorties Numériques

### 2.1 la fonction `pinMode()`

Elle se place dans la phase `setup`. Elle permet de définir les entrées et les sorties.

EXEMPLES :


- `pinMode(7, OUTPUT)` ; : la broche numérique 7 est considérée comme une sortie
- `pinMode(3, INPUT)` ; : la broche numérique 3 est considérée comme une entrée

### 2.2 La fonction `digitalWrite()`

Elle permet de placer une broche numérique définie comme une sortie à l'état haut (5 Volts, état "1" ou "HIGH") ou à l'état bas (0 Volt, état "0" ou "LOW")

UN CODE CLASSIQUE :

On souhaite faire clignoter une LED placée en sortie de la broche 13 avec une période de 1 seconde.



CODE ARDUINO™

```

(
(
const int LED = 13 ;
(
void setup() {
(
pinMode(LED, OUTPUT) ;
(
}
(
void loop() {
(
digitalWrite(LED, HIGH) ;
(
delay(500) ;
(
digitalWrite(LED, LOW) ;
(
delay(500) ;
(
}
(
)

```

- `digitalWrite(LED, HIGH)` ; est équivalent à `digitalWrite(LED, 1)` ; : la broche numérique 13 est placée à l'état haut : elle délivrera 5 Volts.
- `digitalWrite(LED, LOW)` ; est équivalent à `digitalWrite(LED, 0)` ; : la broche numérique 13 est placée à l'état bas : elle délivrera 0 Volt.
- `delay(500)` ; permet de réaliser une pause pendant 500 ms.


[Lien vers le programme ci-contre](#)

→ TP PRODUIRE UN SON EN UTILISANT UN MICROCONTRÔLEUR

### 2.3 La fonction `digitalRead()`

Elle permet de lire l'état d'une broche numérique définie comme une entrée :

**La broche est-elle à un état haut ou à un état bas ?**



CODE ARDUINO™

```

(
(
int etat ;
(
const int broche = 8 ;
(
void setup() {
(
pinMode(broche, INPUT) ;
(
}
(
void loop() {
(
etat = digitalRead(broche) ;
(
}
(
)

```

Dans ce programme :

- On déclare une variable dont le nom est `etat` de type "entier".
- On déclare de même une constante entière dont le nom est `broche`, qui vaut 8 (elle servira pour définir la broche numérique utilisée).
- La broche 8 est définie comme une entrée
- On affecte alors à la variable `etat` (locale) la valeur numérique de l'état de la broche 8 (à savoir 1 ou 0 selon l'état de la broche).

REM. : La fonction `digitalRead` retournant 0 ou 1, elle retourne donc un booléen. On aurait donc pu définir la variable `etat` et son type par `boolean` `etat` au lieu de `int` `etat`.

### 3 Communiquer via le port série

La bibliothèque est `Serial`. Elle est importée d'office par Arduino. On peut donc utiliser les fonctions directement.

#### 3.1 Vitesse de communication : `Serial.begin()`

La carte Arduino peut communiquer avec le moniteur série de l'IDE Arduino ou un programme Python via le port série.

Mais pour cela, il faut que les deux parties soient en accord sur la vitesse de communication.

Dans le moniteur série (cliquer sur la petite loupe en haut à droite de l'IDE), la vitesse de communication est notée en bas à droite (en bauds – bits par seconde).

On peut donc choisir la vitesse de communication sur le port série dans la liste défilante.

Il faut définir la même vitesse dans le programme implanté sur la carte **dès la phase de réglages** :



CODE ARDUINO™

```
((
)) void setup() {
)) Serial.begin(9600) ; // on initialise la communication et on fixe la vitesse à 9600 bauds
)) }
((
```

REMARQUE :

Pour communiquer avec Python, il faudrait de même définir dans le programme Python la vitesse de communication (mais aussi le nom du port série).

#### 3.2 La carte envoie des informations via le port série : `Serial.print()`

On souhaite que la LED de la broche 13 soit placée à l'état haut pendant 3 secondes ; pendant ce temps, le moniteur série affiche "allumee" – ensuite, la LED passe à l'état bas pendant 2 secondes ; à ce moment, le moniteur série affiche "eteinte".



CODE ARDUINO™

```
((
)) void setup() {
)) Serial.begin(9600) ; // vitesse de communication fixée
)) pinMode(13, OUTPUT) ; // broche 13 vue comme une sortie
)) }
((
)) void loop() {
)) digitalWrite(13, HIGH) ; // on place la broche 13 à l'état haut, 5V
)) Serial.println("allumee") ; // On affiche "allumee" sur le moniteur série
)) delay(3000) ; // on attend 3 secondes
)) digitalWrite(13, LOW) ;
)) Serial.println("eteinte") ;
)) delay(2000) ;
)) }
((
```

Lien vers le programme ci-dessus

→ Modifiez le programme pour comprendre la différence entre `Serial.print()` et `Serial.println()`.

## 4 Entrées analogiques : La fonction `analogRead()`

Les entrées analogiques sont repérées par : A0, A1, ..., A5.

Elles disposent d'un CAN 10 bits qui convertit une tension d'entrée comprise entre 0 et 5 Volt en valeur numérique comprise entre 0 et 1023.



### attention

- A0, A1, ..., A5 sont forcément des ENTRÉES
- Il ne sera donc pas nécessaire de préciser `pinMode(A0, INPUT)` ;
- Attention de ne pas dépasser 5 Volts pour le signal reçu par les broches analogiques



### CODE ARDUINO™

```
(
)
void setup() {
    // on ne met rien dans le setup, A0 est forcément une entrée
}

void loop() { // boucle infinie
    int val ; // on crée une variable "val", entière ; c'est une variable locale,
    connue uniquement dans la phase loop
    val = analogRead(A0) ; // on affecte la valeur en sortie du CAN comprise entre 0
    et 1023 à la variable "val" précédemment créée.
}
```

→ TP UN BÊTE POTENTIOMÈTRE PARTIE 1

## 5 Gestion du temps

### 5.1 Temps de pause : fonctions `delay()` et `delayMicroseconds()`

- `delay(300)` ; : réalise une pause de 300 ms
- `delayMicroseconds(50)` ; : réalise une pause de 50  $\mu$ s

### 5.2 Temps écoulé depuis le début : fonctions `millis()` et `micros()`

Ces fonctions permettent de connaître le temps écoulé depuis le début du programme (allumage ou reset) en millisecondes ou en microsecondes.



### attention

- D'après leurs définitions, ces fonctions vont retourner des nombres qui très vite vont devenir très grands.
- Il est donc conseillé de placer l'appel à ces fonctions dans des variables dont le type est étendu
- Par exemple : `unsigned long temps_ecoule = millis()` ;  
 $\Rightarrow$  la variable `temps_ecoule` contient donc le nombre de millisecondes écoulées depuis le démarrage (ou redémarrage) du programme.

→ TP UN BÊTE POTENTIOMÈTRE PARTIE 2