Codes ASCII de quelques caractères spéciaux. *(En décimal.)*

CAR	code	CAR	code	CAR	code	CAR	code
à	224	ì	236	ù	249	Ç	199
á	225	í	237	ú	250	È	200
â	226	î	238	û	251	É	201
ä	228	ï	239	ü	252	Ê	202
è	232	ò	242	œ	156	Î	206
é	233	ó	243	±	177	Ô	212
ê	234	ô	244	µ	181	À	192
ë	235	ö	246	ç	231	Â	194

Pour obtenir tous les codes possibles utiliser Codes_SPECIAUX.ino

Envoyer des caractères sur la ligne série avec print().

C'est l'instruction à privilégier quand on désire envoyer sur la ligne série des caractères représentant le contenu d'une variable, quelle soit de type numérique ou de type chaîne de caractères. Comme pour le cas de Sérial.write les caractères spéciaux seront altérés et doivent être exprimés explicitement par leurs codes ASCII.

```
Serial.print(char(212)); // Affiche un "Ô" non altéré.
byte X = 73; // Sans format print affichera la valeur 73.
int Y = 12345; // Sans format print affichera la valeur 12345.
float Z = 12345.67890123; // Affiché avec 2 décimales par défaut.
```

Formats pour les nombres entiers avec l'instruction print() :

```
Serial.println(X,5);    // Affiche la valeur de X en base 5.
Serial.print(Y,DEC);  // Affiche Y en décimal avec format spécifié.
Serial.print(Y,OCT);  // Affiche Y en binaire codé OCTAL.
Serial.print(X,HEX);  // Affiche X en binaire codé HEXADECIMAL.
Serial.println(Y,BIN);  // Affiche la valeur de Y en BINAIRE pur.
```

NOTE : Les zéros en tête ne sont pas affichés.

Formats pour les nombres float avec l'instruction print() :

```
Serial.println(Z,0); // Pas de décimales.
Serial.println(Z,1); // Une seule décimale.
Serial.print(Z,4); // Quatre décimales et @.
Serial.print(Z,15); //Quinze décimales et @.
```

@ : Dans tous les cas float est limité à sept ou huit chiffres significatifs.

RAPPEL : Si la valeur pour imposer un nombre de décimales est remplacée par BIN, OCT, DEC ou HEX, le nombre de décimales sera respectivement de 2, 8, 10 ou 16.

SYNTAXE du langage C d'ARDUINO

Les séparateurs et caractères spéciaux.

"**;**" : Termine une instruction. *(Obligatoire à la fin de chaque instruction)*
"**{**" et "**}**" : Délimitent un "bloc" d'instructions.
NOTE : Le ";" après le délimiteur "}" n'est pas impératif.
"**#**" : Précise une **entité qui sera traitée par le Précompilateur**.
"**//**": La fin de cette ligne est un commentaire ignoré par la compilation.
"**/***" et "*/" : Délimiteurs d'un commentaire étalé sur plusieurs lignes.

Les structures de base impératives.

```
// Instructions d'initialisation de la carte.
void setup() { Instructions ; }
// Programme principal : Boucle infinie.
void Loop() { Instructions ; }
// Procédure de servitude.
```

Sont obligatoires mêmesi elle ne contiennent pas d'instruction.

```
void NomProcedure(Paramètres) { Instructions ; }
```

NomProcedure :
- Des lettres, des chiffres et "_".
- Pas de chiffre en premier caractère.
- Majuscules et minuscules sont différenciées.

Les caractères accentués sont interdit en langage C d'Arduino sauf dans les commentaires.

Travail d'écriture sur la ligne série :

Pour dialoguer avec le monde extérieur via une ligne série, Arduino va échanger des octets avec les tampons de l'UART tant en entrée qu'en sortie. En fonction de la nature des informations échangées, textes ou entités numériques, des traitements spécifiques s'imposent, mais dans tous les cas les transferts se font octet par octet.

Envoyer des caractères sur la ligne série avec write().

Conceptuellement l'instruction write('caractère') ou write("chaine") est destinée à envoyer des octets considérés comme des caractères ASCII dont on donne le code à convenance sous divers formats possibles :

```
Serial.write('A'); // Caractère "A" directement précisé.
Serial.write(66); // Affiche "B" code ASCII donné en décimal.
Serial.write(B1000011); // Affiche "C" code ASCII donné en binaire.
```

Puisque cette instruction ne manipule que des octets, une valeur > 255 va conduire à faire ignorer les bits de poids fort supérieurs au rang 7 :

```
Serial.write(33572); // Affiche "$". Dans cet exemple, la valeur 33572 se traduit en binaire par 10000011-00100110. Seul l'octet de poids faible est utilisé par le compilateur. C'est le code ASCII du "$" soit 36 en décimal.
Serial.write("TEXTE"); Se contente de lister en ligne les caractères du texte sans interprétation. Noter que writeln() n'est pas accepté. (Voir '\n')
```

Caractères réservés pour l'instruction write() :

```
Serial.write('\n'); // Affiche CR suivi de LF sur la ligne série USB.
Serial.write('\t'); // Affiche une tabulation.
Serial.write('\r'); // Affiche un CR sur terminal standard mais ignoré
    sur le terminal série USB de l'environnement Arduino.
```

Afficher des caractères spéciaux sur la ligne série.

L'utilisation directe de caractères spéciaux conduit à des aléas d'affichage car le compilateur traduit ces derniers en utilisant la norme UTF-8 pour stocker les valeurs équivalentes en mémoire. Pour obtenir des affichages cohérents il faut impérativement passer par leurs codes ASCII exprimés en décimal ou en Binaire. Deux méthodes possibles :

- **Donner leur code ASCII** et préciser que c'est un char.

```
Serial.write(char(233)); // Affiche un "é" non altéré.
```

Cette structure est également valides avec **Serial**.print().

- **Affecter une variable** byte ou int **avec leur code ASCII**.

```
Caractere = B11100111; // Code ASCII du "ç" exprimé en binaire.
Serial.write(Caractere); // Affiche correctement le "ç".
```

... / ...

les textes issus de la ligne série. Chaque octet extrait de la pile est alors défini comme un caractère de type char et concaténé à la variable String. Le programme Saisie_CHAINES_sur USB.ino et le programme ESSAI_de_SERIAL_EVENT.ino sont deux variantes d'une telle méthode. Dans les exemples proposés sur Internet, la détection de la fin de la saisie est effectuée sur le CR codé '\r'. Sur le terminal d'Arduino ce caractère ne convient pas. Il faut indiquer la fin de la saisie par un caractère laissé au choix du programmeur et noté "sentinelle" dans ces deux programmes.

```
#define Sentinelle '*' // Choisir un caractère pour la "sentinelle".
char Caractere;
String Chaine_Memorisee = ""; // Emplacement pour la chaîne entrante.
boolean Sentinelle_detectee = false;  // Délimiteur de fin détecté.
void setup() {Serial.begin(115200); Chaine_Memorisee.reserve(200);}
 // Reserve 200 octets pour la chaine. Taille possible quelconque.
void loop() {
 while (Serial.available()) { Caractere = Serial.read(); // Lire un Octet.
  // Puis concaténer ce caractère sauf si c'est la sentinelle.
  if (Caractere != Sentinelle) Chaine_Memorisee += Caractere;
  // Si sentinelle détectée, informer la boucle de base :
  if (Caractere == Sentinelle) Sentinelle_detectee = true; } ;
 if (Sentinelle_detectee) { Instructions pour TRAITER la CHAINE;
 Chaine_Memorisee = ""; // Vider le tampon pour une nouvelle capture.
 Sentinelle_detectee = false; } } //Réarme une attente de texte.
```

Aléas liés à la lenteur de transmission sur un terminal.

Quand on effectue des lectures d'octets sur un terminal de type série, une foule de comportements anormaux surgissent et sont la conséquence d'une lenteur relative avec laquelle la ligne série débite ses octets. Même à la vitesse maximale de 115200 bauds, il faut environ 100µs pour fournir un caractère. Le microcontrôleur qui cadence à 16MHz a largement le temps d'exécuter plusieurs instructions. Par conséquent, entre la réception de deux caractères il peut vider la PILE de réception. Si on désire effectuer le traitement quand toute la chaîne est extraite du tampon de la ligne série, on utilise la fonction **Serial**.available(). Mais compte tenu des explications qui précèdent, elle revoie une valeur erronée puisque le buffer est provisoirement vidé. La parade consiste à :

- Utiliser la vitesse maximale sur le terminal d'Arduino. *(115200bauds)*
- Toujours mettre un delay(5) après un **Serial**.available() afin de laisser le temps à la transmission de tous les octets de se terminer. C'est particulièrement vrai si les chaînes saisies sont longues.

La procédure setup() est appelée au démarrage du programme pour initialiser les variables, le sens des broches, les librairies utilisées ...
La procédure setup() n'est exécutée qu'une seule fois, après chaque mise sous tension ou sur un RESET de la carte Arduino.
La procédure loop () s'exécute en boucle sans fin et constitue le corps du programme. Elle est obligatoire, même vide, dans tout programme.
En C standard c'est la fonction main() qui intègre setup() et loop().

Les constantes. *(Voir page 6)*

```
byte const Masse = 2;  // Sortie 2 pour la masse.
const byte Bleu = 3;   // Sortie 3 pour piloter le BLEU.
```

Constantes prédéfinies :

HIGH, LOW, INPUT, OUTPUT, false, true.
PI : Constante = 3.14159274 limitée à 7 chiffres. *(Voir page 4)*

NOTE : Les constantes #define et const peuvent se trouver n'importe où dans le programme, mais doivent être déclarées avant leur utilisation.

Les variables.

Une variable est un identificateur typé, qui à la compilation réserve un emplacement de mémoire adapté à sa taille présumée.

Les bascules de type boolean.

Deux variantes sont possibles pour inverser l'état d'un booleen. La première méthode consiste à utiliser l'opérateur NON :

```
boolean LED_active;
LED_active = !LED_active;
```

Mais le codage interne de false et de true étant respectivement "0" et "> 0", on peut aussi utiliser l'astuce suivante **qui fait gagner 4 octets** :

```
LED_active = 1 - LED_active;
```

Conversions de types de variables.

Fonction	Valeur renvoyée	Taille
char(X)	Valeur de type char	8 bits
byte(X)	Valeur de type byte	8 bits
int(X)	Valeur de type int	16 bits
word(X)	Valeur de type word	16 bits
Long(X)	Valeur de type Long	32 bits
float(X)	Valeur de type float	32 bits

X est une variable de n'importe quel type.
word(H,L) H : Octet de poids fort / L : Octet de poids faible.

Les variables de type float et de type double.

Déclarent des variables de type virgule-flottante, c'est à dire des nombres décimaux. Ces variables sont souvent utilisées pour l'expression des valeurs analogiques et continues, parce qu'elles ont une meilleure résolution que les nombres entiers. Les nombres décimaux ainsi stockés peuvent prendre des valeurs aussi élevées que 3.4028235E+38 et aussi basses que -3.4028235E+38. Les variables float et double présentent **seulement 6 à 7 chiffres significatifs de précision**. Ceci concerne le nombre total de chiffres, partie entière et décimale comprises. À la différence d'autres plateformes sur lesquelles on peut obtenir plus de précision en utilisant une variable de type double, sur le compilateur d'Arduino, les variables double ont même précision que les float.

> La précision d'un réel de type float et celle d'un type double est limitée à une définition totale de 6 à 7 chiffres significatifs, partie entière et partie fractionnaire confondues.

Les nombres à virgule flottante sont forcément des approximations, et peuvent conduire à des résultats imprévus quand ils sont comparés. Exemple : 6.0 divisé par 3.0 peut ne pas être strictement égal à 2.0 raison pour laquelle il ne faut pas utiliser l'égalité dans une condition. Il est recommandé de vérifier que la valeur absolue de la différence entre le rapport calculé et la borne testée soit inférieure à un seuil très petit.
Exemple de déclaration et d'utilisation de réels :

```
int X; // Déclare une variable entière de type int appelée X.
int Y; // Déclare une variable entière de type int désignée Y.
float Z; // Déclare unréel de type float identifié Z.
X = 1;
Y = X / 2; // Y vaut 0 car les entiers n'ont pas de décimales.
Z = (float)X / 2.0; // Z vaut actuellement 0.5.
```

(Remarquer la conversion de X en float pour avoir 2.0 et non 2.)

Optimisation des programmes :

Les opérations mathématiques sur les nombres à virgules flottantes sont plus lentes que les calculs effectués sur les nombres entiers. S'ils doivent s'exécuter à une vitesse maximale, les programmateurs adoptent souvent la stratégie qui consiste à convertir les calculs de virgule flottante vers un traitement sur des entiers pour améliorer la rapidité d'exécution.

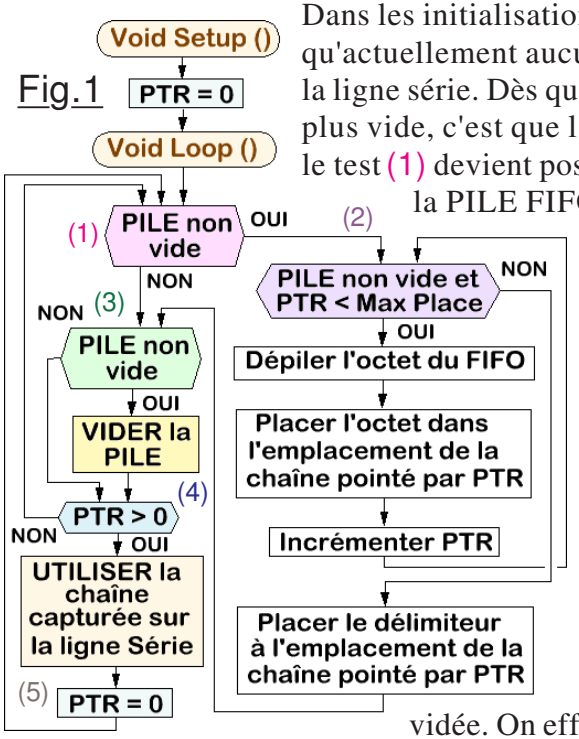
Récupérer une chaîne de caractères sur la ligne série :

Deux méthodes de base sont applicables pour récupérer une chaîne de caractères issue de la PILE d'attente de la ligne série. Soit on procède caractère par caractère dans une variable chaîne, soit on extrait octet par octet que l'on stocke dans un tableau de caractères. Dans tous les cas, si l'on désire éviter les effets de bord, il faut gérer les espaces réservés aux chaînes dans le programme.

Récupérer du texte dans un tableau de caractères.

Un programme très complet avec vérification du non débordement est proposé dans Lire_Chaine_LGR_verifiee.ino dont voici la structure :

Fig.1



Dans les initialisations de base, PTR=0 signifie qu'actuellement aucune chaîne n'est arrivée sur la ligne série. Dès que la PILE de réception n'est plus vide, c'est que l'on a reçu des caractères et le test (1) devient positif. On va alors extraire de la PILE FIFO le texte saisi sur la ligne série. Sur la Fig.1 les instructions de lecture sont non colorées. Mais avant de lire l'octet en attente dans la PILE, le test (2) vérifie que l'on ne va pas déborder la taille réservée à la chaîne et pointée par PTR. Il est possible que la chaîne saisie sur la ligne série dépasse la taille autorisée. Dans ce cas la PILE n'est pas entièrement vidée. On effectue la vérification (3) que l'on peut faire suivre de la purge du tampon de donnée, ou laisser en attente et la traiter après l'instruction repérée en orange. Le test (4) a pour fonction de déterminer si une réception sur la ligne série a été traitée. Enfin l'instruction (5) signale qu'il n'y a plus de chaîne à traiter.

Récupérer du texte dans une variable de type String.

La technique consiste à réserver un emplacement de type String pour

... / ...

chaine.charAt(N);

Retourne un caractère copié dans chaine de type String et pointé par N, l'index du premier étant égal à 0. *(Attention au débordement de N)*

chaine1.concat(chaine2); *(Voir également l'opérateur +)*

Concatène dans chaine1 les deux objets String chaine1 et chaine2. *Contrairement à ce que précise la "documentation mise en ligne" il ne semble pas possible de concaténer dans une String tierce mais uniquement dans la String "d'origine" chaine1.*

chaine.toLowwerCase(); chaine.toHupperCase();

Fonction qui **transforme le contenu d'une chaîne** de caractères en son équivalent Minuscule ou Majuscule. **Seuls les caractères ASCII alphabétiques sont modifiés**, les autres caractères restent inchangés.

chaine.trim();

Purge une chaîne de type String de ses 'espaces' en tête et en queue.

Opérateurs de la classe String.

Accéder à un caractère particulier d'un objet String.

```
Serial.println(texte[3]); // Affichera "J". (Premier caractère = n°0)
Serial.println(LCD[1]); // Affichera "Ligne 2". (Première chaîne = n°0)
```

(Exemples issus du listage donné en page 42.)

Concaténer des String avec des chaînes ou des caractères.

```
String Texte1, Texte2, TOTAL; int Valeur = -23456; byte Octet = 123;
void setup() { Serial.begin(115200); Serial.println();
 Texte1 = String("-111111-"); Texte2 = String("-222222-");}
void loop() { String texte1; String texte2; // Instancier deux String.
 texte1 = "BONJOUR"; texte2 = String(" LES COPAINS");
 String TOTAL = texte1 + texte2; // Instancier TOTAL par concaténation.
 texte2 = texte2 + texte2; // Une String concaténée avec elle-même.
 TOTAL = String("aaaa") + String("zzzzzz"); // Concaténer des chaînes.
 TOUT =  Texte1 + Octet; // Concaténer une String avec une variable byte.
 TOTAL =  Texte1 + Valeur; // Concaténer une String avec une variable int.
 TOUT = Texte2 + 123456789; // Concaténer une String avec un entier.
 TOTAL =  Texte1 + 'A'; // Concaténer une String avec un caractère.
 TOUT =  'A' + Texte2;  // Concaténer est "commutatif".
 TOTAL =  Texte1 +  "abc"; // Concaténer une String avec une chaîne.
```

Fonction comparaison de String.

La fonction chaine1 == chaine2 ne retourne true que si les deux chaînes sont strictement égales **en longueur et en contenu**.

chaine1 peut être une "chaîne" ou une String, mais chaine2 est obligatoirement une variable de type String.

Le type Énumération (enum) :

Fondamentalement, un type énuméré sert à générer une famille de constantes relatives à un thème commun avec un identificateur de substitution pour chacune élément de l'ensemble déclaré. Exemple :

```
enum couleur_carte {
  TREFLE, // Par défaut valeur 0.
  CARREAU, // Par défaut valeur 1.
  COEUR, // Par défaut valeur 2.
  PIQUE }; // Par défaut valeur 3.
```

Ils sont équivalents à des entiers valant dans l'ordre 0, 1, 2, 3, 4, 5 etc.

Par exemple COEUR * PIQUE retournera la valeur 6.

Les énumérateurs ont une portée globale ou limitée à un bloc en fonction de leur localisation dans le programme. Il est possible de choisir explicitement les valeurs des constantes d'énumération *(Ou que pour certaines d'entre elles. Dans ce cas, les autres suivent la règle d'incrémentation donnée précédemment)* Exemple :

```
enum couleur_carte {
  TREFLE, // Par défaut valeur 0.
  CARREAU, // Par défaut valeur 1.
  COEUR=45, // Forcé à la valeur 45.
  PIQUE }; // Par défaut valeur 46.
```

Glups !

ATTENTION : Si un NOM ne précède pas le bloc de l'ensemble, le compilateur accepte, mais tous les éléments auront la première valeur.

Définir une énumération améliore significativement la lisibilité du code source. Il est logique de désirer utiliser un type énuméré comme paramètre ou comme résultat d'une fonction. Exemple :

```
enum Couleur{TREFLE=1231,CARREAU,COEUR,PIQUE};
int CARTE;
void setup() {Serial.begin(19200);}
void loop() { CARTE = JEU(TREFLE); // L'appel "donne TREFLE".
   Serial.println(CARTE); // Affiche la valeur retournée par JEU.
   infini: goto infini; }

int JEU(int CARTE) {  // Récupère le paramètre "TREFLE".
   Serial.println(CARTE); // Utilise le paramètre reçu.
   int VALEUR = CARREAU; // Affecte la variable locale VALEUR.
   return VALEUR; } // Retourne la valeur "CARREAU" en sortie de fonction.
```

(Ce programme retourne "1231" puis "1232" sur la ligne série USB)

Les constantes en C d'Arduino :

Incontournable, le mot clé const est un qualificateur qui modifie le comportement de l'identificateur en générant une variable de type "lecture seule". L'entité créée peut être utilisée comme n'importe quelle autre variable du même type, mais sa valeur ne peut pas être changée dans le code du programme. Toute tentative de réaffectation génèrera un message d'erreur. Les constantes définies avec const obéissent aux mêmes règles de portée *(Gestion des types)* que celles qui gouvernent les autres variables. C'est la raison pour laquelle, outre les pièges de l'utilisation de #define, le mot-clé const fournit une meilleure stratégie de programmation pour définir les constantes et sera préféré à #define. On peut utiliser aussi bien const que #define pour créer des constantes numériques ou des chaînes. Mais pour les tableaux, il faut impérativement utiliser l'instruction const.

```
#define pi 3.141592654
const float pi = 3.141592654;
```

} *Sont équivalents. Les deux n'occupent aucune place en mémoire du µP.*

NOTE : Les constantes #define et const peuvent se trouver n'importe où dans le programme, mais doivent être déclarées avant leur utilisation.

La directive #define :

(Voir encadré page 7)

Utile en programmation, cette directive permet au programmeur de donner un nom à une constante numérique avant que le source ne soit compilé. Les constantes ainsi définie ne consomment aucune place en mémoire dans le microcontrôleur. Le compilateur remplacera les références à ces constantes par la valeur définie au moment de la compilation. Par ailleurs, remplacer dans le code une constante par un identificateur peut augmenter la lisibilité d'un programme.

Exemple : `#define LED_Arduino 13 // Broche 13 utilisée.`

Un point-virgule après l'instruction #define ou un "=" entre l'identificateur et sa valeur génèrent une erreur de compilation.

La directive #define peut cependant présenter quelques effets de bord indésirables. Par exemple, un nom de constante qui a été défini par #define est inclus dans d'autres constantes ou nom de variable : Dans ce cas, le texte de ces constantes ou de ces variables sera remplacé par la valeur définie avec #define.

D'une manière générale, le mot clé const est préférable pour définir les constantes et doit être utilisé plutôt que #define.

`chaine1.endsWith(chaine2);`

Teste si un objet String se termine avec les mêmes caractères que tous ceux d'un autre objet String ou chaîne. Retourne true ou false.

`chaine1.startsWith(chaine2);`

Teste si un objet String commence avec les mêmes caractères que tous ceux d'un autre objet String ou chaîne. Retourne true ou false.

`chaine1.lastIndexOf(TEXTE,Position);`

Fonction qui localise un caractère ou une séquence de caractères ASCII dans un objet de type String. Mais effectue la recherche en sens rétrograde et par défaut commence à la fin. Peut aussi débuter une recherche inverse à partir d'une Position *(Option)* donnée, pour rechercher toutes les instances. TEXTE peut être un caractère, une chaîne, ou une String. Retourne **-1** si non trouvé ou sa position.

`chaine.length();`

Fonction qui retourne la valeur de longueur de la chaîne d'un objet String, sans inclure le "nul" marqueur de fin de chaîne.

`chaine1.replace(chaine2,chaine3);`

Fonction qui permet de remplacer dans un objet chaine1 de type String toutes les instances d'une sous-chaîne chaine2 par une autre sous-chaîne chaine3. Pour remplacer toutes les instances d'un caractère il suffit de définir pour chaque sous-chaîne un seul caractère.

`chaine.setCharAt(Position,'caractère');`

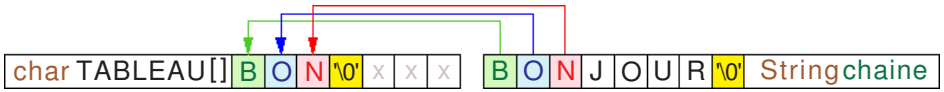
Remplace le caractère d'un objet String dont on précise la Position et le caractère de substitution. Aucun effet sur les emplacements débordant l'espace de l'objet String existant. *(Position > longueur)*

`chaine.substring(Début,Fin);`

Retourne une sous-chaîne d'un objet String depuis le caractère pointé par Début jusqu'à celui pointé par Fin **-1**. Si le pointeur de fin est omis, l'extraction renvoyée s'étend jusqu'à la fin de l'objet chaine.

`chaine.toCharArray(TABLEAU,NB+1);`

Copie les NB premiers caractères d'un objet String chaine dans un tableau de caractères TABLEAU. Il faut donner le nombre augmenté de 1 car le délimiteur '\0' fait partie du bloc recopié. Si le nombre NB dépasse l'étendue possible de TABLEAU il y aura débordement.



... / ...

Fonctions de la classe String.

String(valeur,base); *(base est une option non impérative)*

Ce constructeur crée ou retourne une instance de la classe String qui transforme la valeur en une séquence de caractères. Exemples :

```
String texte = "BONJOUR"; // Utilisation d'une chaîne de caractères.
String TEXTE = String('a');// Conversion d'un caractère en objet String.
String Car = String(char(234));// Conversion d'un caractère spécial. ("ê")
String TEXTE = String(texte + "XXX");// Concaténation String + chaîne.
String TEXTE = String(13);// Conversion d'un entier en base 10 par défaut.
String E5 = String(analogRead(5), DEC); // Conversion d'un int en base 10.
String ADR = String(-18273, HEX);// Conversion d'un int en base 16.
String Octet = String(123, BIN);// Conversion du nombre 123 en base 2.
String Duree = String(millis(), DEC);// Conversion d'un long en base 10.
String LCD[4] = {"","Ligne 2","","Ligne 4"}; // TABLEAU de String.
```

ATTENTION : String TEXTE = ou String TEXTE; déclare la chaîne et peut se situer en global ou en local. Mais une seule instance doit figurer dans le programme. Par la suite seul l'identificateur TEXTE devra être utilisé pour manipuler cette chaîne de caractères.

chaine1.equals(chaine2);

Compare l'égalité totale de deux objets String. Les objets sont comparées caractère par caractère, par leurs valeurs ASCII, la casse est prise en compte ainsi que le nombre de caractères.

chaine1.equalsIgnoreCase(chaine2);

Compare l'égalité de deux objets String. Les objets sont comparées caractère par caractère, par leurs valeurs ASCII, la casse est IGNORÉE mais tient compte du nombre de caractères si ≠.

chaine1.compareTo(chaine2);

Compare deux objets String, testant si l'un vient avant ou après l'autre, ou s'ils sont égaux. Les objets sont comparées caractère par caractère, par leurs valeurs ASCII. Par exemple, 'a' vient avant 'b' mais après 'A'. Les chiffres viennent avant les lettres etc.

chaine1.indexOf(TEXTE,Position);

Fonction qui localise un caractère ou une séquence de caractères ASCII dans un objet de type String. Par défaut, recherche depuis le début de l'objet String à l'indice zéro, mais peut également débuter la recherche à partir d'une Position *(Optionnel)* donnée, permettant la recherche de toutes les instances. TEXTE peut être un caractère, une chaîne, ou une String. Retourne **-1** si non trouvé, ou sa position.

Glossaire.

AVR : Nom de la famille des microcontrôleurs à laquelle appartiennent l'ATmega328P de la carte "UNO" ou le 2560 du modèle "Mega".

Bibliothèque : Ensemble de procédures utilitaires, regroupées et mises à disposition des utilisateurs d'Arduino.

Segment BSS : En informatique, ce sigle est utilisé par les compilateurs et les éditeurs de liens pour désigner une partie du segment de données contenant les variables statiques déclarées et initialisées à zéro au démarrage du programme. Historiquement, **B**lock **S**tarted by **S**ymbol était une étape en assembleur *(UA-SAP)* développé dans les années 1950 pour l'IBM 704.

IDE : *(Integrated development environment)*
L'environnement de développement intégré *(IDE)* est un programme regroupant un ensemble d'outils pour la création de logiciels. Il regroupe un éditeur de texte, un compilateur, des fonctions automatiques et souvent un débermineur.

Librairie : Anglicisme pour Bibliothèque. *(Voir Bibliothèque)*

Shield : Carte comprenant un circuit complexe qui se connecte directement à l'Arduino et qui assure une fonction spécifique.

Sketch : programme dans le langage Arduino.

void : Mot clef utilisé pour spécifier qu'il n'y a pas de variable ou de valeur. On le retrouve généralement dans le préfixe d'une procédure ou dans la liste éventuelle des paramètres pour indiquer que celle-ci ne renvoie pas de paramètre. *(Ou qu'elle n'en reçoit pas)*
EXEMPLES pour lesquels on précise qu'il n'y a pas de paramètre :

```
void setup(void)
void loop(void)
void Afficher_Date_et_Heure(void)
```

"Bug et aléas sur #define".

Attention : On peut faire suivre une définition d'un commentaire de type */ xxx */ mais il ne faut pas que ce dernier commence à la ligne du #define et se termine sur une autre ligne, ou le compilateur va générer une cascade d'erreurs incompréhensibles.

La structures de contrôle if / else :

Sous sa forme la plus complète l'instruction s'écrit :
if (Condition) {Instructions} else {Instructions}
(Le ; n'est pas utile après un bloc de type {})

- Condition est une expression booléenne.
- L'instruction else est optionnelle.

Voir également l'opérateur ternaire "?" en page 33.

Opérateurs logiques de comparaison :

X == Y (X est égal à Y)
X != Y (X est différent de Y)
X < Y (X est inférieur à Y)
X > Y (X est supérieur à Y)
X <= Y (X est inférieur ou égal à Y)
X >= Y (X est supérieur ou égal à Y)

ATTENTION : Prendre garde à ne pas utiliser accidentellement le signe = unique **qui est l'opérateur d'affectation d'une valeur à une variable**. La condition est alors toujours vraie. Le test est faussé et la variable est modifiée intempestivement.

Conditions composées :

Imposer des conditions composées est possible :

```
if (Test1 && Test2 && Test3) {Action();} // Trois tests en chaîne.
```

Imposer une condition négative impose de parenthèser :

```
if (Test1 && ! (Test2) || Test3) {Action();} // Trois tests en chaîne.
```

Opérateurs logiques de combinaisons conditionnelles :

Ces opérateurs peuvent être utilisés à l'intérieur de la condition d'une instruction if pour associer plusieurs conditions (ou opérandes) à tester.

(Condition1 && Condition2) : ET logique.
(Condition1 | | Condition2) : OU logique.
! (Condition) : NON logique >>> true si l'opérande est false.

ATTENTION : Il importe de ne pas confondre &&, l'opérateur ET utilisé dans les expressions booléennes des expressions if, avec l'opérateur logique & utilisé avec des masques logiques dans les instructions bit à bit. *(Voir page 31)*
De la même façon, ne pas confondre | |, l'opérateur OU dans les expressions booléennes des expressions pour if, avec l'opérateur | pour le OU utilisé dans les instructions spécifiques bit à bit.

Exemple : Programme ESSAI_sur_if_et_else.ino

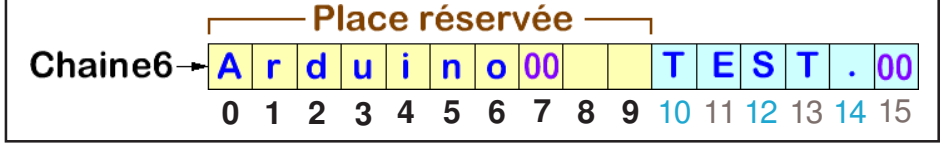
de caractères. Comme les chaînes de caractères sont elles-mêmes des tableaux, c'est un cas particulier de tableau à deux dimensions.

```
char *Tableau_de_textes[]={"Chaine de caracteres 1", "Bonjour.",
"Ceci est la chaine 3", "Et de 4", "5 !",">FIN<"};

void setup(){ Serial.begin(19200);
  for (int i = 0; i < 6; i++)
      { Serial.println(Tableau_de_textes[i]); } }

void loop(){}
```

Dans le code listé ci-dessus, l'astérisque après la déclaration de donnée de type char * indique qu'il s'agit d'un tableau de pointeurs. *(Voir p13)*

Modifier un tableau char élément par élément.

La chaîne sera complétée caractère par caractère en précisant la position. L'indice doit varier entre 0 et la longuer de la chaîne. *(Longueur de texte délimiteur non compris)* Attention, si on place un caractère sur le délimiteur nul, la chaîne va se prolonger jusqu'au prochain nul. **Si l'indice est "hors calibre", le caractère sera placé plus avant dans la mémoire et écrasera l'octet qui s'y trouve. Le compilateur ne vérifie pas la validité des indices.** Un débordement peut engendrer un comportement aléatoire du programme si le caractère vient écraser "du code exécutable".



La classe String() :

Intégrée dans la version 0019 du logiciel Arduino, la classe String() permet d'utiliser et de traiter des chaînes de caractères dans des utilisations plus complètes que ce que permettent les chaînes de caractères simples. Par exemple on peut concaténer des String, y ajouter du texte, chercher ou remplacer des parties etc. Leur utilisation occupe plus de place en mémoire qu'une simple chaîne de caractère, mais la programmation en est bien plus aisée pour certaines fonctions.

NOTE : Les chaînes de caractères constantes du type "Texte libre" sont prises en charge comme des **tableaux de caractères**, et non gérées comme des instances de la classe String.

... / ...

Les chaînes de caractères :

Fondamentalement les chaînes de caractères sont instanciées sous forme de **tableaux** de variables de type char et se terminent par un octet nul. Plusieurs syntaxes de déclaration de chaînes sont possibles :

- Déclarer un tableau de caractères sans l'initialiser.
  `char Chaine1[15] ; // Ici pour 14 caractères plus le nul.`
- Déclarer un tableau de caractères et laisser le compilateur ajouter lui même l'octet nul requis *(Sentinelle)* :
  `char Chaine2[8] = {'A', 'r', 'd', 'u', 'i', 'n', 'o'} ;`
- Déclarer explicitement le caractère nul :
  `char Chaine3[8] = {'A', 'r', 'd', 'u', 'i', 'n', 'o', '\0'} ;`
- Initialiser avec une chaîne de caractères constante entre guillemets, doubles : Le compilateur dimensionnera le tableau pour stocker la chaîne de caractère constante et ajoutera le caractère nul :
  `char Chaine4[ ] = "Arduino" ;`
- Initialiser le tableau avec une taille explicite et une chaîne de caractères qui restera constante en taille dans le programme. Exemple :
  `char Chaine5[8] = "Arduino" ;`
- Déclarer un tableau plus large, en le prévoyant pour une chaîne plus étendue que celle de l'initialisation. Exemple :
  `char Chaine6[10] = "Arduino" ;`
  (Pour compléter cette chaine voir l'encadré page 41)

Les délimiteurs :

Les chaînes de caractères sont toujours définies entre des guillemets *("ABC")* et les caractères sont toujours définis entre deux cotes. *('A')*

Déclaration de longues chaînes de caractères :

Il est possible pour des raisons de lisibilité du source de scinder de longues chaînes de caractères de la façon suivante :

```
char LongueChaine[] = "Ce long texte ne va " "constituer"
" dans le programme qu'une"" seule et unique chaine";
```

Tous les "blocs" de texte sont encadrés par des guillemets et se suivent sans virgule de séparation. Les espaces ne sont pas obligatoires. C'est le point virgule qui achève la déclaration complète.

Tableaux de chaînes de caractères :

Il est pratique de déclarer plusieurs textes, par un tableau de chaînes

Si le bloc d'instructions ne comporte qu'une seule instruction il n'est pas obligatoire de placer les accolades pour délimiter le bloc.
if (Condition) *Instruction_Sans_Acolade*;

Structures If /else en cascade :

Le bloc relatif à else peut contenir un autre test if, et donc des tests multiples **mutuellement exclusifs** peuvent être mis en cascade. Chaque test sera réalisé après le précédent jusqu'à ce qu'un test true soit rencontré. Quand une condition true est rencontrée, les instructions associées sont réalisées, puis le programme saute son déroulement à l'instruction suivant l'ensemble de la structure if / else. Si aucun test n'est true, le bloc d'instructions par défaut else sera exécuté, **s'il est présent dans le code**, déterminant ainsi le comportement par défaut :

⇦⇦⇦⇦ Au premier true rencontré {Ai} est réalisé puis "sortie".

```
if (true) {A1;} else if (true) {A2;} else if (true) {A3;} else {A4;}
if (false) {A1;} else if (true) {A2;} else if (true) {A3;} else {A4;}
if (false) {A1;} else if (false) {A2;} else if (true) {A3;} else {A4;}
if (false) {A1;} else if (false) {A2;} else if (false) {A3;} else {A4;}
```

NOTE : Le else if ne peut pas être utilisée seul, il faut un if avant.
Une autre façon plus lisible de réaliser des branchements de tests multiples mutuellement exclusifs consiste à utiliser switch case.

La structures de boucle while : *(While : Tant que)*

while(Condition){Action();}

La structure while boucle en interne jusqu'à ce que la Condition ou l'expression booléenne entre les parenthèses devienne false. Condition doit se modifier sinon la boucle ne se terminera jamais. *(Voir l'exemple de boucle infinie tout en haut de la page 12)* Condition est une expression booléenne qui retourne la valeur true ou false. Si en entrée dans la structure Condition = false, l'Action(); n'est jamais réalisée.

La structures de boucle do / while : *(Do : Faire)*

do {Action();} while(Condition);

La boucle do / while fonctionne exactement comme la boucle while simple, mais la condition n'est testée qu'après avoir exécuté le bloc d'instruction {Action();}. Ce dernier sera donc exécutée au moins une fois quelles que soit l'état de la Condition en entrée de structure.
NOTE : Le ";" est impératif puisque la ligne de code se termine par une instruction de test et non un bloc placé entre "{" et "}".

... / ...

Incidence de l'ordre des opérations composées :

L'ordre dans lequel est effectué l'incrémentation par rapport à la variable de test doit être pris en compte dans les structure de boucles de type while ou celle de type do / while.

EXEMPLE :

```
I = 0;                                  ➢➢➢
do {Serial.print(X);} while (X++ < 5); // Affiche 012345.
I = 0;                                  ➢➢➢
do {Serial.print(X);} while (++X < 5); // Affiche 01234.
```

Dans le premier essai le test est effectué sur X puis il est incrémenté.
Dans le deuxième exemple X est incrémenté, puis il y a le test. Donc on sort de la boucle une passe avant celle du premier exemple.

La structures de boucle for :

```
for (Initialisation; Condition; Incrémentation) {Action();}
```

L'instruction for est utilisée pour répéter l'exécution d'un bloc d'instructions regroupées entre des accolades. Un compteur incrémental est habituellement utilisé pour faire évoluer et se terminer la boucle. L'instruction for est très utile pour toutes les opérations répétitives et est souvent utilisées en association avec des tableaux de variables pour agir sur un ensemble de données ou de broches.
L'Initialisation a lieu en premier et une seule fois en entrée dans la structure. À chaque exécution de la boucle, la condition est testée; si elle est true, le bloc d'instructions {Action();} et Incrémentation sont exécutés. Puis la condition est testée de nouveau. Quand Condition devient false, il y a sortie de la structure et saut au code suivant.
La boucle for en C est beaucoup plus flexible que généralement dans les boucles des autres langages de programmation. Tout ou partie des trois élément de l'entête de la boucle peuvent être omis, les points-virgules ";" sont toutefois obligatoires. De plus, le codage pour Initialisation, Condition ou Incrémentation peut être n'importe quelle instruction valide en langage C, avec des variables quelconques non liées.

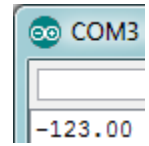
Exemple dans lequel l'initialisation se fait hors structure et l'évolution de la variable de test dans le bloc d'instructions :

```
byte  = 5;
for (; X < 8;) {Serial.print(X); X++;} // Affiche 567.
```

Exemple : Programme ESSAI_sur_for_et_sur_while.ino

La conversion s'arrête au premier caractère invalide autre que des chiffres et le point. Exemple :

```
FLOAT = atof("-123A4567.12"); Serial.println(FLOAT);
```



Convertion valeurs numériques / chaîne de caractère :

Convertir la valeur d'une variable numérique de type byte, int, long, unsigned int et unsigned long est immédiat par usage d'une instance String() comme développé en page 41. Par contre, pour transformer en chaîne de caractère la valeur d'un float la technique est plus compliquée.

Convertir un entier en chaîne de caractères.

Il suffit d'affecter la variable dans une instance de type String :

```
long VALEUR = -21474836;

void setup() { Serial.begin(115200); Serial.begin();}

void loop() {
  VALEUR = VALEUR * 3; // Petit calcul sur le nombre entier.
  String TEXTE = String(VALEUR); // Transformation en chaîne.
  Serial.println(TEXTE); Serial.println(TEXTE[3]);
  INFINI: goto INFINI; }
```

Convertir un float en chaîne de caractères.

à déterminer ...
Convertir du décimal en string :
http://forum.arduino.cc/index.php?topic=228246.0

Décodage de valeurs représentées en BCD :

Un nombre codé en BCD, par exemple sur un octet, loge les dizaines *(Exprimées en décimal)* sur les quatre bits de poids forts, et les unités sur les quatre bits de poids faibles.
0xF & OctetBCD; Isole les 4 bits de poids faibles.
Pour décoder les 4 bits de poids fort :
(10 * (OctetBCD >> 4)) + (0xF & OctetBCD);

NOTE IMPORTANTE sur le CAN de l' ATmega328.

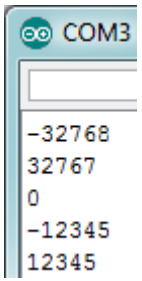
Tout convertisseur analogique / numérique présente une imprécision propre qui en limite le nombre de chiffres significatifs fiables. Le convertisseur de l'ATmega328 est très linéaire, sauf pour les deux bits de poids faibles. Sa précision est de ±2LSB. *(± 4 unités)*

Convertion chaînes de caractères / valeurs numériques :

Savoir transformer une chaîne de caractère en un nombre pouvant servir dans des fonctions de calcul est un impératif, tout particulièrement quand les données sont issues de la ligne série qui ne délivre que "du texte". Dans la chaîne de caractères à convertir en nombre, le signe + en tête n'est pas impératif.

Convertir une chaîne en un int : atoi.

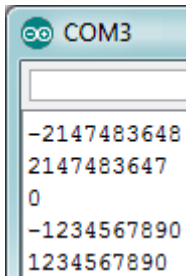
```
int INT;
void setup() {Serial.begin(19200);}
void loop(){
  INT = atoi("-32768"); Serial.println(INT);
  INT = atoi("+32767"); Serial.println(INT);
  INT = atoi("0"); Serial.println(INT);
  INT = atoi("-12345"); Serial.println(INT);
  INT = atoi("+12345"); Serial.println(INT);
```



ATTENTION : Il n'y a pas de vérification sur la valeur de la donnée chaîne qui doit rester dans les limites d'un int où le résultat sera incorrect.

Convertir une chaîne en un long: atol.

```
long LONG;
void setup() {Serial.begin(19200);}
void loop(){
  LONG = atol("-2147483648"); Serial.println(LONG);
  LONG = atol("2147483647"); Serial.println(LONG);
  LONG = atol("0"); Serial.println(LONG);
  LONG = atol("-1234567890"); Serial.println(LONG);
  LONG = atol("+1234567890"); Serial.println(LONG);
```

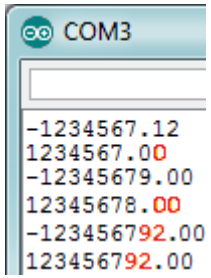


ATTENTION : Il n'y a pas de vérification sur la valeur de la chaîne qui doit rester dans les limites d'un long où le résultat sera incorrect.

Convertir une chaîne en un float : atof.

Accepte aussi un double, mais dans les deux cas quand on dépasse les huit chiffres significatifs la conversion n'est plus parfaite.

```
FLOAT = atof("-1234567.12"); Serial.println(FLOAT);
FLOAT = atof("+1234567.091"); Serial.println(FLOAT);
FLOAT = atof("-12345678.9845"); Serial.println(FLOAT);
FLOAT = atof("+12345678.4321"); Serial.println(FLOAT);
FLOAT = atof("-123456789.3333"); Serial.println(FLOAT);
FLOAT = atof("+123456789"); Serial.println(FLOAT);
```



Exemple : Programme Conversion_Chaine_Nombre.ino

La structures de contrôle switch / case :

L'instruction switch / case permet au programmeur de construire une liste de "cas" *(Possibilités)* définis à l'intérieur d'un "bloc" d'instructions. Le programme compare chaque cas avec la variable de test, et exécute le code si une correspondance true est détectée. Quand un true est rencontré, son bloc de code est exécuté. Mais on peut décider de sortir de la structure avec l'instruction break, ou de continuer à comparer les autres cas si cette instruction n'est pas insérée dans le bloc à exécuter.

```
switch (Var) { // Début de la structure
  case Val1 : {Instructions; break;}  ➢ Sortie de la structure.
  case Val2 : {Instructions;}
  case Val2 : Instruction; break;     ➢ Sortie de la structure.
  case Val2 : {Instructions; break;}  ➢ Sortie de la structure.
   ...
  defalt : {Instructions; break;} // Option non impérative.
   } // Fin de la structure.
```

Si aucune condition true n'est rencontrée lors du balayage, le code defalt sera exécuté si cette instruction est présente dans la structure. Pour éviter des comportements inattendus, *(Valn étant modifiée dans les Instructions;)* il est recommandé de mettre une instruction break à la fin du code de chaque Valn analysée. Il n'y a que dans le cas de conditions imbriquées entre-elles que l'on pourra ne pas mettre le break. Les Valn de comparaison peuvent être ordonnée de façon quelconque.

L'instruction de contrôle continue :

L'instruction continue est utilisée pour passer outre certaines portions de code dans les boucle do, for ou while. La condition est toujours évaluée, mais le code restant dans le bloc est ignoré. Exemple :

```
int X = 0;

while (X <= 999) {X++; if (X > 3) {continue;} Serial.print(X);}
Serial.println(); Serial.print(X); (1)
```

Dans cet exemple le programme ne va afficher que 1, 2et 3. Zéro n'est pas affiché car l'incrémentation de X est effectuée avant son affichage. À partir de X = 4, continue fait ignorer la suite du bloc et il y a nouveau test sur la variable X. L'instruction (1) permet de vérifier l'achèvement de la boucle. Noter que la valeur finale est X = 1000 et non 999.

Le goto en langage C :

Bien que le goto ne soit pas très recommandé sachant qu'il peut conduire facilement à des programmes délicats à déverminer, car il devient difficile de suivre de cheminement lors du déroulement des instruction, un Goto peut parfois s'avérer utile.

SYNTAXE :

- Nométiquette suivi de "**:**" sans espace pour créer le pointer du Goto.
- Goto NomÉtiquette; pour imposer le saut inconditionnel.

Exemple simple :

```
INFINI: goto INFINI; // Façon de remplacer while(true);
```

Les tableaux de variables :

Les tableaux sont des collections de variables de types identiques, situées dans un espace contigu dans la mémoire et accessibles à l'aide d'un numéro d'index. *(De type byte, char, int,long, double…)*

Les tableaux à une dimension.

Diverses façons de déclarer un tableau à une dimension :

- Ne déclarer que la taille pour que le compilateur lui réserve la place :

```
int Tableau[6]; // 6 emplacements indexés de 0 à 5.
```

- Initialiser les données sans préciser la taille. Le compilateur compte alors les éléments et crée un tableau de la taille appropriée :

```
char Car_valides[] = {'S', 'a', 'N'}; // 4 octets car délimiteur '\0'.
```

- Déclarer la taille et initialiser les données :

```
byte Filtres[5] = {2, 4, -8, 3, 2}; // 5 octets indexés de 0 à 4.
```

- Cas particulier des caractères et des chaînes : Il faut prévoir pour la taille déclarée l'emplacement du délimiteur '\0' :

```
char Message[8] = "Bonjour"; // 7 caractères plus le '\0'.
```

Accéder aux éléments d'un tableau.

```
X = Tableau[2]; // X prend la valeur du troisième élément.
Tableau[0] = X; // Le premier élément prend la valeur de X.
```

Le premier élément d'un tableau est indexé avec zéro.
Le dernier élément d'un tableau est indexé avec [Dim] - 1 ou par la valeur [Dim] - 2 si c'est un tableau de type chaîne ou de caractères.

ATTENTION : Le compilateur ne teste pas le débordement d'indice. Lire hors tableau retourne une valeur quelconque. Écrire hors limites écrase du code où une valeur du logiciel et conduit à un comportement imprévisible du programme.

Exemple : Programme Travail_sur_les_tableaux.ino

Allocation dynamique de la mémoire.

La réservation dynamique de mémoire SRAM passe par l'utilisation d'un pointeur. On peut procéder en deux étapes : Déclarer le pointeur, puis réserver la zone mémoire avec ce dernier. Exemple :

```
long int *PTR = NULL;// Déclaration du pointeur.
```

Puis utiliser le pointeur, par exemple dans la boucle de base :

```
PTR = (long int*) malloc(NB_OBJETS *sizeof(long int));
      // Allocation de NB_OBJETS.
```

Ou utiliser une instruction unique directement dans la boucle de base :

```
long int *PTR = (long int*) malloc(NB_OBJETS *sizeof(long int));
```

Le seul paramètre à passer à malloc() est le nombre d'éléments typés à allouer. La **fonction** malloc() retourne comme valeur l'adresse du premier octet de la zone mémoire allouée.

Si l'allocation n'a pu se réaliser par manque de mémoire disponible, la valeur retournée est la constante NULL.

Libération de la zone réservée en mémoire dynamique.

La libération de la mémoire précédemment allouée via malloc() se fait avec la **procédure** free() dont la syntaxe n'exige comme paramètre que l'identificateur du pointeur utilisé pour réserver la zone allouée :

```
free(PTR); // Libère tout l'espace réservé avec PTR.
```

Initialisation d'une zone de mémoire.

```
memset(PTR,Valeur,NB_OCTETS);
```

Procédure qui permet de remplir une zone mémoire avec des Valeurs désirées. Par exemple pour vider le contenu d'une chaîne de caractères en forçant tout ses octets à 0, remplir un tableau avec des "1" etc.

Le pointeur PTR précise l'adresse de la première cellule mémoire à initialiser. Valeur est l'octet de remplissage en hexadécimal, décimal, un code ASCII, 'caractère'. NB_OCTETS indique le nombre de cellules mémoire à initialiser avec Valeur. Noter que **cette procédure ne fonctionne qu'avec des octets**, donc le pointeur sera de type byte.

```
sizeof(Variable)
```

Cette fonction retourne le nombre d'octets constituant une variable, ou le nombre d'octets occupés par un tableau.
Pour l'argument, n'importe quel type de Variable ou de tableau peut être précisé : int, float, byte, char etc ...

Exemple : Programme Tester_memset_initialisation_sone_SRAM.ino

Allocation dynamique de la mémoire :

malloc() est en informatique une fonction de la bibliothèque standard stdlib.h permettant d'allouer dynamiquement de la mémoire. Cette bibliothèque est fournie en standard pour le langage C. Elle est implicitement déclarée dans Arduino, de ce fait une instruction #include <stdlib.h> n'est pas impérative.

L'allocation dynamique de la SRAM.

Elle peut s'effectuer de trois façons :

- Statiquement, au cours de la compilation par la déclaration de variables statiques : Variables globales ou variables locales déclarées en utilisant le mot-clé static,
- Dynamiquement, au cours de l'exécution :
  - * Soit de façon automatique sur la pile d'exécution : Variables locales déclarées dans un bloc d'instructions,
  - * Soit à la demande du programmeur, sur le TAS, en utilisant des fonctions spécifiques d'allocation de la mémoire.

L'allocation statique oblige le développeur à connaître à l'avance la quantité de mémoire qui sera utilisée. Un "gaspillage" de mémoire peut survenir si l'on réserve trop de mémoire par rapport au besoin réel du programme. Avec la réservation automatique, la libération de la mémoire n'est réalisée qu'à la fin du bloc d'instructions dans lequel est déclarée la variable, ce qui peut également être un facteur de gaspillage lorsque cette mémoire restant allouée, elle n'est plus utilisée.

C'est dans ce cas typique que l'allocation dynamique de mémoire fournit une solution élégante au programmeur. La réservation de la mémoire se fait au cours de l'exécution du programme ; la libération de cette mémoire n'est plus gérée par le compilateur, mais à convenance par le programmeur. La complexité du programme augmente, mais la gestion de la mémoire est plus fine. Si un programme alloue de la mémoire avec malloc() sans la libérer ensuite avec l'instruction free(), on utilise le vocable "fuite de mémoire". Pour éviter ce type d'inconvénient et faciliter l'écriture des programmes, certains langages disposent d'un mécanisme nommé "Ramasse-miettes". Ce n'est pas le cas pour le langage C pour lequel la gestion dynamique de la mémoire est entièrement laissée à la charge du programmeur.

Bien que malloc() fasse partie du C, il est très peu conseillé de l'utiliser avec des microcontrôleurs car sont effet peut s'avérer imprévisibles.

Exemple : Programme Tester_malloc_et_free.ino

Exemple : Programme Travail_sur_tableau_a_trois_dimensions.ino

Les tableaux à plusieurs dimensions.

Leur déclaration est identique à celle d'un tableau à une dimension, sauf qu'il faut préciser autant de crochets que l'on désire de dimensions. Tous les éléments sont de même type. Exemple simple :

```
char TEXTES[5][5] = {"AAAA","BBBB"} ; // Deux dimensions.
```

Exemple avec trois dimensions :

```
char TROIS_DIM[4][3][2] = { (1)
 {{'A','B'},{'C','D'},{'E','F'}}, // Première ligne. (n°0)
 {{'G','H'},{'I','J'},{'K','L'}},  // Deuxième ligne. (n°1)
 {{'M','N'},{'O','P'},{'Q','R'}}, // Ligne n°2
 {{'S','T'},{'U','V'},{'W','X'}}, // Dernière ligne. (n°3)
 }; // Fin de déclaration / initialisation du tableau.
```

(1) : Les dimensions d'un tableau sont obligatoirement déclarées comme des constantes, ou le compilateur génère des erreurs. Il n'est donc pas très utile d'employer des identificateurs. Par contre, l'accès aux éléments peut se faire avantageusement avec des identificateurs :

```
byte LIGNE, COLONNE, ELEMENT;
LIGNE = 3; COLONNE = 2; ELEMENT = 1; // Pointe le 'X'.
Serial.println(TROIS_DIM[LIGNE][COLONNE][ELEMENT]);
```

Initialisation des tableaux.

Toutes les valeurs non-précisées pour un tableau sont automatiquement mises à zéro. On peut procéder à une mise à zéro de tout un tableau très rapidement et facilement en ne précisant que sa première valeur :

```
int Tableau[2][3][4] = { { {0} } };
```

En C mais pas en C++.

Déclaration incomplète.

Il est possible lors de la déclaration de ne pas préciser la taille de la première dimension. Par exemple ce code est parfaitement valide :

```
int Tableau[][3] = { {0,1, 2, 3, 4, 5}, {6, 7, 8} };
```

Pour cet exemple, la première dimension vaudra 6. Par contre ce n'est accepté que pour la première dimension. Si la taille des autres dimensions n'est pas précisée, le compilateur génère une erreur.

Tableau de pointeurs.

Il est parfaitement possible de déclarer un tableau de pointeurs :

```
int * ADRESSES[7];
```

*Cette instruction créé un tableau de pointeurs et non un pointeur sur le tableau ADRESSES car l'opérateur [] est prioritaire sur *. Le compilateur évalue donc l'instruction comme étant un tableau contenant des pointeurs.*

Les fonctions avec passage de paramètres :

Fonction et procédures sont deux structures assez analogues. Toutes deux effectuent un traitement. Mais la fonction retourne un paramètre dont le type doit être déclaré en entête. Comme pour les procédures une fonction peut manipuler des variables globales, mais en général elle utilise le passage de paramètres. L'appel à une fonction peut se faire à la place de tout paramètre dont elle respecte le type. Le programme Fonction_avec_paramètres.ino est un exemple qui utilise plusieurs variantes de fonctions avec ou sans paramètres.

type NomFonction(type Param1, type Param2 ...)
{Instructions;return Resultat} // Les noms ParamN sont optionnels.

La structured'une fonction commence par la déclaration de son type qui peut être quelconque : ent, string, boolean etc. Entre parenthèses sont listées les paramètres dont le type doit également être déclaré. L'intégralité du traitement de la fonction doit être placé entre les deux accolades d'un bloc { et }, y compris le return. Resultat peut être aussi-bien l'identificateur d'une variable qui a été traitée qu'une instruction de traitement. La valeur doit être compatible avec le type.

Une fonction peut parfaitement travailler exclusivement sur des variables externes. Si elle ne doit pas utiliser de paramètre, les parenthèses seront "vides" mais leur présence est obligatoire. Exemple :

```
float X = -123.456; byte Y = 2;
void loop() { Serial.println(Division(),6); }
float Division(void) {return X / Y ;}
```

Exemple : Programme Fonctions_avec_parametres.ino

La portée des variables :

Une variable globale est une entité qui occupe un emplacement durable et figé dans la mémoire, et qui peut être "vue" et utilisée par n'importe quelle fonction ou procédure du programme. Les variables locales ne sont visibles que par la fonction dans laquelle elles ont été déclarées. Elles sont crées au moment de leur utilisation dans l'espace disponible en RAM, puis leur emplacement est libéré en sortie de procédure ou de fonction. Dans le langage Arduino, toute variable déclarée en dehors du corps d'une procédure ou d'une fonction *(Corps délimité par { })* est une variable globale. Il est fortement recommandé d'utiliser des variables locales, donc déclarées à l'intérieur du corps des procédures ou des fonctions, pour éviter les effets de bord, ceci tout particulièrement pour les indice d'évolotion des boucle de type for.

detachInterrupt(NumINT);

Instruction qui libère la broche del'interruption externe NumINT. Seule la broche liée à NumINT ne provoquera plus d'interruptions. l'instruction attachInterrupt() permet d'en rétablir le fonctionnement, ou de la réorienter sur une autre routine de traitement d'interruptions.

noInterrupts(NumINT);

Instruction qui désactive toutes les interruptions. **Les interruptions** permettent à certaines tâches importantes de survenir à tout moment, et **sont activées par défaut avec Arduino**. Plusieurs procédures et fonctions sont désactivées ou perturbées lorsque les interruptions sont déclenchées. *(Communications série, millis, delay etc)* Les interruptions pouvant perturber le déroulement du code dans certaines sections critiques, noInterrupts permet de suspendre en bloc et momentanément toutes les interruptions de l'ATmega328.

interrupts(NumINT);

Instruction qui réactive les interruptions après qu'elles aient été désactivées par noInterrupts. *Naturellement, la paire d'instruction de désactivation et de rétablissement des interruptions peut se trouver dans toute procédure ou fonction du programme.*

```
void loop() {
 "code courant" ...
 noInterrupts(); // Désactivation des interruptions.
 // Code critique sensible aux fonctions temporelles.
 interrupts(); // Réactivation des interruptions.
 Suite du "code courant" ... }
```

Quelques conseils sur l'utilisation des interruptions.

- L'usage des interruptions est idéal pour des tâches qui consistent à surveiller les touches d'un clavier, la détection d'un changement d'état rapide et aléatoire sur un capteur etc, sans avoir pour autant à constamment surveiller l'état de la broche d'entrée concernée.
- Idéalement, une fonction attachée à une interruption doit être la plus courte et la plus rapide possible. Une bonne pratique consiste à s'en servir pour intercepter un événement aléatoire et positionner sa valeur dans une variable globale déclarée en type volatile. Le traitement de cette variable est ensuite confié à la boucle de base ou à l'une de ses procédures de servitude.

Gestion des interruptions :

Le microcontrôleur ATmega328 dispose de deux entrées externes d'interruptions sur les broches spécialisées 4 et 5 de son boitier, respectivement nommées INT 0 et INT 1. Ces deux broches sont reliées respectivement aux entrées binaires n°2 et n°3 sur la carte électronique Arduino.

NOTE : Les ATMega328 peuvent également gérer des interruptions de changement d'état sur vingt de leurs broches. Mais leur traitement n'est pas aussi simple que pour les interruptions externes car il faut aiguiller le code après avoir déterminé d'ou vient le requête. La librairie Arduino arduino-pinchangeint.h a été développée afin de permettre l'utilisation de ces interruptions internes au µP.

attachInterrupt(NumINT,Procedure,MODE)

Instruction d'initialisation qui configure l'une des deux interruptions externes. Cette instruction spécifie la Procedure *(Sans parenthèses pour la déclaration d'appel mais avec pour son code)* à appeler lorsqu'une interruption externe est détectée sur NumINT. Elle remplace tout autre procédure qui était attaché à cette interruption. Le paramètre MODE précise l'événement qui déclenchera les interruptions :

- LOW : Un niveau "0" sur la broche.
- CHANGE : Un changement d'état sur la broche.
- RISING : Une transition de "0" vers "1". *(Un front montant)*
- FALLING : Une transition de "1" vers "0". *(Un front descendant)*

ATTENTION : La routine de service d'interruption Procedure ne peut recevoir aucun paramètre, ne doit traiter que des variables de type volatile et ne peut retourner aucune valeur. *(Routine ISR)*

IMPORTANT : Procedure attachée à l'interruption étant basées sur l'utilisation des timers du microcontrôleur, la routine delay ne fonctionne pas et la valeur renvoyée par millis ne s'incrémente pas durant le code de cette dernière. Les données séries reçues pendant l'exécution de la Procedure sont perdues et le protocole I2C est également affecté.

NOTE : *Les variables volatile sont toute déclarées en GLOBAL et peuvent être modifiées par le programme de base et les subroutines.*

Les expressions numériques entières :

Les expressions numériques entières sont des nombres utilisés directement dans un programme, sous forme de constantes explicitées. Par défaut les expressions numériques entières sont traitées comme des nombres entiers exprimés en base 10. Mais avec certains **préfixes** il est possible d'imposer d'autres bases. Dans les exemples de syntaxe qui suivent, la variable X a été déclarée en int.

X = 128; // Valeur décimale par défaut donc vaut 128_{10}.
X = B1011; // Valeur exprimée en BINAIRE. (Donc 11_{10}.)

NOTE sur le format BINAIRE : Une expression binaire ne peut manipuler que des OCTETS, donc huit bits au maximum. (On peut écrire les zéros en tête mais ce n'est pas impératif) Il suffit pour tourner cette difficulté d'utiliser une "macro instruction" avec poids fort et poids faible différenciés. Penser à parenthèser la multiplication.
~~X = B100000001;~~ n'est pas acceptée, car elle dépasse huit BITs.
X = (B11 * 256) + B01; permet de traiter 16 BITs et sera bien interprétée comme valant 769 en décimal..

X = 0111; // Un zéro en tête impose un format OCTAL.

ATTENTION : Dans l'exemple ci-avant on pourrait lire 111 pour la valeur de X. Mais exprimée en OCTAL, en réalité elle sera interprétée comme valant 73 en décimal. Inclure par inadvertance un zéro en tête dans une expression numérique conduit à un leurre très délicat à repérer.

X = 0xA3; // Valeur exprimée en HEXADÉCIMAL. (163_{10}.)
Attention >>> Il faut un zéro dans 0x et non la lettre "O" !
X = (B10000001 * 256) + 255; // Valeur -32257 car le bit de poids fort à "1" est le bit de signe négatif sur un int.

Par défaut, ces nombres sont traités comme des variables de type int, donc des nombres signés exprimés en "complément à deux". Mais il suffit d'en changer le type. Par exemple utiliser unsigned pour X donnera 33269 pour la même expression. Dans les documents en ligne il est précisé qu'il est possible de changer le type d'une expression avec les **suffixes** "U" pour unsigned et "L" pour long. Mais ce code ne semble plus avoir cours, donc autant déclarer le type désiré pour X.

Les expressions numériques "floating point" :

Comme pour les expressions numériques entières, les expressions numériques **à virgule flottante** sont utilisées pour preciser des constantes numériques de type float. Plusieurs formats sont possibles pour soumettre ces valeurs au compilateur qui les traduira en binaire :

```
float X = .005;  // Les zéros en tête ne sont pas impératifs.
X = 000.00005;  // Les zéros en tête sont ignorés.
X = 10.0;       // Les zéros en queue sont ignorés.
X = 2.34E5;     // Exposant positif.
X = 1.234567e-12; // Exposant négatif.
X = -1.2345E5;    // Exposant positif et mantisse négative.
X = -123.45E2;  // Mantisse avec plusieurs chiffres avant la ",".
```

Pour exprimer l'exposant on peut utiliser aussi bien un "E" qu'un "e"; par contre **il ne faut pas d'espace entre la mantisse et l'exposant**.

Opérateurs arithmétiques.

Variable = Valeur_Numérique; *(Voir l'encadré rouge en page 8)*

L'opérateur d'assignation= impose au programme d'évaluer la valeur de la constante ou de l'expression numérique et de la stocker par écrasement dans Variable. L'entité Variable à gauche de = doit pouvoir "contenir" la valeur calculée par l'instruction. Si sa taille déclarée est insuffisante la valeur stockée sera incorrecte ou tronquée.

Variable = Valeur1 Opérateur Valeur2;

Les Opérateurs **+**, **-**, *, / retournent respectivement la somme, la différence, le produit ou le quotient effectuée sur les deux opérandes Valreu1 et Valeur2 dans Variable par écrasement.

- L'opération est réalisée en utilisant le type des données des opérandes. *(Par exemple, 9 / 4 donne 2 si 9 et 4 sont de type int)* L'opération risque de déborder si le résultat est plus grand que le type déclaré pour les données. Dans ce cas le résultat est alors erroné.
- Si les opérandes sont de deux types de donnée différents, la taille du plus "grand" est utilisée pour effectuer le calcul.
- Si l'un des opérandes est du type float ou double, le traitement se fait en virgule flottante. *(Par exemple 18.9/5 donnera 3.78)*
- Les constantes entières sont par défaut de type int. Si un calcul effectué entre deux constantes déborde, le résultat devient négatif.
- Choisir des tailles de variables dimensionnées pour permettre de

Exemple : Programme Tester_operateur ternaire.ino

L'opérateur ternaire "?" :

L'opérateur ternaire, qui n'est qu'une autre forme pour effectuer un test de condition de type ***Si/Alors/Sinon***, tient son nom du fait qu'il est le seul en langage C à s'écrire avec trois items.

(Condition) **?** Instruction_pour_vrai **:** Instruction_pour_faux **;**

Les parenthèses ne sont pas obligatoires pour encadrer la Condition. Concrètement c'est l'équivalent d'un if / else condensé, plus rapide à écrire, mais qui rend bien plus délicate la lisibilité du code.

Contraintes de l'instruction ternaire ? :

- Les accolades ne sont pas admises pour les instructions,
- Un appel à procédure n'est pas acceptée,
- **Une seule instruction est possible pour chaque alternative.**
- Il ne faut pas de ";" à la fin de l'instruction relative au castrue, mais uniquement à la fin de l'instruction ternaire complète.

```
void loop() { HEURES++; if (HEURES==24) HEURES = 0;
  Serial.print("HEURES = "); Serial.print(HEURES);
  /* Forme standard sans les parenthèses */
  HEURES <= 12 ? Serial.print(" : Matin") : Serial.print(" : Apres midi");
  /* Avec parenthèses et sur trois lignes pour faciliter la lecture */
  (HEURES <= 12)
    ? Serial.println(" > AM.")  // Pas de ";" après l'affectation du "then".
    : Serial.println(" > PM."); // ";" pour terminer l'instruction complète.
  delay(500); }
```

Dans certains cas, l'écriture compacte d'un ***Si/Alors/Sinon*** peut s'avérer avantageuse dans la mesure où l'instruction est d'une interprétation évidente. Par exemple gérer un singulier et un pluriel dans un affichage :

```
Compteur++; if (Compteur==6) Compteur = 0;
  Serial.print("Compteur = "); Serial.print(Compteur);
  Serial.print(" comptage");
  /* Cas ou l'écriture simplifiée peut s'avérer plus simple : */
  Compteur > 1 ? Serial.println("s") : Serial.println();
```

Instruction ternaire = Affectation de valeur.

Fondamentalement, l'opérateur ternaire affecte une valeur à une variable en fonction d'une condition. Cette valeur peut être affectée dans une expression quelconque ou directement pour un affichage. Ex :

```
Calibre = (Taille > 25) ? "Valide" : "Rejet";
```

Portée des variables et qualificateurs :

Les variables en langage C ont implicitement une portée qui dépend de l'endroit du programme où elles sont déclarées. Toute variable *(Ou constante)* déclarée hors des fonctions ou des procédures ont une portée GLOBALE. Elles sont alors "vues" et peuvent être utilisées par n'importe quelle fonction ou procédure du programme. Toute variable déclarée dans le corps { } d'une fonction ou d'une procédure est une variable LOCALE qui n'est vue et manipulable que par cette dernière. Les déclarations des variables *(Ou des constantes)* peuvent se situer n'importe où dans le programme, mais doivent être déclarées avant leur utilisation. Il est fort recommandé pour éviter les "effets de bord" de déclarer en local la variable d'évolution d'une boucle for.

Le mot-clé const. *(Voir également en page 3)*

Le mot-clé **const** est une directive qui se place juste avant ou juste après le type d'un identificateur. Cet objet déclaré peut être utilisé comme n'importe quelle autre variable, mais sa valeur ne doit pas être changée dans le programme. *(Toute tentative de modification génèrera une erreur)*

Le mot-clé static.

Le mot-clé **static** est utilisé pour créer des **variables locales** dans une fonction qui conservent leurs valeurs entre deux appels de cette dernière. Les variables déclarées avec static sont créées et initialisées uniquement au tout premier appel de la fonction ou de la procédure.

Exemple : Programme Tester_volatil-et_Interruptions.ino

```
void setup() { Serial.begin(115200); }
void loop() {Experience_avec_static(); delay(1000);}
void Experience_avec_static () {
  static byte COMPTEUR = 0; // Static donc initialisé au 1ier appel.
 COMPTEUR++; Serial.print("Valeur de la variable statique = ");
Serial.println(COMPTEUR); }
```

Le mot-clé volatile.

Le mot-clé **volatile** est une directive qui précède la déclaration du type d'une variable pour indiquer au compilateur de placer cette dernière en RAM *(Adresse stable)* et non dans un registre de stockage, emplacement temporaire de la mémoire où les variables sont stockées et manipulées. Sous certaines conditions, la valeur de la variable stockée dans les registres peut-être perturbée. Avec Arduino, la seule situation où le problème peut se produire risque d'arriver dans les sections de codes associées aux interruptions. *(Routines de service des interruptions)*

stocker les plus grands résultats potentiels issus des calculs.

- Pour les mathématiques qui nécessitent des décimales ou des fractions, utiliser les variables de type float, mais ne pas oublier l'inconvénient de leur taille empiétée en mémoire, et la "lenteur" d'exécution des calculs qui en résulte.
- Utiliser les opérateurs de conversion de type pour transposer une variable d'un type en un autre type "à la volée". (Exemple int(MyFloat))

```
Variable = Dividende % Diviseur;
```

Fonction **Modulo** qui retourne **le reste de la division d'un entier par un autre**. L'opérateur modulo ne fonctionne pas avec les variables de type float. Ex : [7 % 5 ➤ 2] [9 % 5 ➤ 4][5 % 5 ➤ 0] [4 % 5 ➤ 4]

NOTE : Si on désire obtenir la **valeur entière d'un modulo** il suffit de soustraire à Variable le résultat de l'opérateur %.
Exemple : Val_entiere = Variable - (Variable % Diviseur);

Opérateurs composés.

```
Y = X++; // Affecte X à Y puis Incrémente X.
Y = ++X; // Incrémente X puis affecte sa valeur dans Y.
Y = X--; // Affecte X à Y puis Décrémente X.
Y = --X; // Décrémente X puis affecte sa valeur dans Y.
```

```
Y += X;  // Équivaut à l'expression Y = Y+X;
Y -= X;  // Équivaut à l'expression Y = Y- X;
Y *= X;  // Équivaut à l'expression Y = Y * X;
Y /= X;  // Équivaut à l'expression Y= Y / X;
```

Forme d'écriture non conseillée.

L'utilisation de ces opérateurs ne conduit à aucun gain de place pour le programme. Mais ils demeurent potentiellement des sources d'erreurs car la lecture de ces expression n'est vraiment pas naturelle.

Exemple : Programme ESSAI_de_base_sur_OP_COMPOSES.ino

Y &= X; *(ET BIT à BIT)* Y |= X; *(OU BIT à BIT)*

Les opérateurs composés BIT à BIT réalisent leur opérations au niveau le plus élémentaire, celui des BITs individuels constituant les variables.

```
Y &= X;  // Équivaut à l'expression Y = Y&X;
Y |= X;  // Équivaut à l'expression Y = Y|X;
```

X est une constante ou une variable entière de type char, int ou long.
Y est une variable entière de type char, int ou long.
Comme pour les opérateurs arithmétiques cette forme d'écriture n'est pas avantageuse, préférer l'opérateur d'affectation classique.

Fonctions arithmétiques.

X = min(Valeur1,Valeur2); Retourne le plus petit des deux nombres.
(X=min(X,MAX) permet de maintenir X en dessous de MAX.)
X = max(Valeur1,Valeur2); Retourne le plus grand des deux nombres.
(X=max(X,MIN) permet de maintenir X au dessus de MIN.)
Valeur1 et Valeur2 : n'importe quel type de donnée.
X = abs(Y); Retourne la valeur absolue de Y. *(Toujours positif)*

> Suite à la façon dont le compilateur implémente les instructions min(), max() et abs() il faut éviter d'utiliser d'autres fonction entre les parenthèses, car de faux résultats peuvent en résulter.
> min(X++, 100); // Fournit un mauvais résultat.
> X++; min(X, 100); // Méthode correcte.

X = constrain(X,INF,SUP);
Fonction qui impose à la variable X de rester dans la plage de valeurs située entre INF et SUP. Retourne la valeur de X si elle est contenue dans la fourchette, INF si X est plus petit et SUP si X est supérieur.
(X=max(X,INF,SUP) maintien X dans la plage [MIN-MAX])

map(X, "Plage d'entrée", "Plage de sortie"); *(Voir page 30)*

X = pow(BASE,EXPOSANT);
Fonction qui retourne la valeur du float BASE élevée à la puissance du float EXPOSANT. Les paramètres BASE et EXPOSANT comme constantes numériques peuvent être quelconques et en particulier des fractionnaires. Le résultat retourné est un double. La fonction pow() ne s'utilise qu'avec des variables de type float. Si on propose des int, long, etc, le résultat sera erroné. Dans la syntaxe décrite en ligne il est suggéré de transposer type "à la volée", mais les essais personnels avec les exemples suggérés n'ont pas fonctionné. Il semble donc impératif d'affecter aux paramètres le type float.

X = sq(Y); // Équivalent à pow(Y,2);
Fonction qui retourne le carré de Y d'un type quelconque.

X = sqrt(Y);// Équivalent à pow(Y,0.5);
Fonction qui retourne la racine carrée du type quelconque Y sous le format d'un long. Attention : Ne vérifie pas si Y est de signe positif.

NOTE : Avec pow() il est facile d'obtenir des cubes etc. Exemples :
X = pow(Y,3); // Y est élevé au **cube**.
X = pow(Y,0.33333333); // Y est élevé à la **racine cubique**.

Fonctions de gestion du temps.

Temps_Ecoule_depuis_RESET = millis();
La fonction millis() sans paramètre retourne en millisecondes le temps qui s'est écoulé depuis le redémarrage du microcontrôleur sous la forme d'un unsigned long. Par un simple calcul on peut en déduire que l'horloge interne de l'ATmega328 ne recyclera pas à zéro avant une durée de : 4294967,295 / 86400 = 49,71 jours. (@)

Temps_Ecoule_depuis_RESET = micros();
Fonction strictement analogue à millis(), mais retourne la valeur en microsecondes avec une résolution de 4µS. Le débordement de TIMER 0 se fait donc toutes les 4294,967 / 60 = 71,6 minutes. (@)

delay(DURÉE);
Fonction qui **suspend le déroulement du programme** pendant DURÉE millisecondes. Le programme est arrêté, mais des activités prioritaires sont déclenchées par toutes les interruptions qui continuent en tâche de fond. *(Dialogue série RX, génération PWM etc)*. (@)

(@) : La valeur retournée étant un unsigned long, et des erreurs de calculpeuvent se produire si le programme utilise des opérations mathématiques avec d'autres opérandes qui ne sont pas des entiers.
(@) : Cette fonction fse sert du "TIMER 0" de l'ATmega328.
Contrairement à ce que signale la documentation officielle, les interruptions seraient suspendues par ces fonctions.

delayMicroseconds(DURÉE);
Réalise une temporisation de DURÉE microsecondes dont la valeur ne doit pas dépasser 16383 pour que le délai soit parfaitement respecté. La précision est obtenue à partir d'un minimum de 3µS. Pour respecter la consigne, cette fonction désactive les interruptions durant son exécution. Son codage spécifique ne fait intervenir aucun "Timer" et ne perturbe pas le fonctionnement des interruptions. Donc à privilégier pour des pauses courtes dans des routines d'interruption.

> **NOTE :** Les procédures delay() ne sont réellement utilisables que pour des délais très courts car elles "stoppent" le programme. *(Par exemple 5mS pour les dialogues série ...)* Pour des temporisations plus importantes qui doivent laisser le logiciel de base se dérouler, il faut faire appel à des bascules booléennes gérées avec l'horloge millis(). Le petit programme Temporisation_sans_utiliser_delay.ino en donne un exemple avec deux chronométrages simultanés.

Transpositions de valeurs avec la fonction map :

Analysée en détail sur une fiche dédiée, la transposition de valeurs y utilise des écritures logicielles personnelles. Mais ce besoin en traitement se retrouve presque dans chaque programme. Le langage C d'Arduino permet une simplification d'écriture pour faciliter la tâche des développeurs. C'est la fonction map qui propose une option plus directe informatiquement que de faire un "produit en croix". La fonction map s'exprime sous la forme syntaxique suivante :

```
map(Num_en_Entrée, "Plage d'entrée", "Plage de sortie");
```

En tant que fonction, map retourne une valeur numérique de type int ou long qui résule d'un "produit en croix" à partir de Num_en_Entrée.
"Plage d'entrée" sera définie par deux entiers séparés par une ",".
"Plage de sortie" sera définie par deux entiers séparés par une ",".

À partir d'une valeur de Num_en_Entrée, d'un intervalle "Plage d'entrée" et d'un intervalle "Plage de sortie", la fonction retourne la valeur équivalente linéairement, comprise dans le deuxième intervalle.

Nous allons reprendre la transposition proposée Fig.1 sur la fiche : Transposer la plage de [0 à 1024] en [0 à 255] que nous avions codé :

```
Transposée = (uint32_t) Analog * 255 / 1023
```

Avec la fonction map cette écriture devient :

```
Transposée = map(Analog, 0, 1023, 0, 255);
```

Cette technique résous le problème de cohérence des variables abordé en page 28. Les plages peuvent avoir des bornes avec signe négatif. Par contre **le résultat d'une fonction** map est **un entier**. De ce fait la transposition fait perdre les décimales éventuelles. C'est la raison pour laquelle dans l'exemple ci-dessus on exprime la "Plage de sortie" en milli Volts. Il suffit ensuite de diviser par 1000 pour avoir le résultat en Volts ce qui avec A0 ≈ 2849V, donne en fonction du codage :

```
Transposée =(map(Analog, 0, 1023, 0, 5000)) / 1000;
// Retourne 2.000 les décimales sont perdues. Les décimales /
// sont perdues, car la division par 1000 se fait sur un entiers.
Transposée = map(Analog, 0, 1023, 0, 5000) ;
// Retourne 2849.000 mais dans un réel qui conservera les
// décimales lors de la division par 1000.
Serial.println(Transposée/1000); // Affiche correctement 2.849.
```


Exemple : Programme Test_du_generateur_ALEATOIRE.ino

Exemple : Programme Fonctions_TRIGONOMETRIQUES.ino

Fonctions trigonométriques.

X = sin(ANGLE) ; Fonction qui calcule sous la forme d'un double le sinus du paramètre ANGLE supposé exprimé en radians.

X = cos(ANGLE) ; Fonction qui calcule sous la forme d'un double le cosinus du paramètre ANGLE supposé exprimé en radians.

X = tan(ANGLE) ; Fonction qui calcule sous la forme d'un double la tangente du paramètre ANGLE supposé exprimé en radians. Pour les valeurs π/2 et 2π/3 ne retourne pas "infini" mais la valeur -22845570.

- Pour transformer ANGLE exprimé en degrés vers les radians :
  `Radians = ANGLE * PI / 180;`
- Pour avoir la **cotangente** d'un angle on peut coder :
  `X =cos(Radians)/sin(Radians) ;`

Génération de nombres aléatoires.

X = random(MAX); *(random est une fonction)*
Génère un nombre pseudo aléatoire compris entre 0 et MAX-1.

X = random(MIN,MAX);
Génère un pseudo aléatoire compris entre MIN et MAX-1 quels que soient les signes des bornes si elles sont données dans l'ordre croissant.

- Si aucun paramètre n'est indiqué génère des valeurs de type long.
- Si MAX est négatif et indiqué sans MIN le signe négatif est ignoré.
- Si MAX est plus petit que MIN, ne génère que la constante MIN à chaque appel de la fonction quel que soit le signe de MAX.

La séquence génère une **suite de valeurs aléatoire** très longue, mais **systématiquement identique** à chaque redémarrage du programme.

randomSeed(VALEUR)
Procédure qui impose à la séquence générée de démarrer en "un point arbitraire". Si l'on désire obtenir une séquence différente à chaque RESET, il faut que VALEUR soit une entité différente à chaque redémarrage du programme. Dans ce but il est possible d'utiliser la valeur retournée par la lecture d'une broche analogique non connectée, par exemple avec l'instruction : randomSeed(analogRead(0)). Par contre, si l'on désire utiliser des séquences pseudo-aléatoires qui se répètent exactement à des fins de comparaison expérimentales par exemple, il faut que le paramètre VALEUR soit une constante, et initialiser avant de faire appel à la séquence de génération random().

Les opérateurs LOGIQUES bit à bit :

Mis à part l'opérateur de négation ~, les autres opérateurs sont appliqués sur deux entités en relation binaire. Plus précisément l'opération s'effectue bit à bit avec une correspondance directe en fonction de leurs poids binaires.

L'opérateur de négation :

~ inverse chaque bit : Les "0" deviennent des "1" et réciproquement.
À noter que le type de la variable à une importance :
int x = B101; alors ~x; vaudra B1111111111111010 et non B010.

Les opérateurs logiques standards :

• Le ET : B110 & B100; vaut B100 *(Forcer des bits à zéro)*
• Le OU : B110 | B100; vaut B110 *(Forcer des bits à 1)*
• Le OU exclusif : B110 ^ B100; vaut B010 *(Inverser des bits)*
Noter que les formes compactes sont utilisables sur Arduino :
X &= Y est équivalent à X = X&Y. **(Voir encadré page 31)**
X |= Y est équivalent à X = X|Y.
X ^= Y est équivalent à X = X^Y.
Exemples : Programme Travail_sur_les_OP_LOGIQUES.ino.

Décalage à GAUCHE.

Les bits qui "sortent" sont perdus, et les bits insérés sont des zéros.
L'opérateur est << et le deuxième paramètre est le nombre de décalages à gauche qui seront effectués, donc le nombre de zéros ajoutés à droite.
Si int X = B101, X << 2; vaudra B10100
et X << 14; vaudra B1000000*00000000*.

Les fonctions SHIFT :

Décalage à DROITE.

Les bits qui "sortent" sont perdus, les bits ajoutés ne sont pas forcément des zéros. Plus précisément, si la variable est signée *(De type char, int ou long)*, ils auront la même valeur que le bit le plus à gauche : Il y a extension du signe. **Si la variable n'est pas signée**, le comportement classique s'observe : Des zéros sont systématiquement ajoutés.
int X = B1000000000001010; • X >> 2; vaudra B1110000000000010
unsigned int Y = B1000000000001010;
• Y >> 2; vaudra B0010000000000010
La solution pour imposer un comportement standard à X consiste à le transtyper : (unsigned int)X >> 2 vaudra B0010000000000010.
Exemples : Programme SHIFT et PORTS.ino.

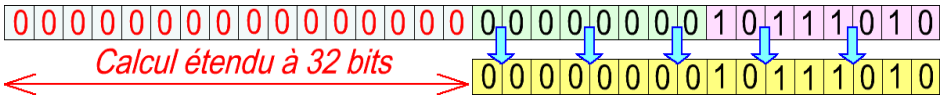
Exemple : Programme Travail_sur_les_OP_LOGIQUES.ino
Exemple : Programme SHIFT_et_PORTS.ino

Forcer la taille des données dans un calcul :

Dans l'exemple de la page 28 il suffit de donner à la variable Commande le type unsigned long et l'affectation devient homogène. Mais ce n'est pas idéal car une variable qui peut être contenue sur deux octets va alors en exiger le double. En C, tous compilateurs confondus, ce n'est pas le type de la variable "calculée" *(Celle à gauche de l'opérateur d'affectation "=")* qui définit le type du résultat mais le type le plus étendu des variables impliquées dans l'expression définissant le traitement : Le calcul est effectué avant l'affectation, c'est lui qui impose le format. Une solution, sans user de variables surdimensionnées existe, il suffit de forcer le calcul à une "taille suffisante" lors du calcul. On nomme ça effectuer un cast. Le codage consiste à faire précéder l'expression de calcul d'une directive de type : (uintNN_t) où NN est le nombre de bits réservés aux éléments durant le calcul. *(Exemples : (uint16_t), (uint32_t), (uint64_t))*

```
unsigned int Commande; // Valeur mesurée pour éclairer.
unsigned long Eclairage_LED; // Valeur pour PWM.
void loop() {
  Commande = analogRead(Commande_lumiere);
  Eclairage_LED = (uint32_t) Commande * 255 / 1023;
  analogWrite(LED,Eclairage_LED); }
```

Avec l'ajout de (uint32_t) avant la définition de l'expression de calcul la variable de réception Eclairage_LED fait toujours 16 bits, mais les calculs sont forcés en 32 bits. Donc les résultats intermédiaires se font sans être tronqués avec de l'interprétation "en complément à deux". Une fois l'expression calculée, les 32 bits présentent des zéros en tête et seul les poids faibles sont affectés à la variable Eclairage_LED.



Utilisation des E/S binaires :

Utiliser un B.P. pour générer une action.

Exemple avec un B.P. qui appuyé génère un "0" logique :

```
const byte BoutonPoussoir = 12;  // Entrée 12 pour tester.
if ((digitalRead(BoutonPoussoir) == 0)) {Actions;} ;
```

Pour une action sur état logique "1" prendre ==1 dans le test.

Exemple : Programme Test_sur_les_calculs.ino

Problème de cohérence du type des variables :

C'est un aspect du compilateur qui peut engendrer des résultats erronés alors que fondamentalement le choix du type des données n'a rien d'illogique. L'évaluation d'un calcul ou d'une expression d'un test pour une structure **if** peuvent provoquer des dysfonctionnements sournois pas faciles à déceler lors de la mise au point d'un programme.

Exemple pour un calcul utilisant des entiers.

Le programme Pilotage_Analogique_LED.ino *est cohérent au sens mathématique. On multiplie deux entiers, donc c'est homogène :*

```
unsigned int Commande; // Valeur mesurée pour éclairer.
unsigned long Eclairage_LED;// Valeur pour PWM.
void loop() {
  Commande = analogRead(Commande_lumiere);
  Eclairage_LED = Commande * 255 / 1023;
  analogWrite(LED,Eclairage_LED); }
```

Commande varie entre [0 et 1023] donc choisir unsigned int pour rester dans la plage des valeurs et travailler avec des entiers positifs.

Eclairage_LED doit varier entre [0 et 255] mais choisir le type unsigned long car le calcul rouge peut arriver à la valeur calculée de 1023 * 255 = 260865. Pour transposer [0 à 1023] en [0 à 255] le calcul consiste à diviser par 1023 et multiplier par 255, mais il faut commencer par la multiplication car on travaille avec des entiers et 255 / 1023 donnerait 0,25 donc zéro.

unsigned int : Valeurs possible = [0 à 65535]
unsigned long : Valeurs possible = [0 à 4294967295]
Donc des intervalles de valeurs théoriquement compatibles.

Le résultat du calcul et la valeur envoyée à PWM sont incorrect car les deux coté de l'expression d'affectation ne sont pas de types homogènes. Eclairage_LED varie de 0 à 30 pour 635 puis saute à 4294967268 à partir de 655 environ. *(Quatre fois sur la course du potentiomètre)*

CONCLUSION : Si un traitement réalise des opérations arithmétiques, toutes les variables de l'expression doivent être du même type et d'une taille qui accepte la plus grande valeur intermédiaire pouvant être générée dans les calculs.

Opérateurs booléens et masques logiques.

Il importe de ne pas confondre &&, l'opérateur ET utilisé dans des expressions booléennes avec l'opérateur logique & utilisé avec des masques logiques dans les instructions bit à bit.

De la même façon, ne pas confondre | |, l'opérateur OU dans des expressions booléennes avec l'opérateur |, pour le OU bit à bit.

Les fonctions orientées OCTETS :

lowByte(Data);
Fonction qui retourne l'octet de poids faible *(Situé le plus à "droite")* d'une variable entière >>> Data : byte, char, word,int, long.

highByte(Data);
Fonction qui retourne l'octet de poids fort d'une variable entière. *(Situé le plus à "gauche d'un type word ou le second pour une variable de taille plus grande")* Data : byte, char, word,int, long.

Les fonctions orientées BITS :

bitRead(Data,N);
Fonction qui retourne "0" ou "1" en fonction du BIT de position N testé dans la variable Data. Le rang du bit à lire est N, en partant de 0 pour le bit de poids faible. Ne vérifie pas un débordement sur N.

bitSet(Data,N);
Procédure qui force à "1" le BIT de rang N dans Data. Le rang du bit est N, en partant de 0 pour le poids faible sans vérifier le débordement.

bitClear(Data,N);
Procédure qui force à "0" le BIT de rang N dans Data. Le rang du bit est N, en partant de 0 pour le poids faible sans vérifier le débordement.

bitWrite(Data,N,ETAT);
Procédure qui force à l'ETAT logique le BIT de rang N dans Data. Le rang du bit est N, en partant de 0 pour le poids faible sans vérifier le débordement. ETAT peut être une variable booléenne. Toute valeur supérieure à 0 pour ETAT se traduira par l'écriture d'un "1" .

bit(N);
Fonction qui calcule la valeur numérique d'un bit spécifié. *(Retourne 2 élevé à la puissance du bit indiqué : 2^0=1, 2^1=2, 2^2=4, etc)*

(Voir également Utilisation des E/S binaires en page 26)

Exemple : Programme Test_des_PORTS.ino

Travail sur les ports de l'ATmega328 :

Les registres des PORTS d'E/S permettent la manipulation plus rapide des broches d'interfaçage du microcontrôleur utilisé sur une carte Arduino. Les contrôleurs de la famille AVR *(ATmega8 et ATmega168)* possèdent trois ports spécifiques :

- **B** pour les E/S binaires de 8 à 13,
- **C** pour les Entrées Analogiques,
- **D** pour les E/S binaires de 0 à 7.

Chaque port du micro contrôleur est géré par trois "variables registres" *(Identificateurs réservés)* définies dans le langage C d'Arduino.

L'identificateur DDRx :

C'est le registre de direction des données qui détermine si la broche relative est une Entrée ou une Sortie en fonction de l'état "0" ou "1".

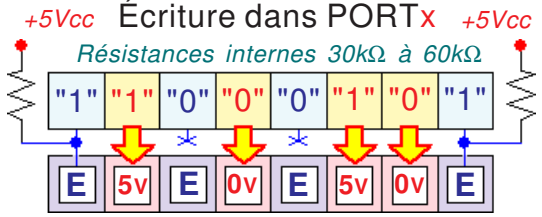
- En écriture définit le sens des données BIT à BIT.
- En lecture permet de saisir l'état actuel d'initialisation de ce registre.

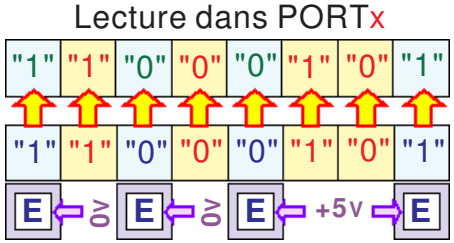
DDRx	0	1	0	1	0	1	1	0
PORTx	E	S	E	S	E	S	S	E

```
DDRB = B00111001; // Force 4 sorties : D8, D11, D12 et D13.
DDRD = DDRD | B11111100; // Force 6 sorties sans changer 0 et 1.
```

L'identificateur PORTx :

C'est l'instruction qui permet de forcer des états logiques dans les BITs du registre de données qui sont initialisés en sorties. Les broches qui sont forcées en entrées voient l'utilisation de la résistance interne de forçage du niveau haut **connectée sur un état "1"**, et **isolée sur un état "0"**.





Une lecture dans PORTx retourne les valeurs qui ont été écrites dans le registre qui impose les états sur les broches de sorties. Les valeurs lues sur les bits correspondant à des entrées ne sont pas influencées par l'état électrique de la broche correspondante, mais uniquement par l'état des bits qui conditionnent la mise en service ou l'isolation de la résistance interne de forçage de niveau.

Exemple : Programme Buzzer_passif.ino

Exemple : Programme Tester_pulseIn.ino

Les Entrées / Sorties évoluées.

tone(Broche, Frequence, Durée);

Procédure qui **génère une onde carrée** de Frequence spécifiée sur une Broche quelconque d'E/S. La Durée exprimée **en millisecondes** peut être précisée, sinon le signal continue jusqu'à l'appel de l'instruction noTone(). Une seule note peut être produite à la fois. Si une note est déjà générée sur une autre broche, l'appel de la fonction tone() n'aura aucun effet tant qu'une instruction noTone() n'est pas rencontrée dans le programme. Si la note est requise sur la même broche, l'appel de la fonction tone() modifiera la fréquence de l'onde générée. Toute broche d'E/S d'Arduino peut être utilisée, y compris les "entrées analogiques". La déclaration en OUTPUT de Broche n'est pas nécessaire.

ATTENTION : Une fois déclenchée la génération du signal est autonome. Si avant la fin de Durée une instruction tone() est rencontrée dans le programme, la signal sera généré "en continu".

noTone(Broche);

Procédure qui stoppe sur Broche la génération du signal carré déclenchée par l'instruction tone(). Cette instruction ne provoque aucun effet si aucune tonalité n'est en cours de génération.

---

pulseIn(Broche, ETAT, Delai);

Fonction qui retourne la durée d'une impulsion de niveau ETAT mesurée sur Broche configurée en ENTREE. Elle attend que Broche passe à ETAT, puis effectue le chronométrage jusqu'au niveau contraire, stoppe la mesure et retourne la valeur exprimée en µS. Si l'option Delai est précisée, il y a sortie de la fonction et retour de la valeur 0 si aucune impulsion n'est survenue dans le Delai spécifié en µS. *(Valeur par défaut 1S)* La plage fiable de mesures se situe entre 10µS et 3 minutes.

---

shiftOut(Data, CLOCK, OrdreTX, OCTET);

Procédure de type SPI pour un protocole de communication série synchrone. Emet un par un les bits d'un OCTET de données dans l'ordre fixé par OrdreTX sur la broche Data après quoi la broche CLOCK est inversée pour indiquer que le bit est disponible. Les deux entités sont synchronisées et communiquent à vitesse maximale car elles partagent la ligne CLOCK. Le paramètre OrdreTX est défini par MSBFIRST ou LSBFIRST. La broche Data et la broche CLOCK doivent toujours être configurées en SORTIES à l'aide de pinMode().

```
int Valeur = 500; // il faut deux octets pour transmettre cette valeur.
shiftOut(Data, CLOCK, MSBFIRST, (Valeur >> 8)); // Décalage de 8 bits
    vers la droite pour isoler les bits forts.
shiftOut(Data, CLOCK, MSBFIRST, Valeur); // Envoyer l'octet LSB.
```

Les Entrées / Sorties binaires :

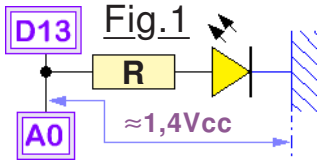


Fig.1

Arduino-Uno dispose de quatorze broches d'Entrée/Sortie binaires dont six peuvent fonctionner en mode PWM. *(Voir page 25)* Mais si ce nombre est insuffisant, utiliser les entrées analogiques comme des SORTIES binaires s'impose. Noter également que la broche binaire 13 est reliée à une LED et sa résistance de limitation de courant soudées sur le circuit imprimé de la carte Arduino. *(Fig.1)* Activer la résistance interne de rappel *(Voir plus bas)* force la tension à environ 1,4V au lieu des 5V théoriques. Il faut donc en tenir compte.

pinMode(Broche,Mode);

Configure la Broche spécifiée pour qu'elle se comporte soit en entrée, si Mode est défini par INPUT, soit en sortie avec OUTPUT.

digitalRead(Broche);

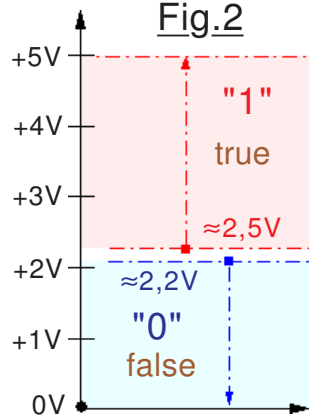


Fig.2

Fonction qui lit l'état logique de Broche et retourne les entiers "0" ou "1" en fonction de la tension présente sur la broche déclarée en ENTRÉE. Ces valeurs numériques sont interprétées comme des états false et true par les opérateurs logiques. Si la broche n'est pas connectée, les tensions statiques qui s'y trouvent génèrent des états aléatoires. La valeur de l'état logique retournée est fonction de la tension présente sur l'entrée Binaire. La Fig.2 donne les valeurs, avec un léger effet "Trigger de Schmitt" au franchissement du seuil de référence. Naturellement la broche doit avoir été déclarée en entrée avec pinMode. Enfin, si un état extérieur n'est pas forcé, il faut, avec l'instruction spécifique digitalWrite(Broche, HIGH); activer le "rappel interne au +Vcc".

>>> Voir l'utilisation logique des Entrées en bas de la page 24.

digitalWrite(Broche, Etat);

Force un niveau logique "1" avec HIGH ou un état logique "0" avec LOW **si la** Broche **est déclarée en SORTIE** avec pinMode. **Si la broche est déclarée en ENTRÉE**, HIGH valide la **résistance de rappel interne de "pullup" de 20 kΩ** ou passe l'entrée en mode haute impédance avec la constante prédéfinie LOW.


```
byte Etat_port_B;
void setup() {DDRB = B110011;  // Impose 4 sorties sur le port B.
  PORTB = B011001;  // Impose 4 états initiaux en sortie.}
void loop() {
  Etat_port_B = PORTB; // Lecture du port B. @
  DDRB = B111111; // Passe B en six sorties pour afficher la lecture @.
  PORTB = Etat_port_B; } // Recopie sur les LED l'état lu en @.
```

Dans l'exemple ci-dessus B2 est isolée et en B3 la résistance de forçage du niveau haut est mise en service. En rouge les états logiques sont recopiés sur les quatre sorties. En lecture @ on va obtenir 011001 quels que soient les états logiques sur les entrées B2 et B3.

L'identificateur PINx :

Cette instruction en LECTURE permet de saisir simultanément l'état de toutes les broches déclarées en entrées sur le PORTx. En lecture sur des bits de sorties on retrouve l'état logique de ces dernières.

```
PORTD = PINC;  // Recopier sur 0 à 5 l'état des entrées Analogiques.
if (PINB & 4) {Action du if}; // Réalise l'action si B10 = "1".
```

Contrairement aux informations trouvées sur Internet qui stipulent que PINx ne fonctionne qu'en lecture, on peut affecter PINx en écriture. Cette instruction utilisée en ÉCRITURE permet d'**INVERSER l'état LOGIQUE des SORTIES** avec un l'état "1" sur le bit correspondant. Les sorties associées à un état logique "0" ne sont pas affectées.

```
void setup() {DDRB = B00111111;  // Impose 6 sorties sur 8 à 13.
  PORTB = B001100; }  // Impose 6 états initiaux en sortie.
void loop() {
  PINB = B101010; // Inverse 3 sorties sans modifier les autres.
  delay(200); }
```

Circuit minimal pour un ATmega328 :

Mémoire de l'ATmega328 programmée, le microprocesseur peut fonctionner avec trois fois rien. Broche 7 au +5Vcc, Broche 8 *(Ou broche 22)* à la masse. Un quartz quelconque entre les broches 9 et 10 et le circuit fonctionne. Normalement les broches 9 et 10 doivent être reliées à la masse par un condensateur de 12pF à 22pF. Lors des essais ces deux composants n'étaient pas branchés, les capacités parasites étant suffisantes pour faire osciller sans problème des quartzs dont la fréquence était comprise entre 1,8MHz et 18MHz.

Les Entrées analogiques : (Sorties binaires)

La fonction de transfert N = k * V du CAN d'Arduino est agencée pour retourner des valeurs comprises entre 0 et 1023 quand la tension en entrée An varie entre 0 et +Référence. Le temps de conversion avoisine 100µS, la fréquence maximale d'échantillonnage est donc de 10 kHz. Fonctionnant en haute impédance, une entrée analogique laissée "en l'air" retourne les valeurs fluctuantes des tensions électrostatiques.

analogRead(Broche_analogique); *(Voir encadré en bas de page 39)*

Fonction qui retourne l'équivalent numérique de la tension présente sur Broche_analogique. Le CAN de 10 bits génère une valeur comprise entre 0 et 1023 quand la tension en entrée par défaut varie entre 0 et +5Vcc. La résolution est donc de : 5 / 1024 = 4,88 mV. Avec cette fonction les broches sont implicitement des entrées et le paramètre Broche_analogique peut prendre les valeurs de 0 à 5.

analogReference(ModeCNA);

Procédure qui impose la référence pleine échelle du CNA. ModeCNA : DEFAULT, INTERNAL ou EXTERNAL.

- DEFAULT : La valeur de référence pour 1023 est +5 volts.
- INTERNAL : Une tension de référence interne égale à 1,1 volts.
- EXTERNAL : La tension de référence est prise sur la broche AREF utilisée en tant que Référence de tension. Il est fortement recommandé d'insérer une résistance de protection **R** d'environ 4,7 kΩ en entrée de AREF pour éviter des dégradations internes à l'Atmega si la configuration logicielle de l'instruction analogReference est incompatible avec l'intensité qui résulterait d'un branchement externe. *(AREF passe en basse impédance avec le paramètre INTERNAL)* **R** diminue la tension qui sera réellement utilisée comme référence, car il y a une résistance interne de 32 kΩ sur la broche AREF.

Entrées Analogiques utilisées en SORTIES :

Bien que fondamentalement dédiées aux fonctions d'entrées analogiques, les six broches An fonctionnent également de façon banale si elles sont initialisées en sorties. Pour les différencier des autres broches Dn elles sont par convention notées de 14 pour A0 à 19 pour A5. Exemple :

```
pinMode(15, OUTPUT); // L'entrée A1 devient une sortie.
digitalWrite(15, LOW);  // État LOW ou HIGH naturellement.
```

digitalWrite(15, HIGH); active également une résistance de pullup interne qui force une tension de +5Vcc si A5 est initialisée en ENTRÉE.

Exemple : Programme Test_mesure_analogique.ino

Exemple : Programme Pilotage_Analogique_LED.ino

Les sorties analogiques PWM :

Contrairement à ce que laisse supposer le terme ANALOGIQUE utilisé pour les sorties PWM 3, 5, 6,9, 10 et 11, il s'agit bel et bien de sorties binaires pouvant fonctionner en mode impulsions à modulation de largeur. *(Pulse Width Modulation)* Le rapport cyclique est directement proportionnel à la valeur N passée en paramètre :

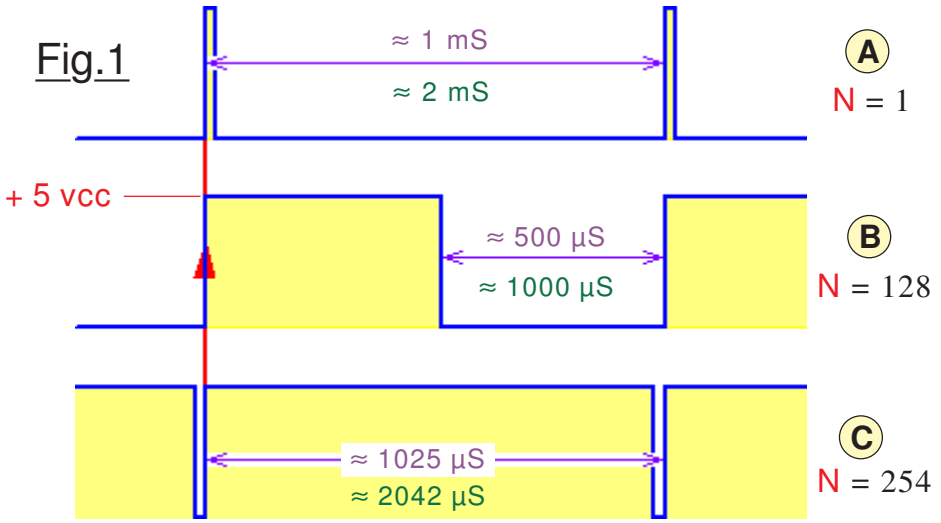
analogWrite(Sortie_PWM, N)

Sortie_PWM : Numéro de la broche de sortie utilisée.

N : Valeur qui impose la durée au niveau 1.

La fréquence de répétition mesurée est de 490 Hz avec une **période T ≈ 2042 µS** pour PWM 3, 9, 10 et 11. Pour les sorties PWM 5 et 6 la **fréquence de répétition mesurée est de 976 Hz** avec une **période T ≈ 1025 µS**. La Fig.1 montre un dessin des signaux observés avec un oscilloscope sur une carte Arduino Uno en fonction de la valeur de N.

Fig.1



Utiliser une sortie déclarée PWM en mode BINAIRE pur :

- Quand N = 0 la sortie reste en permanence à 0 V.
- Quand N = 255 la sortie reste constamment à = 5 VCC.

Le programme Test_mesure_analogique.ino recopie sur la sortie PWM n°3 la valeur de la tension mesurée sur l'entrée analogique A0. *(Valeur moyenne puisque le signal est pulsé et de forme "carrée")*

Noter que pour convertir une valeur mesurée en une donnée valide sur un octet il suffit de diviser la "définition" du CNA par quatre.