# Fiches diverses pour ARDUINO


**Différents types de données utilisables avec Arduino.**

**TABRE DES CARACTÈRES ASCII affichables sur LCD**

***Détection d'un front sur une entrée analogique.***

***Détection d'un front sur une entrée binaire.***

***Calculs de transpositions de valeurs.***

***Différents types de mémoires sur Arduino.***

***Gestion du "temps processeur".***

## Différents types de données utilisables avec Arduino.

| TYPE | Valeurs possibles | BITS | Octets |
|---|---|---|---|
| char | -128 à +127 | 8 | 1 |
| int | -32768 à +32767 | 16 | 2 |
| long | -2147483648 à +2147483647 | 32 | 4 |
| Variables Décimales. | | | |
| float | 3,4 * 10^(-38) à +3,4 * 10^(38) | 32 | 4 |
| double | 1,7 * 10^(-308) à +1,7 * 10^(308) | 64 | 8 |
| La gestion des variables de type double dans Arduino est exactement la même que celle des variables de type float, sans gain de précision. | | | |
| Variables NON SIGNÉES : unsigned | | | |
| char | 0 à 255 | 8 | 1 |
| int | 0 à 65535 | 16 | 2 |
| long | 0 à 4294967295 | 32 | 4 |
| Variables propres au langage C d'Arduino | | | |
| byte | 0 à +255 | 8 | 1 |
| word | 0 à +65535 | 16 | 2 |
| boolean | 0 ou 1 | 1 | 1 |
| Variables booléennes | | | |
| boolean Nom Variable = FALSE ou TRUE | | | |
| Toute variable peut servir de variable booléenne :<br>• Si elle vaut 0 ce sera intreprété comme FALSE.<br>• Si elle est ≠ 0 ce sera intreprété comme TRUE. | | | |
| *Conversion Numérique analogique pour les entrées A0 à A5* | | | |
| CAN | 0 à 1023 | 10 | 2 |
| "boolean" NomBrocheSortieBinaire = LOW ou HIGH | | | |

## TABRE DES CARACTÈRES ASCII affichables sur LCD

| DEC | B | Car | DEC | B | Car | DEC | B | Car |
|---|---|---|---|---|---|---|---|---|
| 32 | 00100000 | SPC | 64 | 01000000 | @ | 96 | 01100000 | |
| 33 | 00100001 | ! | 65 | 01000001 | A | 97 | 01100001 | a |
| 34 | 00100010 | " | 66 | 01000010 | B | 98 | 01100010 | b |
| 35 | 00100011 | # | 67 | 01000011 | C | 99 | 01100011 | c |
| 36 | 00100100 | $ | 68 | 01000100 | D | 100 | 01100100 | d |
| 37 | 00100101 | % | 69 | 01000101 | E | 101 | 01100101 | e |
| 38 | 00100110 | & | 70 | 01000110 | F | 102 | 01100110 | f |
| 39 | 00100111 | ' | 71 | 01000111 | G | 103 | 01100111 | g |
| 40 | 00101000 | ( | 72 | 01001000 | H | 104 | 01101000 | h |
| 41 | 00101001 | ) | 73 | 01001001 | I | 105 | 01101001 | i |
| 42 | 00101010 | * | 74 | 01001010 | J | 106 | 01101010 | j |
| 43 | 00101011 | + | 75 | 01001011 | K | 107 | 01101011 | k |
| 44 | 00101100 | , | 76 | 01001100 | L | 108 | 01101100 | l |
| 45 | 00101101 | - | 77 | 01001101 | M | 109 | 01101101 | m |
| 46 | 00101110 | . | 78 | 01001110 | N | 110 | 01101110 | n |
| 47 | 00101111 | / | 79 | 01001111 | O | 111 | 01101111 | o |
| 48 | 00110000 | 0 | 80 | 01010000 | P | 112 | 01110000 | p |
| 49 | 00110001 | 1 | 81 | 01010001 | Q | 113 | 01110001 | q |
| 50 | 00110010 | 2 | 82 | 01010010 | R | 114 | 01110010 | r |
| 51 | 00110011 | 3 | 83 | 01010011 | S | 115 | 01110011 | s |
| 52 | 00110100 | 4 | 84 | 01010100 | T | 116 | 01110100 | t |
| 53 | 00110101 | 5 | 85 | 01010101 | U | 117 | 01110101 | u |
| 54 | 00110110 | 6 | 86 | 01010110 | V | 118 | 01110110 | v |
| 55 | 00110111 | 7 | 87 | 01010111 | W | 119 | 01110111 | w |
| 56 | 00111000 | 8 | 88 | 01011000 | X | 120 | 01111000 | x |
| 57 | 00111001 | 9 | 89 | 01011001 | Y | 121 | 01111001 | y |
| 58 | 00111010 | : | 90 | 01011010 | Z | 122 | 01111010 | z |
| 59 | 00111011 | ; | 91 | 01011011 | [ | 123 | 01111011 | { |
| 60 | 00111100 | < | 92 | 01011100 | \ | 124 | 01111100 | \| |
| 61 | 00111101 | = | 93 | 01011101 | ] | 125 | 01111101 | } |
| 62 | 00111110 | > | 94 | 01011110 | ^ | 126 | 01111110 |  126 |
| 63 | 00111111 | ? | 95 | 01011111 | _ | 127 | 01111111 |  127 |

## *Détection d'un front sur une entrée analogique :*

Dans son principe, la détection d'un front n'est pas très compliquée, mais impose la mémorisation de t'état en cours, car c'est le changement de ce dernier qui est significatif d'un basculement et non le fait qu'il soit à "1" ou à "0". C'est la "contradiction" entre l'état actuel et l'état mémorisé qui permet de déterminer la présence d'un front descendant. Il en résulte le test @ qui dans le programme rétablit la cohérence des deux booléens et déclenche l'action sur front détecté.
L'organigramme de la Fig.1 présente la logique du traitement de détection du front descendant. Ce diagramme est à lier au dessin de la Fig.2 qui précise l'ordre chronologique des événements.

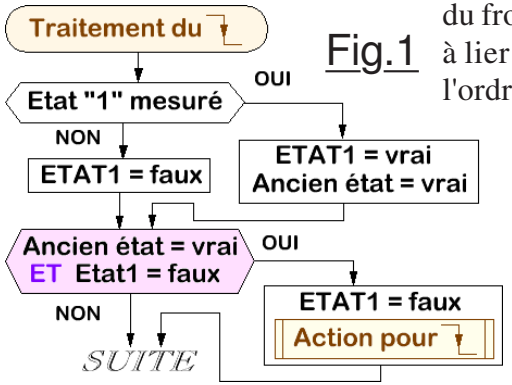


Fig.1

En bleu nous avons l'évolution de l'état logique sur l'entrée testée en fonction du temps, avec un changement d'état en **1** puis en **2**. En **3** chaque trait vertical symbolise la lecture de l'entrée durant la boucle principale du programme. Ce n'est qu'en **4** que l'état "1" est pris en compte par le programme, et en **5** que le passage à zéro est détecté. La zone violette et la zone jaune sont représentatives du retard de prise en compte de l'état mesuré. La trace rouge est relative à la valeur affectée à la variable Ancien_Etat. En **6** le changement est immédiat, mais pour le passage à zéro en **7** il faut que le programme effectue le test de détection de la transition négative @ d'où le petit retard repéré en vert sur le diagramme. Les états "stables" qui suivent les transitions **4** et **5** ne doivent plus avoir d'effet si ce n'est de confirmer pour le niveau "1" la stabilité de sa valeur jusqu'à la prochaine transition **5**.

```
/* Exemple simple de détection d'un front descendant */
int CNA; // Mesure analogique retournée par le CNA.
const byte Entree_mesuree = 0; // Mesure analogique sur A0.
boolean Etat1; // État binaire issu de la mesure analogique.
boolean Ancien_Etat; // État binaire mémorisé.

void loop() {
  CNA = analogRead(Entree_mesuree);
  if (CNA > Seuil) { Etat1=true; Ancien_Etat=true;}
     else {Etat1=false;};
  // ========== Gérer le front descendant. ==========
  if ((Ancien_Etat=true) && (!Etat1)) @
     {Ancien_Etat=false; ActionSurFrontDescendant();} ; }
```

Voir aussi le cas d'un bouton poussoir.

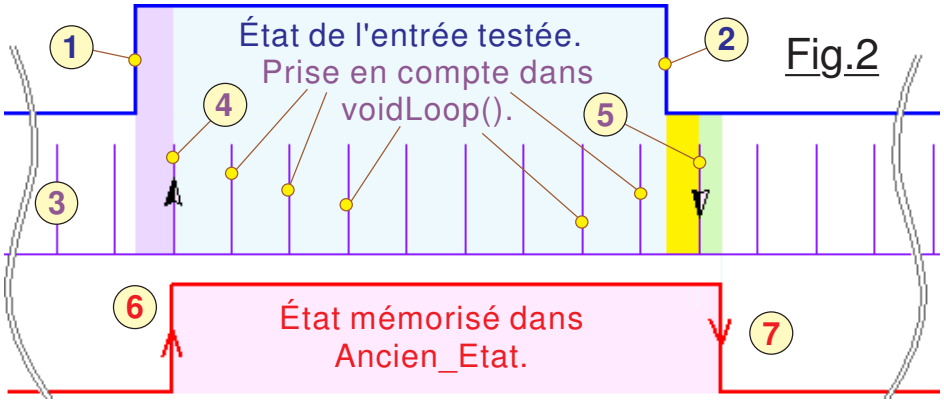


Fig.2

## *Détection d'un front sur une entrée binaire :*

Le principe est strictement identique et fait appel aux mêmes booléens que ceux utilisés pour une entrée analogique et ainsi traiter la chronologie des événements. Comme il n'y a plus de traitement à effectuer pour évaluer l'état en entrée, c'est directement le résultat de la lecture sur EntreeBinaire qui sert de test pour positionner Etat1.

```
const byte EntreeBinaire = 10;  // Broche 10 pour détecter.
boolean Etat1; // État binaire issu de la lecture sur l'entrée.
boolean Ancien_Etat; // État binaire mémorisé.

void setup() { pinMode(EntreeBinaire, INPUT);}

void loop() {
  if ((digitalRead(EntreeBinaire) == 1))
            {Etat1=true; Ancien_Etat=true;}
     else {Etat1=false;};
  // ========== Gérer le front descendant. ==========
  if ((Ancien_Etat) && (!Etat1))
    {Ancien_Etat=false; ActionSurFrontDescendant(); } ; }
```

***Cas simplifié d'un bouton poussoir.***

```
if(digitalRead(MonBouton)==LOW) {Actions}; // Front détecté.
while(digitalRead(Monbouton)==LOW); // Attente du relâcher.
```

## *Calculs de transpositions de valeurs :* *(Voir page 30)*

Fréquemment une variable variant dans une certaine plage de valeurs doit être linéairement transposée dans une autre fourchette numérique avec pour les bornes aussi bien des signes positifs que négatifs. Un exemple typique est explicité en page 24 pour transposer la valeur d'une mesure analogique [0 à 1023] en [0 à 255] apte à piloter une sortie binaire en mode PWM.

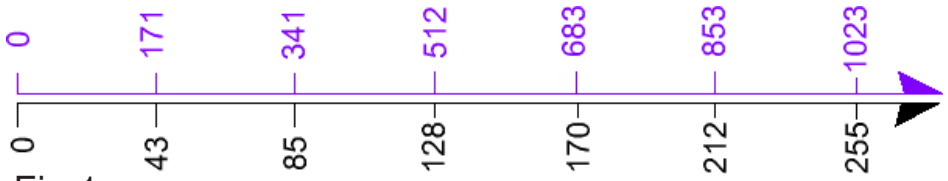
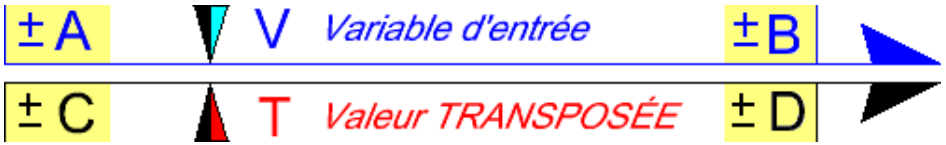



Fig.1 Transposée = (uint32_t) Analog * 255 / 1023

### *Codage pour des bornes algébriques quelconques.*

La figure ci-dessous résume la transformation à effectuer : Quand la variable **V** varie linéairement de la valeur **A** à la valeur **B**, la transposition **T** doit retourner une valeur "proportionnelle" variant linéairement entre les valeurs **C** et **D** bornes comprises. Les bornes sont toutes des entiers algébriques. Naturellement **B** et **D** sont supérieures à **A** et **C**.




Formule de calcul qui traite de tous les cas possibles.

$$T = \frac{(D - C) * (V - A)}{(B - A)} + C$$

Pour le codage Arduino il faut commencer par la multiplication pour ne pas perdre "les décimales" avec la division si les éléments sont des entiers non signés. Dans l'écriture de la formule il faut commencer par la valeur de (D-C) qui est une constante. Ainsi les parenthèses pour (V-A) n'interfèreront pas avec la directive (uintNN_t). Prendre garde dans les calculs à la valeur de NN, et particulièrement si l'une des bornes est négative et que le calcul de la multiplication conduit à des grandeurs importantes. Le programme Formules_de_TRANSPOSITION.ino est un exemple de codage avec un jeu d'essais complet. *(Voir .../...)*

| | [ V ] | | [T] | | | |
|---|---|---|---|---|---|---|
| CAS | A | B | C | D | NN | Formule de calcul |
| (1) | 0 | 1023 | -8 | 25 | 32 | V * 33 / 1023 - 8 |
| (2) | -8 | 25 | -2 | 333 | 32 | 335 * (V + 8) / 33 - 2 |
| (3) | -2 | 333 | 3 | 10 | 8 | 7 * (V + 2) / 335 + 3 |
| (4) | 3 | 10 | -50 | 5 | 16 | 55 * (V - 3) / 7 - 50 |
| (5) | -50 | 5 | -100 | -30 | 16 | 70 * (V + 50) / 55 - 100 |
| (6) | -100 | -30 | -40 | -20 | 16 | 20 * (V + 100) / 70 - 40 |
| (7) | -40 | -20 | 5000 | 10000 | 32 | 5000 * (V + 40) / 20 + 5000 |
| (8) | 5000 | 10000 | -800 | -400 | 32 | 400 * (V - 5000) / 5000 - 800 |

**Programme Formules_de_TRANSPOSITION.ino :**

```
// Définition des variables pour ce programme.
int Commande; // Valeur mesurée pour la variable V.
long TRANSPOSITION; // Pas de unsigned car signes négatifs possibles.

void loop() {
  Commande = analogRead(EntreeAnalogique)+1; // [0 à 1024].
// >>>>>>>>>>>>>>>>>> Traitement du cas (1) <<<<<<<<<<<<<<<<<<
  TRANSPOSITION = (uint32_t) Commande * 33 / 1023 - 8;
// Pas de parenthèse car priorité des opérateurs respectée.
// >>>>>>>>>>>>>>>>>> Traitement du cas (2) <<<<<<<<<<<<<<<<<<
  TRANSPOSITION = (uint32_t) 335 * (TRANSPOSITION + 8)  / 33 - 2;
// >>>>>>>>>>>>>>>>>> Traitement du cas (3) <<<<<<<<<<<<<<<<<<
  TRANSPOSITION = (uint8_t) 7 * (TRANSPOSITION + 2)  / 335 + 3;
// >>>>>>>>>>>>>>>>>> Traitement du cas (4) <<<<<<<<<<<<<<<<<<
  TRANSPOSITION = (uint16_t) 55 * (TRANSPOSITION - 3)  / 7 - 50;
// >>>>>>>>>>>>>>>>>> Traitement du cas (5) <<<<<<<<<<<<<<<<<<
  TRANSPOSITION = (uint16_t) 70 * (TRANSPOSITION + 50)  / 55 - 100;
// >>>>>>>>>>>>>>>>>> Traitement du cas (6) <<<<<<<<<<<<<<<<<<
  TRANSPOSITION = (uint16_t) 20 * (TRANSPOSITION + 100)  / 70 - 40;
// >>>>>>>>>>>>>>>>>> Traitement du cas (7) <<<<<<<<<<<<<<<<<<
  TRANSPOSITION = (uint32_t) 5000 * (TRANSPOSITION + 40)  / 20 +5000;
// >>>>>>>>>>>>>>>>>> Traitement du cas (8) <<<<<<<<<<<<<<<<<<
TRANSPOSITION = (uint32_t) 400 * (TRANSPOSITION - 5000) / 5000 - 800;
```

**NOTE :** La transposée de la Fig.1 peut aussi s'écrire : Transposée = (uint32_t) Analog * 4.88 que l'on peut lire comme le produit des incréments du CAN par sa définition, ou la constante résultant du calcul de 255 / 1023. (Code plus compact mais moins lisible)

## *Différents types de mémoires sur Arduino :*

Il existe trois types de mémoires internes aux microcontrôleurs de la famille AVR *(Atméga 168 / 328 etc ...)* équipant les cartes Arduino :

- La mémoire FLASH est la mémoire **non-volatile** dans laquelle le programme est "Téléversé" et y réside jusqu'à réécriture. Pour toutes les cartes Arduino, 2 ko de la mémoire FLASH sont mobilisés par le programme de téléchargement résidant. *(Bootloader)*
- La SRAM à accès aléatoire ou **mémoire vive**. Le programme y crée et y modifie **les variables** durant son exécution. C'est une **mémoire volatile** qui perd ses données lors de la mise hors tension.
- L'EEPROM est une mémoire **non-volatile** mise à la disposition du programmeur pour y stocker des données destinées au long-terme.

| Type | ATmega168 | ATmega328 | ATmega1280 | ATmega2560 |
|---|---|---|---|---|
| FLASH | 16 ko | 32 ko | 128 ko | 256ko |
| SRAM | 1024 Octets | 2048 Octets | 8 ko | 8 ko |
| EEPROM | 512 Octets | 1024 Octets | 4 ko | 4 ko |

La mémoire vive ne peut contenir que très peu de données. *(2Ko sur Arduino Uno)* La SRAM est rapidement saturée si l'on ne prend pas garde à la taille des variables et surtout à la déclaration des constantes "texte". La gestion d'un tableau de variables de grande dimension peut facilement devenir impossible par manque de place.

### *Utilisation de l'EEPROM pour loger des données.*

La façon la plus simple pour économiser de la place en SRAM consiste à loger en EEPROM les chaînes de caractères constantes et les tables de données constantes. Pour la procédure, voir en page 4 du livret les Méthodes de la bibliothèque EEPROM.h et consulter les deux petits programmes ECRIRE_en_EEPROM.ino et LISTER_EEPROM.ino. L'EEPROM présente un accès relativement lent puisque toute écriture ou lecture exige 3,3mS. De plus, le nombre de cycles d'écriture fiables se situe vers 100000. Cette mémoire doit en priorité être utilisée pour loger des constantes. Ce n'est que pour des données "variables" stockées sur le long terme et ne devant pas être perdues sur coupure alimentation que l'EEPROM sera employée. Mais dans ce cas, généralement la fréquence des écritures reste très modérée et compatible avec la durée de vie raisonnable du microcontrôleur utilisé dans l'application.

### *Utilisation de la mémoire FLASH pour loger des données.*

Au même titre que l'EEPROM, la mémoire FLASH prévue pour loger le programme peut également être utilisée pour y loger des données. D'un accès aussi rapide que de la RAM, elle conserve les données sur coupure d'alimentation. Mais, l'estimation de la durée de vie d'une mémoire flash se situe à environ 100000 écritures et effacements. De ce fait, il faut la réserver raisonnablement au stockage de données constantes telles que les tableaux de conversion etc. La mémoire FLASH doit être considérée comme une mémoire accessible uniquement en lecture.

```
PROGMEM type NomConstante (Valeur);
```

PROGMEM est un modificateur de variable, et ne peut s'utiliser qu'avec les types de données définis dans la librairie avr/pgmspace.h qui doit être préalablement déclarée avec un #include. PROGMEM précise au compilateur de placer les données dans la mémoire FLASH", au lieu de la SRAM, où elles devraient résider en standard.

Noter que PROGMEM étant un modificateur de variable, il n'y a pas d'emplacement obligatoire, et le compilateur d'Arduino devrait accepter toutes les définitions équivalentes. L'expérience montre que dans des versions anciennes du compilateur il y a disfonctionnement si PROGMEM est positionné après le nom de la variable :

```
Type NomConstante[] PROGMEM = {};   // Accepté.
PROGMEM  Type  NomConstante[] = {}; // Accepté.
Type PROGMEM NomConstante[] = {};   // Ne pas utiliser.
```

L'utilisation de types de données ordinaires peut conduire à des erreurs énigmatiques. *(Les float ne sont pas supportés)* Il importe donc d'utiliser impérativement les types de données déclarés dans avr/pgmspace.h :

prog_char : Type char signé sur 1 octet : -127 à 128.
prog_uchar : Type char non signé sur 2 octet : 0 à 255.
prog_int16_t : Type int signé sur 2 octet : -32767 à 32768.
prog_uint16_t : Type int non signé sur 2 octet : 0 à 65535.
prog_int32_t : Type long signé sur 4 octet : -2147483648 à 2147483647.
prog_uint32_t : Type long non signé sur 4 octet : 0 à 4294967295.

Bien que PROGMEM fonctionne avec une variable élémentaire, il est plus rentable de s'en servir avec des blocs de données à stocker, comme des tableaux ou d'autres structures de données en langage C. Exemple :

```
PROGMEM prog_uint16_t MesValeurs[] = {65432, 12345, 73, 88, 3333};
```

Le programme Tester_PROGMEM.ino donne un exemple de stockage de données en mémoire de programme.

## Gestion du "temps processeur" :

Sauf s'il est placé en veille, le microcontrôleur est occupé à 100%. Même "s'il ne fait rien" en déroulant des boucles d'attente dans lesquelles il surveille une condition de sortie, le µP n'est jamais arrêté. Le problème pour le programmeur réside dans la charge de travail en tâche de fond qui peut amener à une lenteur exagérée de rotation dans la boucle de base loop ou dans ses procédures et fonctions de service. Pour évaluer la fréquence de rafraichissement de la boucle de base ou du temps passé dans les procédures de servitude ou d'interruptions, on peut utiliser une sortie binaire que l'on fait changer d'état à convenance. Ainsi, à l'aide d'un périodemètre numérique ou d'un oscilloscope il devient facile d'estimer le temps disponible pour effectuer des traitements globaux dans loop, et celui consommé dans les procédures.

L'optimisation pour le choix du type des données et de l'architecture adoptée pour la structure globale du programme sont des incontournables à soigner pour minimiser les temps d'exécution et ceux de réaction à un événement. Cette optimisation relève d'un compromis engendré par l'utilisation de nombreuses procédures visant à favoriser la lisibilité du programme.

### Lisibilité du programme.

Si l'on désire dominer son logiciel, la logique conduit à ne trouver dans la boucle de base pratiquement que des appels à des procédures ou des fonctions dont les identificateurs sont parfaitement évocateurs. Il en est de même pour les routines de servitude. Mais chaque invocation se paye par le temps d'appel et celui de retour.

### Bien choisir le type des données.

Les microcontrôleurs de la famille AVR fonctionnent en mots de 8 bits. Il importe de favoriser à outrance les variables et les constantes de type char ou byte car tout autre choix implique de la consommation de mémoire et du "temps machine". Le traitement d'un int de16 bits est déjà beaucoup plus long qu'un simple byte car il est traité sur deux octets avec prise en compte de la retenue ou du bit de signe. C'est particulièrement vrai pour le choix de la variable d'évolution dans une boucle for.

### Prise en compte des événements courts.

Si le microcontrôleur est très chargé et qu'il "peine" dans la boucle de base, le programme risque de ne pas détecter une transition ou un état temporaire court sur l'une des entrées à surveiller.
Il faut considérer que chaque valeur que l'on veut traiter *(État des broches, valeur des "Timers", données reçues sur un port série, valeur analogique convertie etc)* est mise à jour dans des registres constamment rafraichis par les "événements extérieurs". C'est au programme que revient la responsabilité de prendre en compte ces valeurs fugitives avant leur modification, donc par du code suffisamment réactif pour ne pas perdre un événement significatif.
Mis à part certaines Interruptions, il n'y a pas de tampons d'attente dans Arduino. Les registres peuvent changer à tout moment sans prévenir. Chaque fonction matérielle est un périphérique externe au CPU, associée à plusieurs registres. Il revient au programme d'en vérifier le contenu et de le traiter avec une fréquence suffisante.

### Gestion des événements.

Sans que nous en soyons forcément conscients, le microcontrôleur est constamment sollicité par des requêtes d'Interruptions issues des périphériques connectés que ce soit en interne ou en externe. *(Timers, voie série etc)* Dans la boucle de base, le listage ne montre souvent que des instructions "simples" qui masquent la charge réelle de travail du microcontrôleur. Cette charge de travail peut alors aboutir à un ralentissement significatif de loop si l'on procède par examen cyclique des nombreuses entrées du système. Pour les événements courts qui peuvent être perdus, il faut alors procéder par Interruptions pour éviter les pertes de leur traitement. C'est dans ce cas qu'une sortie qui bascule dans les séquences d'interruption ISR ou dans la boucle de base loop permet d'optimiser le code en phase de développement.

### Notes sur l'utilisation des interruptions. *(Rappels)*

- 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 en différé à la boucle de base ou à l'une de ses procédures de servitude.