ARDUINO

Manuel de référence

DOSSIER RESSOURCE

POUR LA CLASSE

Sommaire

Ce qui suit décrit de manière sommaire les instructions utilisables dans l'environnement de développement Arduino. La syntaxe du langage est celle du C avec éventuellement celle du C++ dans la version 0004.
Ce manuel est divisé en cinq parties :

Lexique

- **ANALOG** : Analogique.
- **AREF** : Abréviation pour Analog REFerence, référence analogique.
- **AVAILABLE** : Disponible.
- **BEGIN** : Début.
- **BIT** : bit, unité d'information informatique pouvant prendre soit la valeur 0 soit la valeur 1.
- **BUFFER** : Tampon, dans le sens de "zone tampon".
- **BYTE** : Octet, soit un groupe de 8 bits.
- **bps** : Abréviation pour Bits Per Second, Bits Par Seconde. Attention, abréviation toujours en minuscules.
- **CHAR** : Abréviation de CHARacter, caractère (typographique). Type de variable d'une taille d'un octet. C'est un synonyme de "byte" utilisé pour déclarer des variables stockant un caractère ou des chaînes de caractères.
- **DEFINE** : Définit.
- **DIGITAL** : Numérique.
- **DO** : Faire.
- **FALSE** : Faux.
- **FOR** : Pour.
- **GND** : Abréviation pour GrouND, la terre. C'est la masse, 0 Volt.
- **HIGH** : Haut.
- **ICSP** : Abréviation pour In Cicuit Serial Programing, programmation série sur circuit.
- **IF** / **THEN** / **ELSE** : Si / Alors / Sinon.
- **IN** : Souvent l'abréviation pour INput, Entrée. Est toujours en rapport avec le sens extérieur vers carte Arduino.
- **INCLUDE** : Inclut.
- **INPUT** : Entrée.
- **IS** : Est (souvent dans le sens d'une question : Est ?).
- **INT** : Abréviation pour INTeger, entier. Groupe de 16 bits, 2 octets groupés, considérés comme représentant un nombre entier négatif ou positif.
- **LONG** : Abréviation pour "entier long". Groupe de 32 bits, 4 octets groupés, considérés comme représentant un nombre entier négatif ou positif.
- **LOOP** : Boucle.
- **LOW** : Bas.
- **OUT** : Souvent l'abréviation pour OUTput, Sortie. Est toujours en rapport avec le sens carte Arduino vers extérieur.
- **OUTPUT** : Sortie.
- **PIN** : Broche.
- **POWER** : Puissance, alimentation.
- **PWM** : Abréviation de (Pulse Width Modulation), soit Modulation en Largeur d'Impulsion.
- **PWR** : Abréviation pour PoWeR, puissance, alimentation.
- **READ** : Lire.
- **RX** : Abréviation pour Receive, réception.
- **SERIA**L : Série.
- **SETUP** : Initialisation.
- **TRUE** : Vrai.
- **TX** : Abréviation Transmit, transmission (émission).
- **WHILE** : Tant que.
- **WORD** : mot, soit dans le mot de langage ; soit dans le sens d'un groupe de 16 bits, 2 octets groupés considérés comme représentant un nombre entier positif (>= 0).
- **WRITE** : Ecrire.

Présentation de la carte et mise en œuvre

Un programme Arduino nécessite deux fonctions au minimum :

* setup ()
* loop ()

La fonction setup() est la fonction d'initialisation et la fonction loop() est la fonction d'exécution.

Dans la fonction setup(), la première fonction exécutée de votre programme, vous devez initialiser la carte et le programme dans la configuration requise pour votre application. Typiquement vous utiliserez des fonctions comme pinMode, beginSerial, etc.

Dans la fonction loop() vous inscrirez le code d'exécution de votre programme : lecture des entrées, basculement des sorties, etc.

Présentation de la carte et mise en œuvre

Syntaxe du langage

* ; (point-virgule)
* { } (accolades)
* // (commentaire d'une ligne)
* /* */ (commentaire multi-lignes)
* #define (directive de définition de constante)

Structures de contrôle

Choix

* if ... else
* switch ... case

Boucles

* for
* while

Variables

Les variables sont des éléments que vous pouvez utiliser dans vos programmes pour stocker des valeurs changeantes. Par exemple la valeur d'un capteur lue sur une entrée analogique. Les variables sont typées. Selon leur type, elles seront interprétées et stockées différemment. Attention, les variables décimales et leurs opérations associées ne sont pas gérées actuellement.

Types de variable :

* char (caractère)
* int (entier)
* long (entier long)

Présentation de la carte et mise en œuvre

Constantes prédéfinies

Les constantes sont des étiquettes de valeurs prédéfinies pour le compilateur Arduino. Vous pouvez les utiliser dans tous vos programmes sans les redéfinir ou les initialiser.

Arduino prédéfini les constantes suivantes :

* HIGH (haut)
* LOW (bas)
* INPUT (entrée)
* OUTPUT (sortie)

Constantes

Vous pouvez définir vos constantes en utilisant la directive #define

Fonctions

E/S numériques

* pinMode (noBroche, mode)
* digitalWrite (noBroche, valeur)
* int digitalRead (noBroche)
* unsigned long pulseIn (noBroche, valeur)

E/S analogiques

* int analogRead (noBroche)
* analogWrite (noBroche, valeur) - PWM

Communication série

Les fonctions suivantes sont utilisées pour la communication série entre la carte Arduino et l'ordinateur, via la liaison USB ou RS232 (les deux liaisons apparaissent sous la forme d'un port série sur l'ordinateur). Vous pouvez aussi utiliser ces fonctions pour des communications série via les E/S numériques 0 (RX) et 1 (TX).

L'objet Serial n'est disponible que dans la version 0004 de l'environnement de développement :

* Serial.begin (debit)
* Serial.available ()
* Serial.read ()
* Serial.print (donnees {, type })
* Serial.println ({ donnees {, type } })

Gestion du temps

* unsigned long millis ( )
* delay (NbMilliSecondes)
* delayMicroseconds (NbMicroSecondes)

Structure minimale d'un programme Arduino

L'extrait de code qui suit, donne la structure minimale d'un programme Arduino.

```
void setup()
{
}
void loop()
{
}
```

Les variables

Les variables

Une variable est un ***espace de stockage*** nommé qui permet de stocker une valeur utilisable par la suite dans un programme. Une variable peut par exemple contenir des données lues sur un des ports analogiques ou numérique.

On affecte une valeur à une variable en utilisant le signe = (égal).

Exemple

L'extrait de code qui suit, déclare une variable maVariable, puis lui affecte la valeur de l'entrée analogique n°2.

```
int maVariable = 0; // déclare la variable ; une fois suffit
maVariable = analogRead(2); // affecte la valeur lue de l'entrée analogique n°2
```

maVariable est la variable elle-même. La première ligne déclare qu'elle contiendra un entier (int, abréviation de integer). La seconde ligne donne à maValeur la valeur lue sur l'entrée analogique n°2. Ceci permet de rendre cette valeur analogique accessible partout dans le code.

Lorsqu'une valeur à été affectée à une variable, vous pouvez tester sa valeur pour savoir si elle répond à certaines conditions ou l'utiliser directement.

Exemple

L'extrait de code qui suit attends un certain temps en fonction de la valeur de maVariable, temps qui sera au minimum de 100 milli-secondes.

```
if (maVariable < 100) // le contenu de maVariable est t'il inférieur à 100 ?
{
// si oui executer ce qui est entre accolades
  maVariable = 100;   // affecter la valeur 100 à maVariable
}
delay (maVariable);   // attendre maVariable milli-secondes
```

Cet exemple montre trois utilisations des variables :

1. il test le contenu de la variable : if (maVariable < 100)
2. il affecte une valeur à la variable : maVariable = 100;
3. il utilise la variable comme paramètre d'une fonction : delay (maVariable)

Les variables

Déclaration des variables

syntaxe : type nomVariable { = valeurInit };

Toutes les variables doivent être déclarées avant leur utilisation. Déclarer une variable signifie définir sont type et éventuellement lui affecter une valeur d'initialisation.

Exemple

Déclare la variable maVariable comme étant de type int et lui donne la valeur initiale zero.

int maVariable = 0;

Les types possibles pour une variable sont :

* char
* int
* long

Conseil : Donnez comme noms à vos variables des noms parlant en rapport avec leur utilisation. Ceci rendra votre code plus lisible et compréhensible pour vous et ceux qui pourraient être amenés à lire votre code. Evitez donc les noms trops vagues comme "valeur", "nombre", ... préférez des noms comme "anaPotentiometreVolume", "boutonStop", ...

ATTENTION : les variables ne peuvent pas prendre comme nom un mot réservé du langage Arduino comme par exemple "if".

Les variables

Types de variable

syntaxe : type nomdelavariable { = valeurInit };

Le type d'une variable sert à définir comment sera interprétée la variable associée et combien d'espace elle occupera en mémoire.

Char : Le type char définit un espace de stockage d'un octet (8 bits) permettant d'affecter à la variable associée une valeur entière comprise entre 0 à 255. Le type char, abréviation de "character" est généralement utilisé pour stocker les caractères.

Exemple

```
char sign = ' ';
```

Int : Le type int définit un espace de stockage de deux octets (16 bits) permettant d'affecter à la variable associée une valeur entièr comprise entre -32 768 à 32 767.

Exemple

```
int ledPin = 13;
```

Long : Le type long définit un espace de stockage de quatre octets (32 bits) permettant d'affecter à la variable associée une valeur entière comprise entre -2 147 483 648 à 2 147 483 647.

Exemple

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

Fonctions : **setup ()**

La fonction setup est appelée à chaque démarrage de votre programme. Utilisez la pour initialiser vos variables, les E/S, librairies - ex. beginSerial(), etc.

Exemple

```
int buttonPin = 3;
// setup initialise la liaison série
// et la broche pour le bouton
void setup()
{
  beginSerial(9600);
  pinMode(buttonPin, INPUT);
}
// loop lit la broche du bouton en boucle,
// et envoie H ou L sur la liaison série selon
// l'état du bouton
void loop()
{
  if (digitalRead(buttonPin) == HIGH)
    serialWrite('H');
  else
    serialWrite('L');
  delay(1000);
}
```

Fonctions : **loop ()**

La fonction loop () fait précisément ce que son nom suggère, elle se répète indéfiniment, "en boucle". Cela permet à votre programme d'écouter et répondre. Utilisez la pour contrôler activement la carte Arduino.

Exemple

```
int buttonPin = 3;
// setup initialise la liaison serie
// et la broche pour le bouton
void setup()
{
  beginSerial(9600);
  pinMode(buttonPin, INPUT);
}
// loop lit la broche du bouton en boucle,
// et envoie H ou L sur la liaison série selon
// l'état du bouton
void loop()
{
  if (digitalRead(buttonPin) == HIGH)
    serialWrite('H');
  else
    serialWrite('L');
  delay(1000);
}
```

Fonctions : { } accolades

Les accolades s'utilisent pour délimiter un bloc d'instructions. Elles permettent de délimiter des parties de code.

Exemple

```
#define LED1 2
#define LED2 3
int i;
// initialisation
void setup()
{
  i = 0;
}
// allume alternativement 2 LEDs
// connectées aux broches 2 et 3
void loop()
{
  if (i == 0)
  {
    digitalWrite (LED1, HIGH);
    digitalWrite (LED2, LOW);
    i = 1;
  }
  else
  {
    digitalWrite (LED1, LOW);
    digitalWrite (LED2, HIGH);
    i = 0;
  }
  delay(1000); // pause 1 seconde
}
```

Fonctions : // et /* ... */ commentaires

Les commentaires sont utilisés dans les programmes pour annoter le code et indiquer sa fonction. Les commentaires ne sont pas compilés, ils ne sont donc pas exécutés.
Les commentaires sont utiles à la compréhension des programmes, ne les négligés pas.
Il existe deux types de commentaires

* utiliser // pour passer en commentaire le reste de la ligne ;
* utiliser la combinaison /* ... */ pour passer en commentaire le texte inclus entre /* et */.

Notez que les commentaires peuvent être pratiques pour la mise au point de votre code. Ils vous permettent de mettre temporairement sous silence certaines parties du code.

Exemple

```
#define LED1 2
#define LED2 3
int i; // initialisation
void setup()
{
  i = 0;
}
/*
allume alternativement 2 LEDs
connectées aux broches 2 et 3
*/
void loop()
{
  if (i == 0)
  {
    digitalWrite (LED1, HIGH);
    digitalWrite (LED2, LOW); //    i = 1; cette partie du code ne sera pas compilée et pas exécutée les LEDs ne clignoteront pas !
  }
  else
  {
    digitalWrite (LED1, LOW);
    digitalWrite (LED2, HIGH);
    i = 0;
  }
  delay(1000); // pause 1 seconde
}
```

Fonctions : **#define**

syntaxe : #define nomConstante valeur

La macro-commande #define est très utile. Elle vous permet de nommer une valeur avant que le programme ne soit compilé. Avant compilation, c'est-à-dire que les valeurs définies avec #define ne prennent aucun espace mémoire, les valeurs sont substituées à leurs noms au moment de la compilation.

Exemple

Dans cet exemple, toutes les occurrences de LED1 et LED2 seront remplacées respectivement par 2 et 3 au moment de la compilation.

```
#define LED1 2
#define LED2 3
int i;
// initialisation
void setup()
{
  i = 0;
}
// allume alternativement 2 LEDs
// connectées aux broches 2 et 3
void loop()
{
  if (i == 0)
  {
    digitalWrite (LED1, HIGH);
    digitalWrite (LED2, LOW);
    i = 1;
  }
  else
  {
    digitalWrite (LED1, LOW);
    digitalWrite (LED2, HIGH);
    i = 0;
  }
  delay(1000); // pause 1 seconde
}
```

Fonctions : **if (...) ... else ...**

syntaxe : if (condition) instruction(s) { else instruction(s) }
l'instruction if test si une condition est "vraie" (c-a-d quelle renvoie une valeur différente de zéro. Si oui l' / les instruction(s) suivantes sont exécutées, sinon, le programme continu à l'instruction suivante. Si l'instruction suivante est else, l' / les instruction(s) suivant le else sont exécutées. c-a-d que les instructions suivant le else ne sont exécutées que si la condition du if correspondant n'est pas respectée. Le else est optionnel, non obligatoire.
Vous pourrez donc à l'aide de l'instruction if comparer des valeurs issues des entrées numériques, analogiques, série, ... et faire réagir votre programme en conséquence.
Exemple
L'extrait de code qui suit, affichera sur la liaison série (initialisée par ailleurs) "J'aime le fromage".

```
char bAfficher = 1;
char bJaime = 1;
char bLaCervelle = 0;
// ...
void loop ()
{
  if (bAfficher)
  {
   if (bJaime)
    printString("J'aime ");
  if (bLaCervelle <= 0)
  {
    printString ('le ");
    printString ("fromage");
  }
  else
    printString ("la cervelle");
    printString (' - ');
  }
  delay(1000);
}
```

Notez qu'il n'est pas nécessaire d'utiliser la comparaison "différent de zéro" - if (bAfficher) ... cette comparaison est implicite.
Pour les opérateurs de comparaison "est supérieur à", "est égal à", ...voyez [ici].
ATTENTION : une erreur courante dans l'utilisation de if est de confondre l'opérateur "=" affectation de variable et l'opérateur de comparaison "==". C'est "==" qu'on utilise dans la condition de if. Il est possible d'utiliser "=" dans une condition mais pas pour obtenir un résultat de comparaison.

Fonctions : **switch (...) { case : ... }**

syntaxe : switch (valeur) { case : ... break; { ... default : ... break; } }
Comme l'instruction if, le combiné switch aide à contrôler le flot d'informations dans les programmes. Switch case vous permet de créer une liste de "cas" à l'intérieur des accolades du switch. La valeur du switch sera comparée à chacun des cas et Arduino effectuera les instructions du cas correspondant s'il y en a un. Dans le cas ou vous utiliser le cas default, les instructions correspondantes seront exécutées si aucun autre cas ne correspond à la valeur du switch.

Paramètres

valeur : variable qui sera comparée aux cas (case) ;
default : c'est le cas par défaut, s'il est présent c'est ce cas qui sera exécuté si aucun autre cas ne correspond à valeur ;
break : break interrompt les comparaisons et quitte l'instruction switch case.

Exemple

```
switch (var)
{
   case 1:
      // faire quelque chose quand var est égale à 1
     break;
   case 2:
      // faire quelque chose quand var est égale à 2
     break;
   default:
      // si aucun cas ne correspond, faire autre chose
}
```

Fonctions : **while (...)**

syntaxe : while (comparaison) ... ;
ou
syntaxe : do ... while (comparaison);
La boucle while permet de répéter une ou plusieurs instructions tant que la comparaison entre parenthèses est évaluée comme étant vraie. Utilisée en combinaison avec do, la comparaison est reportée en fin de boucle;
Exemple

```
int i = 10;
while (i>0)
{
  digitalWrite (i, HIGH);
  i--; // équivalent à i = i - 1;
}
do
{
  digitalWrite (i, LOW);
  i++;
} while (i<10);
```

Fonctions : **pinMode (noBroche, mode)**

Description
Configure la broche spécifiée pour qu'elle se comporte comme une entrée ou comme une sortie.
Syntaxe
* **pinMode (noBroche, mode)**
Paramètres
* **noBroche : le numéro de la broche que vous souhaitez configurer - type int.**
* **mode : la configuration souhaitée, utilisez les constantes pré-définies INPUT (entrée) ou OUTPUT (sortie) - type int.**
Retour
* **pas de retour**
Exemple

```
int brocheLED = 13;               // LED connectée sur la broche numérique 13
void setup()
{
  pinMode(brocheLED, OUTPUT);     // configure la broche numérique 13 en sortie
}
void loop()
{
  digitalWrite(brocheLED, HIGH);  // allume la LED
  delay(1000);                    // pause d'une seconde
  digitalWrite(brocheLED, LOW);   // éteint la LED
  delay(1000);                    // pause d'une seconde
}
```

Fonctions : **digitalWrite (noBroche, valeur)**

Description
Fixe l'état de la broche spécifiée au niveau haut ou bas.
Syntaxe
* **digitalWrite (noBroche, valeur)**
Paramètres
* **noBroche : le numéro de la broche dont vous souhaitez fixer l'état - type int.**
* **valeur : HIGH (haut) ou LOW (bas) - type int.**
Retour
* **pas de retour**
Exemple

```
int brocheLED = 13;              // LED connectée sur la broche numérique 13
void setup()
{
  pinMode(brocheLED, OUTPUT);      // configure la broche numérique 13 en sortie
}
void loop()
{
  digitalWrite(brocheLED, HIGH);   // allumer la LED
  delay(1000);                // pause d'une seconde
  digitalWrite(brocheLED, LOW);    // éteint la LED
  delay(1000);                // pause d'une seconde
}
```

Fonctions : **int analogRead (noBroche)**

Description
Retourne la valeur numérique de la valeur analogique lue sur la broche analogique spécifiée. La carte Arduino opère une conversion analogique/numérique sur 10 bits. Cela signifie qu'elle transforme une valeur analogique comprise entre 0 et 5 volts en valeur numérique entière comprise entre 0 et 1024.
Syntaxe
*** int analogRead (noBroche)**
Paramètres
*** noBroche : le numéro de la broche analogique que vous souhaitez lire de 0 à 5 - type int.**
Retour
*** retourne une valeur entière comprise entre 0 et 1024 - type int.**
Note
*** Les entrées analogiques contrairement aux entrées numériques n'ont pas à être configurées en entrée ou en sortie. Il est donc inutile d'utiliser la fonction pinMode pour ces broches, elle est inutile et inopérante.**
Exemple

```
int brocheLED = 13;   // LED connectée sur la broche numérique 13
int brocheAnalog = 3;   // potentiomètre connecté sur la broche analogique 3
int val = 0;   // variable destinée au stockage de la valeur lue
int seuil = 512;   // seuil
void setup()
{
  pinMode(brocheLED, OUTPUT);   //configure la broche numérique 13 en sortie
}
void loop()
{
  val = analogRead(brocheAnalog);   // read the input pin
  if (val >= seuil)    // la valeur lue sur l'entrée analogique est elle supérieure ou égale au seuil (512) ?
  { // si oui
    digitalWrite(brocheLED, HIGH);   // allume la LED
  }
  else
  { // si non
    digitalWrite(brocheLED, LOW);   // éteint la LED
  }
}
```

Fonctions : **analogWrite (noBroche, valeur)**

Description
Ecrit une valeur analogique (PWM - modulation en largeur d'impulsion - MLI) sur une des broches 9, 10 ou 11 ; respectivement PWM0, PWM1 et PWM2. Après un appel à anlogWrite, la broche spécifiée générera un signal carré à fréquence et rapport cyclique constants jusqu'au prochain appel à analogWrite (ou un appel à digitalWrite ou digitalRead sur la même broche).
Syntaxe
*** int analogWrite (noBroche, valeur)**
Paramètres
*** noBroche : le numéro de la broche "analogique" vers laquelle vous voulez envoyer le signal MLI (PWM) - type int.**
*** valeur : fixe le rapport cyclique, valeur entre 0 et 255. Une valeur à génère un signal constament à 0 volts, une valeur à 255 génère un signal constamment à 5 volts sur la broche spécifiée. Pour les valeurs comprises entre 0 et 255, la broche oscillera rapidement entre 0 et 5 volts. Plus la valeur est élevée plus longtemps restera au niveau haut (5 volts) par rapport au niveau bas (0 volt).**
Par exemple, pour une valeur de 64, le signal sera au niveau bas les 3/4 du temps et au niveau haut 1/4 du temps (64/255 = 1/4) - type int.
Retour
*** pas de retour**
Note
*** L'utilisation de la fonction analogWrite sur les broches 9 à 11 dispense d'utiliser la fonction pinMode pour ces broches.**
*** La fréquence du signal MLI (PWM) est approximativement 30769 Hz.**
*** analogWrite ne fonctionne que sur les broches 9, 10, and 11 ; sur les autres broches numériques le niveau écrit sera 0 ou 5 volts. Mais, utilisez digitaWrite pour ces broches.**
Exemple

```
int brocheLED = 9;   // LED connectée à la broche numérique 9
int brocheAnalog = 3;   // potentiomètre connecté à la broche analogique 3
int val = 0;   // variable destinée à stocker la valeur lue
void setup()
{
  pinMode (brocheLED, OUTPUT);   // configure la broche en sortie
}
void loop()
{
  val = analogRead (brocheAnalog);   // lecture de l'entrée analogique
  analogWrite (brocheLED, val / 4);  // la valeur val varie de 0 à 1023, les valeurs analogWrite de 0 à 255
}
```

Fonctions : **Serial.begin (debit)**

Description
Fixe le débit de transmission, en bps (bits par seconde), de la liaison série.
Syntaxe
*** Serial.begin (debit)**
Paramètres
*** debit : le débit en bps - type int.**
Retour
*** pas de retour**
Note
*** Pour des raisons historiques, on utilise "généralement" la valeur 9600, mais cela n'a plus vraiment de sens actuellement et c'est même "ridicule" avec la carte Arduino équipée en USB. Voilà pourquoi dans beaucoup d'exemples, vous verrez la valeur 9600 en paramètre de cette méthode.**
*** bps vs Baud, l'éternelle question... pour la Carte Arduino, sauf erreur, bps = Baud, la transmission s'opérant sans modulation.**
Exemple

```
void setup()
{
  pinMode(brocheLED, OUTPUT);   //configure la broche numérique 13 en sortie
}
```

Fonctions : **int Serial.available ()**

Description
Retourne le nombre d'octets (caractères) en attente de lecture dans le tampon série.
Syntaxe
*** int Serial.available ()**

Paramètres
*** aucun**
Retour
*** retourne combien d'octets sont en attente de lecture dans le tampon série. Si des données sont présentes dans le tampon série, la valeur retournée par Serial.available sera supérieure à 0 - type int.**
Note
*** La taille du tampon série est de 64 caractères.**
Exemple

```
int octetEntrant = 0; // pour les données série
void setup() {
  Serial.begin(9600);        // ouvre le port série et fixe le débit à 9600 bps
}
void loop()
{
  // n'envoyer des données que l'orsque que des données sont reçues :
  if (Serial.available() > 0) {
    // lecture de l'octet entrant :
    octetEntrant = Serial.read();
    // dire ce que a été reçu :
    Serial.print ("J'ai reçu : ");
    Serial.println (octetEntrant, DEC);
 }
}
```

Fonctions : **int Serial.read ()**

Description
Retourne le prochain octet à lire dans tampon série. Cette méthode lit donc, un par un, les caractères reçus sur la liaison série.
Syntaxe
* **int Serial.read ()**
Paramètres
* **aucun**
Retour
* **retourne le dernier(*) octet reçus sur la liaison série ou -1 s'il n'y a rien à lire - type int.**

(*) : Le terme dernier est inexacte mais, je pense, plus compréhensible. En effet, ce n'est pas nécessairement le dernier octet reçu qui est renvoyé, mais le prochain à lire dans l'ordre d'arrivée.
Note
* **La taille du tampon série est de 64 caractères.**

Exemple

```
int octetEntrant = 0; // pour les données série
void setup() {
  Serial.begin(9600);        // ouvre le port série et fixe le débit à 9600 bps
}
void loop()
{
  // n'envoyer des données que lorsque que des données sont reçues :
  if (Serial.available() > 0) {
    // lecture de l'octet entrant :
    octetEntrant = Serial.read();
    // dire ce que a été reçu :
    Serial.print ("J'ai reçu : ");
    Serial.println (octetEntrant, DEC);
 }
}
```

Fonctions : **Serial.print (donnees {, type })**

Description
Envoie des données vers le port série.
Syntaxe
Cette méthode polymorphe peut prendre plusieurs formes :

- **Serial.print (b), envoie sous la forme d'une chaîne ASCII, la valeur décimale de b.**
- **Serial.print (b, DEC), envoie sous la forme d'une chaîne ASCII, la valeur décimale de b.**
- **Serial.print (b, HEX), envoie sous la forme d'une chaîne ASCII, la valeur hexadécimale de b.**
- **Serial.print (b, OCT), envoie sous la forme d'une chaîne ASCII, la valeur octale de b.**
- **Serial.print (b, BIN), envoie sous la forme d'une chaîne ASCII, la valeur binaire de b.**
- **Serial.print (b, BYTE), envoie l'octet b.**
- **Serial.print (chaine), si chaine est une chaîne de caractères ou un tableau, envoie l'ensemble sous la forme d'une chaîne ASCII.**

Paramètres

- **donnees : données à envoyer, selon la version de print utilisée un octet ou une chaîne de caractères.**
- **type : paramètre présent ou optionnel qui spécifie le le type de conversion à effectuer pour l'envoi d'un octet.**

Retour

- **pas de retour**

Note

- **Dans le cas où l'on considère b égal à 79 :**
  - **Serial.print (b); enverra la chaîne "79".**
  - **Serial.print (b, DEC); enverra la chaîne "79".**
  - **Serial.print (b, HEX); enverra la chaîne "4F".**
  - **Serial.print (b, OCT); enverra la chaîne "117".**
  - **Serial.print (b, BIN); enverra la chaîne "1001111".**
  - **Serial.print (b, BYTE); enverra le caractère "O", qui est le caractère 79, de la table ASCII.**

- **Dans le cas où l'on considère str égale à "Bonjour Monde !" :**
  - **Serial.print (str); enverra la chaîne "Bonjour Monde !".**

Fonctions : **Serial.print (donnees, { type })**

Exemple

```
int valeurAnalogique = 0;
void setup()
{
  Serial.begin (9600);
}
void loop()
{
  valeurAnalogique = analogRead (0); // lecture de la valeur analogique présente
sur la broche analogique 0
  // écriture vers le port série
  Serial.print ("Valeur lue : ");       // "Valeur lue : "
  Serial.print (valeurAnalogique);      // [valeur décimale]
  Serial.print (" en base 10");         //  " en base 10"
  Serial.print ("\t");                  // tabulation
  Serial.print (valeurAnalogique, HEX); // [valeur en hexa]
  Serial.print (" en Hexa.");           // " en Hexa."
  Serial.println ();                    // nouvelle ligne / retour chariot
  // attente de 100 millisecondes avnt bouclage
  delay(100);
}
```

Fonctions : **Serial.println ({ donnees {, type } })**

Description

Envoie des données vers le port série et ajoute un "retour chariot" à la suite des données transmises.

La méthode Serial.println étant très similaire à sa jumelle Serial.print, pour de plus amples informations sur son utilisation consultez la fiche de Serial.print.

Syntaxe

* **Seule différence avec Serial.print la syntaxe Serial.println() (sans paramètre) a un sens : envoyer un "retour chariot" seul.**
* **consultez la fiche de Serial.print.**

Paramètres

* **consultez la fiche de Serial.print.**

Retour

* **pas de retour**

Note

* **Le retour chariot est composé d'un caractère ASCII 13, "\r" ; suivi d'un caractère ASCII 10, "\n".**
* **Notez que la fonction printNewline() de la librairie série de la version 0003 n'envoyait qu'un caractère ASCII 13, "\n".**

Fonctions : **unsigned long millis ()**

Description
Cette fonction renvoie le nombre de millisecondes écoulées depuis le lancement du programme courant.
Syntaxe
*** unsigned long millis ()**
Paramètres
*** aucun**
Retour
*** retourne le nombre de millisecondes écoulées depuis le lancement du programme - type unsigned long.**
Exemple

```
unsigned long tempsEcoule;
void setup()
{
  Serial.begin(9600);
}
void loop()
{
  Serial.print("Ce prog. s'exécute depuis : ");
  tempsEcoule = millis();
  Serial.print (tempsEcoule);
  Serial.println (" millisec.");
  delay (1000);
}
```

Fonctions : **delayMicroseconds (NbMicroSecondes)**

Description
Arrête le déroulement du programme le temps spécifié (en microsecondes) en paramètre.
Syntaxe
*** delayMicroseconds (NbMicroSecondes)**

Paramètres
*** NbMicroSecondes : durée de la pause en microsecondes. Il y a 1 000 000 microsecondes dans 1 seconde et 1 000 microsecondes dans 1 milliseconde - type unsigned int.**

Retour
*** pas de retour**

Exemple

```
// génère un signal carré d'une fréquence de 1KHz
// sur la broche numérique 8
int brocheSignal = 8;              // sortie du signal carré sur la broche numérique 8
void setup()
{
  pinMode(brocheSignal, OUTPUT);      // configure la broche numérique 8 en sortie
}
void loop()
{
  digitalWrite(brocheSignal, HIGH);  // niveau haut
  delayMicroseconds(500);         // pause de 500 microsecondes
  digitalWrite(brocheSignal, LOW);   // niveau bas
  delayMicroseconds(500);         // pause de 500 microsecondes
}
```

Fonctions : **int digitalRead (noBroche)**

Description
Retourne la valeur de la broche spécifiée, HIGH ou LOW.
Syntaxe
* **int pinMode (noBroche)**
Paramètres
* **noBroche : le numéro de la broche que vous souhaitez lire - type int.**
Retour
* **retourne HIGH ou LOW - type int.**
Exemple

```
int brocheLED = 13;    // LED connectée sur la broche numérique 13
int brocheBouton = 7;    // bouton poussoir connecté à la broche numérique 7
int val = 0;    // variable pour stocker la valeur lue
void setup()
{
  pinMode (brocheLED, OUTPUT);      // configure la broche numérique 13 en sortie
  pinMode (brocheBouton
, INPUT);      // configure la broche numérique 7 en entrée
}
void loop()
{
  val = digitalRead (brocheBouton);   // lit l'entrée
  digitalWrite (brocheLED, val);    // recopie de la valeur du bouton sur la LED
}
```

Fonctions : **unsigned long pulseIn (noBroche, valeur)**

Description
Détermine la durée d'une impulsion haute ou basse sur la broche spécifiée. Si valeur est fixée à HIGH (haut), pulseIn attends que la broche spécifiée soit à l'état haut, commence à chronométrer, attends que cette broche passe à l'état bas et arrête le décompte de la durée ; puis renvoie la durée de l'impulsion en microsecondes.
Syntaxe
* **unsigned long pulseIn (noBroche, valeur)**

Paramètres
* **noBroche : le numéro de la broche ou sera attendue l'impulsion - type int.**
* **valeur : type d'impulsion attendue, HIGH (haute) ou LOW (basse) - type int.**

Retour
* **retourne la durée de l'impulsion lue en microsecondes - type unsigned long.**

Exemple

```
int broche = 7;
unsigned long duree;
void setup()
{
  pinMode (broche, INPUT);
}
void loop()
{
  durree = pulseIn (broche, HIGH);
}
```

Fonctions : **PWM**

PWM : Pulse Width Modulation = Modulation en largeur d'impulsion = MLI

<u>Définition rapide</u>

Il s'agit d'une technique de création de signaux qui permet de contrôler de l'analogique avec des sorties numériques d'un microcontrôleur. La commande en PWM consiste en une *succession rapide de signaux numériques*. Ce peuvent être par exemple des signaux tout ou rien, c'est-à-dire alternant sans transition entre une valeur minimale fixe A (extinction) et une valeur maximale fixe B (allumage). Si la fréquence choisie est suffisamment rapide, le résultat de cette rapide alternance d'allumage/extinction se fond dans une valeur moyenne. En fonction de la durée des signaux A en proportion des signaux B, la *moyenne* obtenue donne un grand nombre de valeurs intermédiaires (le protocole choisi pour la commande détermine ce nombre).

Dans notre cas, les circuits ou appareils analogiques concernés sont des actionneurs comme les ampoules basse tension, les moteurs à courant continu ou des électro-aimants.

Fonctions : **PWM**

Considérations sur les signaux analogiques

Ces actionneurs sont normalement alimentés en *tension continue* (analogique). Leur action (lumière ou mouvement) est proportionnelle à cette tension d'alimentation. L'action est nulle quand l'alimentation est à 0, elle est maximale quand l'alimentation est celle indiquée comme normale dans la documentation technique de l'actionneur concerné. La caractéristique fondamentale d'une tension analogique est que sa *résolution* est infinie. La notion de résolution, le nombre de pas possible dans une gamme de valeur, ne prend de sens que lorsqu'on a affaire à une numérisation (donc une discrétisation) des signaux. Dans ce dernier cas, les signaux directement issus d'un contrôleur numérique ne peuvent prendre qu'un nombre fini de valeur.

A priori il semble plus direct et plus riche de possibilités d'utiliser une commande par des signaux analogiques à la finesse illimitée plutôt qu'une commande basée sur du tout ou rien, aux valeurs limitées et fixes. Pourtant c'est souvent la deuxième solution qui est choisie par rapport à la première, pour diverses raisons : les circuits purement analogiques présentent parfois des *variations* au cours du temps que n'ont pas les circuits numériques. La possibilité de rendre fiables, précis et stables les circuits analogiques se heurte à des difficultés techniques, financières et simplement parfois à des problèmes d'encombrement spatial. Ces difficultés ne seraient pas forcément si problématiques si l'analogique n'était pas confronté à deux écueils supplémentaires : la chauffe des composants de puissance et la sensibilité au parasitage électromagnétique. Cette notion de sensibilité au parasitage selon la nature analogique ou numérique du circuit a également beaucoup d'importance dans l'utilisation des capteurs et fera l'objet d'une page spécifique.

Fonctions : **PWM**

Intérêt du PWM

Le PWM permet d'obtenir un équivalent d'une variation de tension continue à l'aide d'un contrôle en tout ou rien (voire en valeurs discrètes, ce qui n'est pas le cas sur nos cartes). Cette technique permet aux composants de puissance de beaucoup moins chauffer qu'en analogique. D'autre part, les signaux numériques sont moins sensibles au parasitage que les signaux analogiques et sont donc plus robustes.

Le principal intérêt de la technique PWM est de *limiter la chauffe* des composants électroniques. En effet, en commande analogique, pour obtenir une variation de puissance il faut dissiper le complément de la puissance maximale consommée. Par exemple : une lampe de 20 Watts allumée au maximum consomme 20 Watt. Si par une commande de gradation elle est allumée au quart de sa puissance, elle consomme 5 Watt. Le composant analogique devrait alors dissiper 15 W, ce qui implique un énorme radiateur... En PWM, la puissance fournie est soit maximale, soit nulle. Lorsqu'elle est maximale, pendant un quart du temps dans cet exemple, il n'y a pas besoin de dissiper de puissance résiduelle. Lorsqu'elle est nulle, il n'y a pas besoin de dissiper non plus de puissance, car elle n'est pas fournie du tout...

La moindre sensibilité au parasitage du PWM n'est pas en fait aussi critique que la diminution de température par rapport à l'analogique mais mérite tout de même une explication. Même si le résultat d'une commande en PWM peut s'assimiler à une valeur moyenne, les signaux sont toujours à l'origine en tout ou rien donc très distincts. L'analogique est très sensible au *parasitage* car une variation sur un signal continu prend tout de suite beaucoup d'importance. En revanche, un parasite électromagnétique peut difficilement affecter un signal On ou Off : il faudrait que le parasite atteigne en amplitude au moins la moitié de la valeur maximale, pour transformer un Off en On ou l'inverse, ce qui suggère un environnement vraiment peu recommandé pour faire fonctionner un actionneur.

Fonctions : **PWM**

Fréquence de PWM

La commande d'actionneurs de puissance par PWM est très liée à la notion de fréquence. Pour que l'impression d'une valeur moyenne constante d'allumage apparaisse, il faut que l'alternance d'allumage/extinction soit suffisamment *rapide* pour qu'elle ne se remarque pas.

Si par exemple le cycle complet de PWM durait une seconde (cette durée est nommée période), ce qui donne une fréquence de 1 Hertz, les durées d'allumage et d'extinction de l'actionneur seraient réparties proportionnellement sur cette seconde. Imaginons une lampe halogène allumée à 40% de sa puissance. Elle doit être alimentée 40% du temps et éteinte durant 60 % du temps. Avec une fréquence de 1 Hz, elle serait allumée pendant 0,4 seconde puis éteinte pendant 0,6 seconde. L'effet de clignotement résultant est parfaitement perceptible... La fréquence du PWM doit donc être beaucoup plus grande que 1 Hz, ou autrement dit la période doit être bien plus courte qu'une seconde.

Selon les utilisations la fréquence de PWM va de 100 Hz (100 cycles par seconde) à 200 kHz. Nos cartes de commande permettent actuellement d'avoir deux fréquences de PWM : 100 Hz et 400 Hz. Dans certains cas d'actionneurs assez réactifs, l'envoi de commandes de valeurs proches de 0 provoque un petit tremblement si la fréquence est peu élevée. Les lampes à incandescence en général montrent un bon lissage des valeurs grâce à leur latence d'allumage et d'extinction.