université de BORDEAUX

geii IUT de BORDEAUX

Mesures physiques

Formation Arduino

Initiation au développement sur Arduino UNO

Damien BLANCHARD

Version initiale 2014

Modifiée et corrigée en 2015

Allégée et modifiée en 2018

v 0.3

Table des matières

Licence du document

Le document d'origine a été écrit pour la formation du réseau des électroniciens et instrumentalistes de la DR15 (CNRS). Cette version en est une version allégée.

Introduction

Si l'on devait définir très brièvement *Arduino* que pourrait-on dire ? Nous pourrions juste énoncer que c'est un système de programmation simplifié pour microcontrôleur. Pourtant nous serions loin du compte, et surtout nous serions incapable d'expliquer son incroyable succès. Car de la programmation simplifiée pour microcontrôleur cela existe depuis déjà fort longtemps. Pratiquement toutes les familles de microcontrôleurs ont eu droit à un compilateur utilisant soit un langage simplifié, comme le PICBASIC, soit des ensembles de bibliothèques facilitant la mise en œuvre. Cependant leur complexité restait très supérieure au système d'*Arduino*.

Arduino c'est un ensemble : une carte à microcontrôleur, peu onéreuse, programmable directement par *USB*, donc il n'est plus nécessaire d'acquérir une coûteuse sonde de programmation, un environnement de développement avec un compilateur libre et gratuit, contrairement aux solutions évoquées précédemment qui étaient payantes et chères, et un « langage » extrêmement simplifié sans pour autant brider les possibilités du microcontrôleur. Seul manque un *debugger*.

Voyons à quoi ressemble une carte *Arduino* UNO, celle que vous avez pour la formation :

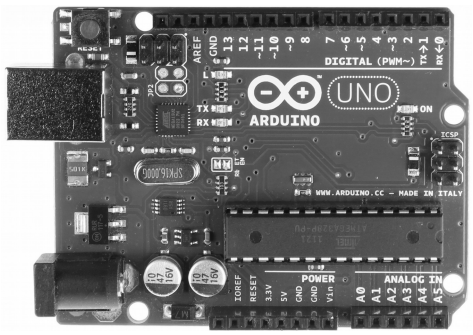


Figure 1: Carte Arduino Uno

Ce que je n'ai pas cité dans l'ensemble et qui pourtant est un point crucial : sa philosophie d'ouverture. *Arduino* est libre et ouvert de bout en bout : les schémas des cartes sont disponibles, on peut construire ses propres cartes compatibles *Arduino*. Le *bootloader* (ce qui permet au microcontrôleur d'être programmé depuis l'*USB*) est libre, l'*IDE* l'est aussi et il repose sur des outils eux-mêmes libres. De même, les *shields,* c'est-à-dire les cartes périphériques qui se branchent dessus sont libres, et n'importe qui peut en créer et en vendre sans devoir s'acquitter d'une licence.

Ici j'ai évoqué des termes qui peut-être ne vous sont pas familiers, ne vous inquiétez pas nous allons y revenir, penchons nous d'abord sur son ouverture en évoquant rapidement l'histoire d'*Arduino*.

Un jour un grand homme, au sens propre comme au figuré, m'a dit *« J'ai passé l'âge de manipuler des bits »*. C'est pour moi une des meilleures raisons d'adopter *Arduino*.

Nous avons opté pour la création d'un *shield* propre à la formation afin de placer dessus la plupart des périphériques qui nous ont semblé nécessaires à une bonne initiation à cette plateforme. Ce *shield* a la particularité d'être aussi prévu pour être utilisé pour apprendre à faire des asservissements de température.

Ce *shield* a entièrement été conçu par mon collègue Martial Leyney qui s'est aussi entièrement chargé de sa fabrication. Je le remercie vivement pour l'excellent travail qu'il a ainsi effectué.

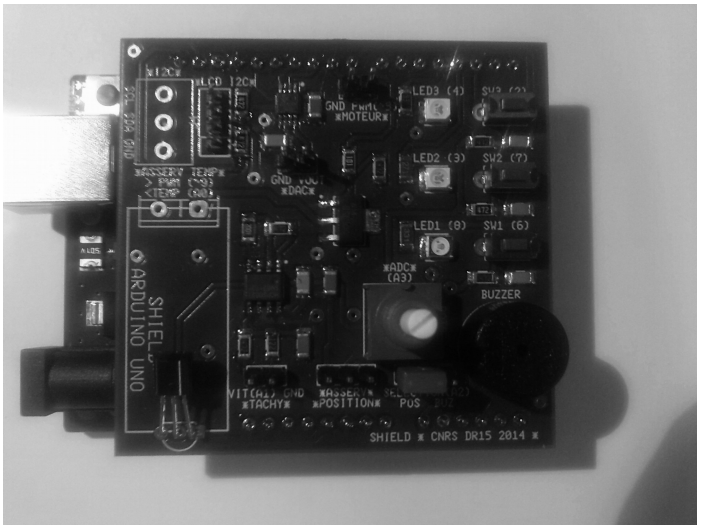


Figure 2: Shield spécifique à la formation

Nous avons sur ce *shield* les périphériques suivants :

- 3 boutons poussoirs, sur les *pins* 2, 6 et 7,
- 3 LEDs, sur les *pins* 4, 3 et 8,
- une résistance chauffante pilotée par un *MOSFET* sur la *pin* 5,
- un connecteur (J6) pour la *PWM* de la *pin* 5,
- un convertisseur numérique-analogique *SPI (MCP4921)* sur les *pins* 10, 11, 12 et 13,
- un capteur de température LM35DZ sur l'entrée analogique A0,
- un connecteur (J1) pour l'entrée analogique A1,
- un connecteur (J2) pour la *pin* A2 partagée avec un *buzzer*, la sélection se faisant par un cavalier sur le connecteur J3,
- potentiomètre relié sur l'entrée analogique A3,
- un connecteur pour un écran *LCD I2C (Grove 811004001)*,
- un connecteur supplémentaire pour relier d'autres périphériques *I2C*.

1. Les bases du langage

Arduino n'est pas à proprement parler un langage mais repose sur les langages C/C++. Deux approches étaient alors possibles : soit aborder en premier les langages C et C++ puis voir les éléments propres à *Arduino* soit aborder le tout comme un ensemble. C'est cette deuxième approche qui a été retenue car même si elle est un peu plus brouillonne, elle semble plus apte à une approche pédagogique fondée sur la pratique. Dans ce chapitre les notions seront vues très succinctement de façon à permettre à quiconque de comprendre et de prendre ses marques. Dans le chapitre suivant nous reviendrons sur ces notions de façon plus approfondie.

1.1. Le strict minimum : le programme qui ne fait rien

Ceux qui ont l'habitude du C ou du C++ peuvent oublier la fonction `main` car il n'y en a pas avec *Arduino* !

A la place nous avons 2 fonctions :

`setup` où l'on va placer toutes les instructions qui ne seront appelées qu'une seule fois lors du démarrage du programme (aussi appelé « *sketch* » ou « croquis » en français),

`loop` qui va contenir toutes les instructions du déroulement du programme, si cette fonction arrive à sa fin elle redémarre automatiquement à son début.

Regardons comment s'écrivent ces deux fonctions :

```
// le minimum pour compiler
void setup ()
{

}

void loop ()
{
}
```

Sans entrer dans les détails, puisque nous aborderons l'écriture de nos propres fonctions plus tard, nous pouvons voir qu'ici il y a `void`, qui nous importe peu pour l'instant, suivi du nom de la fonction, puis une parenthèse ouverte et une fermante. Ensuite viennent deux accolades, une ouverte et une fermante. C'est entre ces accolades que nous allons placer nos lignes de codes.

Il est à noter que les retours à la ligne ne font pas partie du langage, nous aurions pu mettre les accolades ouvrantes sur la même ligne que celle du nom de la fonction.

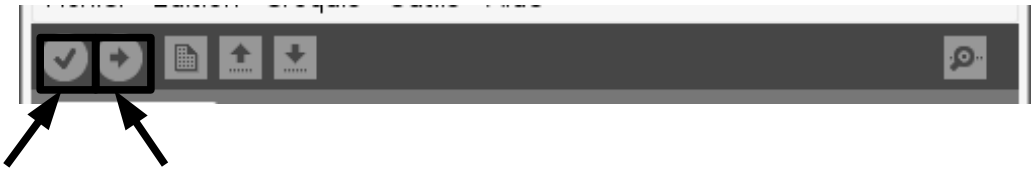
La première ligne est un commentaire, c'est-à-dire, du texte qui sera ignoré par le compilateur. On peut donc écrire ce que l'on veut. Pour faire un commentaire il suffit de placer deux slashs `//` et tout ce qui est après, jusqu'à la fin de la ligne, sera considéré comme un commentaire.

Il est possible de faire un commentaire englobant plusieurs lignes. Pour cela il faut commencer par `/*` et terminer par `*/`. Tout ce qui est entre ces deux séquences sera considéré comme un commentaire.

Cet exemple correspond à la définition de deux procédures telles qu'elles apparaissent dans le cours d'algorithmie :

```
void setup ()              // Procédure setup
{                          // Début
}                          // Fin
```

Nous sommes prêt à compiler et à charger ce programme dans la carte *Arduino*.



Compile Compile et téléverse

1.2. Allumer une LED

Pour pouvoir utiliser une *pin* en entrée numérique ou en sortie numérique il faut l'indiquer au microcontrôleur en faisant appel à la fonction `pinMode()`.

Son utilisation est très simple et en voici un exemple :

```
pinMode (4, OUTPUT);
```

Le premier argument (appelé aussi « paramètre ») est le nom de la *pin* à initialiser. Chaque *pin* de l'Arduino a un nom : de 0 à 13 pour les *I/O* uniquement numériques et de A0 à A5 pour les *pins* pouvant être soit des *I/O* numériques soit des entrées analogiques.

Le deuxième argument peut-être soit `OUTPUT` pour mettre en sortie, soit `INPUT` pour mettre en entrée.

On remarque la présence d'un point virgule à la fin de la ligne. Il est très important, il indique au compilateur que c'est la fin d'une instruction, ici l'appel de la fonction.

Algorithmiquement cette fonction pourrait être déclarée ainsi :

```
Fonction pinMode (pin : nombre, etat : nombre) : aucun
```

`Remarque` : par défaut, lors du *reset* les *pins* sont toutes configurées en entrée numérique.

Maintenant que nous avons configuré la *pin* nous allons voir comment assigner un état. Pour cela on fait appel à la fonction `digitalWrite()` de cette façon :

```
digitalWrite (4, HIGH);
```

De la même façon que la fonction `pinMode()` le premier argument est le nom de la *pin*.

Le deuxième argument peut être soit `HIGH` pour mettre à l'état haut c'est-à-dire à la tension d'alimentation, soit `LOW` pour mettre à l'état bas c'est-à-dire 0V.

Algorithmiquement cette fonction pourrait être déclarée ainsi :

```
Fonction digitalWrite (pin : nombre, etat : nombre) : aucun
```

`Remarque` : chaque *pin* peut fournir ou absorber jusqu'à 40mA chacune mais le courant total doit être inférieur à 200mA.

Voyons donc ce que cela donne :

```
void setup ()
{
  pinMode (4, OUTPUT);
}

void loop ()
{
  digitalWrite (4, HIGH);
}
```

Ce qu'il est important de savoir c'est que toutes les instructions, bien qu'ici il n'y en ait qu'une, sont exécutées dans l'ordre dans lequel elles sont écrites. Et toutes les instructions doivent être écrites à l'intérieur d'une fonction.

Il serait plus intéressant de pouvoir faire clignoter la LED, donc nous avons besoin de savoir comment faire une pause.

Arduino est très bien fourni de ce côté là. Nous avons à notre disposition les fonctions `delay()` et `delayMicroseconds()`.

Ces deux fonctions ne prennent qu'un argument qui est le temps, respectivement en millisecondes et en microsecondes, que doit durer la pause.

Exercice :

A partir de l'exemple précédent faites en sorte de faire clignoter la LED toutes les 500ms.

Remarque :

Bien que vous n'allumiez et n'éteigniez la LED qu'une seule fois elle clignote car la fonction `loop()` est relancée chaque fois qu'elle est terminée.

Exercice :

Réalisez un chenillard avec les LEDs qui sont sur les *pins* 4, 3 et 8.

1.3. Premier aperçu des variables

Dans l'exercice précédent vous avez pu remarquer que vous avez répété le nom de la *pin* plusieurs fois. Ceci n'est vraiment pas très pratique si jamais vous voulez à un moment en utiliser une autre, vous serez obligé de modifier chaque ligne.

Fort heureusement il existe plusieurs façons de pallier ce problème. Par exemple nous pourrions passer par une variable.

Commençons par définir ce qu'est une variable : c'est tout simplement un espace mémoire, de taille fixe, auquel on donne un nom et dans lequel on va pouvoir stocker des informations.

Il existe plusieurs types de variables suivant le type de données que l'on veut y placer, mais pour l'instant nous ne verrons que le `int`, qui peut contenir des entiers positifs et négatifs (on dit alors qu'elle est signée), nous aborderons les autres par la suite.

Pour créer une variable il suffit d'écrire son type, ici `int` suivi du nom que l'on veut donner à la variable et terminer par un point virgule.

Exemple :

```
int maLed;              // maLed : nombre
```

Cette ligne peut être placée à différents endroits dans le programme. Cependant suivant cet emplacement le comportement sera différent. Nous aborderons ceci lorsqu'on nous verrons les variables plus en détail. Donc pour le moment il vous suffit de l'écrire au tout début du code, avant les fonctions `setup()` et `loop()`.

Il était précédemment évoqué que toutes les instructions devaient être faites à l'intérieur d'une fonction, ceci est une exception.

Lors de la déclaration de la variable il n'y a pas de valeur assignée par défaut, la valeur contenue sera celle présente précédemment dans la *RAM* et peut donc être n'importe quoi.

Pour assigner une valeur il suffit d'utiliser l'opérateur = en faisant :

```
maLed = 4;              // maLed ← 4
```

ou directement lors de la déclaration :

```
int maLed = 4;          // maLed : nombre
                        // maLed ← 4
```

Remarque : on ne peut pas donner n'importe quel nom à une variable, par exemple ce ne doit pas être un mot-clés comme `int`, il ne doit pas contenir d'espace ni de caractères spéciaux ni même de caractères accentués. Il ne doit pas non plus commencer par un chiffre. En revanche on peut utiliser l'*underscore* _ et les majuscules.

Exercice :

Modifiez votre 1er programme pour que le nom de la *pin* ne soit plus écrit directement dans les appels de fonctions mais passe par une variable.

Faites de même pour le délai de pause.

Bien que cette méthode soit présente dans la plupart des exemples d'Arduino, il est déconseillé de l'employer. Préférez plutôt l'utilisation d'un nom symbolique via `#define`, à placer tout au début du sketch.

Exemple :

```
#define LED 4

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

void loop()
{
    digitalWrite (LED, HIGH);
}
```

1.4. Lire l'état d'une pin

Nous avons vu que nous pouvions mettre en sortie des *pins* via `pinMode()`. Nous avons rapidement évoqué la possibilité de la mettre en entrée. Essayons cela maintenant.

Pour lire l'état d'une *pin* en entrée, il faut faire appel à la fonction `digitalRead()` de la façon suivante :

```
digitalRead (7);
```

Cette fonction prend comme seul argument le nom de la *pin* à lire.

Algorithmiquement cette fonction pourrait être déclarée ainsi :

```
Fonction digitalRead (pin : nombre) : nombre
```

L'état est retourné par la fonction. Pour récupérer cette valeur c'est très facile, il faut passer par une variable comme ci-dessous :

```
int etat;                       // etat : nombre
etat = digitalRead (7);         // etat ← digitalRead(7)
```

La variable `etat` va donc contenir l'état de la *pin*. Soit `HIGH` soit `LOW`.

`Remarque` : les boutons poussoirs câblés sur le *shield* sont câblés en logique inversée, c'est-à-dire que lorsque le bouton est relâché l'entrée est à l'état haut (*`HIGH`*), et lorsqu'on appuie sur le bouton l'entrée est à l'état bas (*`LOW`*).

`Exercice :`

Modifiez le précédent programme pour que la LED s'allume quand le bouton est relâché, et s'éteigne quand on appuie sur le bouton.

`Remarque :` il est possible d'inverser un état (`HIGH` ou `LOW`) en utilisant l'opérateur '`!`' cette manière :

```
etat = !digitalRead(7);
```

`Exercice :`

Utilisez l'opérateur '!' pour allumer la LED quand le bouton est enfoncé.

1.5. Premier aperçu des structures conditionnelles

a. Le if

Vous avez trouvé l'astuce pour changer l'état d'une sortie à partir de l'état d'une entrée. En revanche nous ne pourrions pas faire clignoter la LED avec ce procédé. Pour cela il nous manque la possibilité d'exécuter une partie de code selon des conditions.

Pour réaliser ceci nous allons utiliser la structure conditionnelle `if`.

Sa syntaxe est très simple et devrait vous rappeler celle des fonctions :

```
if (condition)           // Si condition
{                        // Début
}                        // Fin
```

La condition à évaluer est placée entre parenthèses. Si elle est vraie, c'est-à-dire différente de 0, alors les instructions contenues dans le bloc (pour rappel cela signifie entre les accolades) sont exécutées. Si la condition est fausse, c'est-à-dire égale à 0, alors le programme passe aux instructions qui sont après le bloc, donc après l'accolade fermante.

Pour écrire une condition nous avons à notre disposition quelques opérateurs :

Opérateurs	Conditions
`valeur1 == valeur2`	Vraie si valeur1 et valeur2 sont égales
`valeur1 != valeur2`	Vraie si valeur1 n'est pas égale à valeur2
`valeur1 < valeur2`	Vraie si valeur1 est inférieure à valeur2
`valeur1 <= valeur2`	Vraie si valeur1 est inférieure ou égale à valeur2
`valeur1 > valeur2`	Vraie si valeur1 est supérieure à valeur2
`valeur1 >= valeur2`	Vraie si valeur1 est supérieure ou égale à valeur2

ATTENTION : il est très courant quand on débute de confondre l'opérateur = avec l'opérateur == pourtant ils n'ont rien en commun. L'opérateur = permet d'assigner une valeur alors que l'opérateur == va comparer des valeurs.

Remarque : `valeur1` et `valeur2` peuvent être des variables, des valeurs numériques etc.

Voici un exemple d'utilisation pour allumer la LED si le bouton est appuyé :

```
int bouton = 7;                          // bouton : nombre
                                         // bouton ← 7
int led = 4;                             // led : nombre
                                         // led ← 4
void setup ()                            // Procédure setup
{                                        // Début
    pinMode (bouton, INPUT);             //    pinMode (bouton, INPUT)
    pinMode (led, OUTPUT);               //    pinMode (led, OUTPUT)
}                                        // Fin

void loop ()                             // Procédure loop
{                                        // Début
    int etat = digitalRead (bouton);     //     etat : nombre
                                         //     etat ← digitalRead (bouton)
    if (etat == LOW)                     //     Si etat égal LOW
    {                                    //     Début
        digitalWrite (led, HIGH);        //        digitalWrite (led, HIGH)
    }                                    //     Fin du Si

    if (etat == HIGH)                    //     Si etat égal HIGH
    {                                    //     Début
        digitalWrite (led, LOW);         //         digitalWrite (led, LOW)
    }                                    //     Fin du Si
}                                        // Fin
```

Vous remarquerez qu'on est obligé d'utiliser deux `if`, une fois pour allumer la LED si le bouton est appuyé et une fois pour l'éteindre si le bouton est relâché. En réalité, ce n'est pas la meilleure façon de faire. Il existe dans le langage la possibilité d'exécuter un autre bloc si la condition est fausse.

Il suffit pour cela de mettre après l'accolade fermante du bloc du `if` le mot-clé `else`.

Le programme serait donc écrit ainsi :

```
void loop ()                          // Procédure loop
{                                     // Début
    int etat = digitalRead (bouton); //     etat = nombre
                                      //     etat ← digitalRead(bouton)
    if (etat == LOW)                  //     Si etat egal LOW
    {                                 //     Début
        digitalWrite (led, HIGH);     //         digitalWrite(led,HIGH)
    }                                 //     Fin du Si
    else                              //     Sinon
    {                                 //     Début
        digitalWrite (led, LOW);      //         digitalWrite(led,LOW)
    }                                 //     Fin du si
}                                     // Fin
```

`Exercice :` modifiez votre précédent programme pour que la LED clignote lorsque le bouton est appuyé et qu'elle soit éteinte lorsque le bouton est relâché.

b. while

Vous avez remarqué que votre programme fonctionne car la fonction `loop()` est exécutée de nouveau chaque fois qu'elle se termine, faisant ainsi une boucle (d'où le nom de *loop*).

Or il est fréquent d'avoir besoin de faire une boucle à l'intérieur de cette boucle.

Regardons la boucle `while`.

Sa syntaxe est très proche du `if` :

```
while (condition)                 // Tant que condition
{                                 // Début
}                                 // Fin
```

Tant que la condition examinée est vraie les instructions contenues dans le bloc seront exécutées, lorsqu'elle sera fausse alors le programme ira à l'instruction après l'accolade fermante.

`Remarque` : fréquemment nous avons besoin de faire une boucle infinie, c'est-à-dire qui possède une condition toujours vraie, par exemple pour bloquer l'exécution du programme. On écrira alors `while (1)`.

`Exercice` : modifiez votre précédent programme pour que la LED clignote lorsque le bouton est appuyé et qu'elle s'éteigne lorsque le bouton est relâché mais cette fois-ci en utilisant aucun `if` et en utilisant seulement un `while`.

1.6. Conversion analogique numérique

Les microcontrôleurs utilisés par la plateforme *Arduino* disposent de plusieurs entrées analogiques. C'est-à-dire que l'on peut mettre une tension qui ne sera pas interprétée comme étant un état haut ou un état bas, un 1 ou un 0, mais peut être mesurée. Sur l'*Arduino* UNO il y en a 6 disponibles nommées de A0 à A5. Ces *pins* peuvent néanmoins être utilisées comme entrées / sorties numériques.

Remarque : A0, A1, ..., A5 sont en réalité des nombres. Sur *Arduino UNO* A0 vaut 14, A1 vaut 15, ... et A5 vaut donc 19.

L'utilisation de la conversion analogique se fait par la fonction `analogRead()`. A l'instar de la fonction `digitalRead()` elle prend en paramètre le nom de la *pin* et retourne la valeur lue.

Exemple :

```
int tension;
tension = analogRead (A3);
```

Dans cet exemple nous lisons la valeur de la tension présente sur la *pin* A3. La valeur retournée par la fonction `analogRead()` est placée dans la variable `tension`.

Cette valeur va de 0 (0V) à 1023 (5V).

Cette valeur de 5V correspond à la tension de référence de la conversion. Il est possible de la changer. Deux solutions s'offrent à nous :

- utiliser la ligne suivante `analogReference(INTERNAL);` pour que la tension de référence soit de 3.3V,

- utiliser la ligne suivante `analogReference(EXTERNAL);` pour que la tension de référence soit celle présente sur la *pin* AREF d'*Arduino*. Cependant il vous faudra placer une résistance de 5KΩ entre votre tension et la *pin* AREF pour éviter d'endommager le microcontrôleur.

Changer cette tension de référence permet d'obtenir un résultat plus précis puisque le pas de quantification sera lui aussi plus petit (le pas de quantification est égal à la tension de référence divisée par 1024). La valeur maximale 1023 sera retournée lorsque la tension d'entrée sera égale à la tension de référence.

Exercice : lisez la tension du potentiomètre (*pin* A3). Allumez la LED sur la *pin* 4 si la tension est inférieure à 2.5V , allumez la LED sur la *pin* 7 si elle est supérieure ou égale à 2.5V.

Indice : le calcul des seuils se fait par le calcul suivant :

```
tension = (analogRead (A3) * 5) / 1023;
```

Arduino dispose de la fonction map() permettant de réaliser facilement ces conversions. Elle s'utilise de la fonction suivante :

```
conversion = map (valeur, minAvant, maxAvant, minApres, maxApres);
```

où `valeur` est la valeur de départ à convertir,

`minAvant` la valeur minimale avant conversion,

`maxAvant` la valeur maximale avant conversion,

`minApres` la valeur minimume après conversion,

`maxApres` la valeur maximale après conversion.

Attention cependant car cette fonction ne traite que des nombres entiers, ainsi pour l'exercice précédent nous sommes obligés de multiplier par 10 pour conserver un chiffre après la virgule. Nous aurions alors 0 pour `minAvant` et 1023 pour `maxAvant` puisque ce sont les valeurs minimales et maximales renvoyées par la fonction `analogRead()`, 0 pour `minApres` et 50 pour `maxApres`.

Exemple complet :

```
#define LED3 4
#define LED2 3
#define POTENTIOMETRE A3

void setup()
{
    pinMode (LED2, OUTPUT);
    pinMode(LED3, OUTPUT);
}

void loop()
{
    int potar;
    int tension;

    potar = analogRead (POTENTIOMETRE);
    tension = map (potar, 0, 1023, 0, 50);
    if (tension < 25)
    {
        digitalWrite (LED2, LOW);
        digitalWrite (LED3, HIGH);
    }
    else
    {
        digitalWrite (LED3, LOW);
        digitalWrite (LED2, HIGH);
    }
}
```

Exercice :

En utilisant la fonction `map()`, faites clignoter la LED sur la *pin* 4 de 10ms à 2s selon la valeur du potentiomètre.

1.7. Liaison série

Ce qui est intéressant avec *Arduino* c'est que la liaison qui permet de le programmer permet aussi très simplement l'envoi et la réception de données lors de l'exécution du programme.

Pour l'instant nous allons nous contenter d'envoyer des données de la carte *Arduino* vers le *PC*. Nous verrons dans le chapitre suivant comment réceptionner les données, et comment formater plus efficacement les données à envoyer.

Pour pouvoir utiliser la liaison série, puisque c'est de cette liaison qu'il s'agit, il nous faut utiliser la fonction `Serial.begin()`. Cette fonction prend un paramètre qui est le *baudrate* (vitesse de transmission). Ces vitesses sont normalisées, et la plupart du temps nous utilisons la valeur 9600, mais d'autres fonctionnent tout aussi bien par exemple : 4800, 19200 et même 115200. Cette fonction n'est à appeler qu'une seule fois, donc généralement elle est placée dans la fonction `setup()`.

Exemple :

```
void setup ()
{
    Serial.begin (9600);
}
```

Ensuite pour transmettre des données nous pouvons utiliser la fonction `Serial.println()`.

```
Serial.println (valeur, mode);
```

où

`valeur` est un entier (par exemple un type `int` que nous avons déjà vu),

`mode` est un paramètre optionnel (c'est-à-dire que l'on peut ne pas mettre), qui peut avoir la valeur :

`DEC` pour envoyer en base 10,

`HEX` pour envoyer en hexadécimal,

`BIN` pour envoyer en binaire.

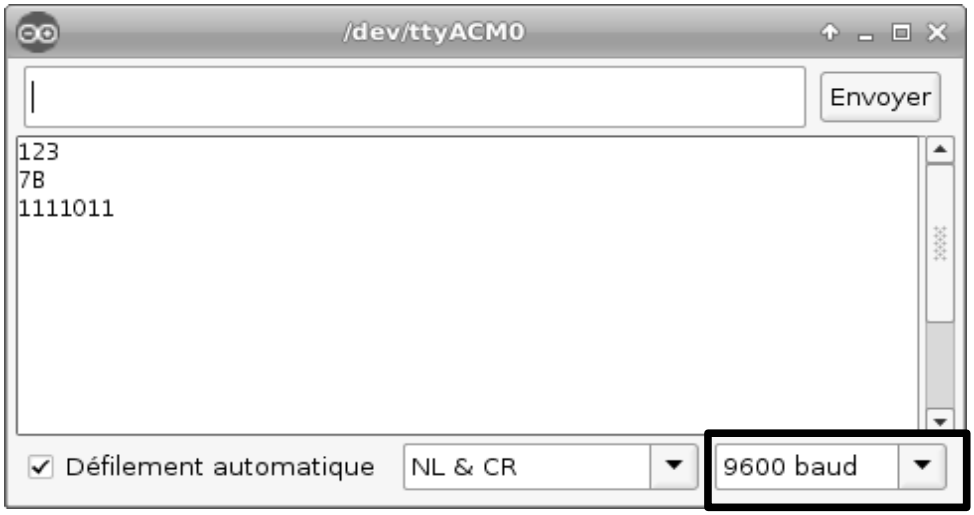
Par défaut, c'est-à-dire si on omet le mode l'envoi sera fait en décimal (base 10).

Exemple :

```
void loop ()
{
    Serial.println (123);
    Serial.println (123, HEX);
    Serial.println (123, BIN);
    while (1); // boucle infinie pour bloquer l'exécution du programme
}
```

Pour accéder au terminal série il faut cliquer sur l'icône en haut à droite dans le programme *Arduino*.

Vous devriez voir ceci dans le terminal



Pensez à bien vérifier que le *baudrate* dans le terminal est le même que celui indiqué dans le code.

Ce que vous voyez affiché sont 3 représentations différentes du même nombre.

`Exercice` : lisez la valeur du potentiomètre et envoyez là sur le terminal. Pensez à mettre une petite pause (par exemple 200ms) afin que la valeur soit bien lisible dans le terminal.

Ceci termine nos premiers pas. J'espère qu'il n'aura pas découragé ceux qui s'initient à la programmation et que ceux qui connaissent déjà, au moins un peu, auront eu l'excellente idée de sauter ce chapitre.

2. L'art du code

Il est nécessaire, avant de voir plus en détail le langage lui-même, d'acquérir les bons réflexes qui permettent de faire du code lisible, fiable et qui fonctionne rapidement.

Avant d'écrire la moindre ligne de code, il est bon de réfléchir à ce que l'on souhaite faire et ne pas se lancer directement dans le code. Pour les touts petits programmes, il est évident que c'est rarement nécessaire, cependant lorsque les programmes gagnent en complexité il faut prendre le temps d'écrire sur une feuille comment on va architecturer le programme. Nous avons chacun un mode de pensée qui nous est propre, il n'y a donc pas de méthode universelle. Certains préféreront écrire tout un organigramme, d'autres juste griffonner des idées hiérarchisées ou encore écrire du pseudo code. Aucune méthode de réflexion n'est mauvaise tant qu'elle est structurée et synthétique.

Ensuite, il faut comprendre que le C est très souple sur la syntaxe, la majorité des espaces ne sont pas obligatoires, les retours à la ligne non plus et il est donc possible de mettre plusieurs instructions sur une même ligne tant qu'elles sont bien terminées par un point virgule. Cependant c'est une très mauvaise chose. Pour des programmes très simple et très court on ne voit pas la nécessité d'adopter des règles d'écriture du code. Mais ces mauvaises habitudes vont rester lors de l'écriture d'un programme plus conséquent. Il est préférable d'écrire proprement le code dès les premières lignes.

Nous voyons dans les exemples fait depuis le début que les instructions qui appartiennent à un bloc (un bloc étant délimité par les accolades) contiennent une tabulation, ceci s'appelle l'indentation. Il est très important de bien délimiter visuellement le contenu de chaque bloc afin que même sans lire vraiment le code on puisse comprendre comment il est architecturé.

Il vaut mieux aussi mettre des espaces entre les opérateurs, par exemple :

```
var = var1 + var2;
```

plutôt que d'écrire

```
var=var1+var2;
```

Le troisième réflexe est la non utilisation systématique du compilateur (et de l'exécution) pour vérifier que le programme fonctionne. Il faut toujours se forcer à relire son code et à exécuter mentalement, pas-à-pas, afin de vérifier que le programme fait bien ce que l'on voulait lors de son écriture. Adopter ce réflexe permet de comprendre beaucoup plus vite le langage, et permet d'éliminer très rapidement la majorité des problèmes plutôt que de passer son temps à modifier au hasard puis à compiler et exécuter.

Et enfin, lorsqu'on teste un programme il faut toujours tester avec plusieurs valeurs, en essayant de choisir celles qui ont le plus de chances d'être mal traitées comme les extrêmes.

3. Plus loin vers des bases plus solides

Dans ce chapitre nous allons reprendre les notions vues précédemment mais nous allons détailler les possibilités qui nous sont offertes.

3.1. Liaison série

Nous avons vu rapidement comment envoyer des données via la liaison série de la carte *Arduino*. Comme vous avez pu le constater chaque donnée étaient affichée sur une nouvelle ligne. Évidemment *Arduino* permet un peu plus de flexibilité.

Pour qu'il n'y ait pas de saut de ligne après l'envoi il suffit d'utiliser la fonction `Serial.print()` qui s'utilise de la même façon de la fonction `Serial.println()`.

Il est important à ce stade de faire une mise au point et de différencier les types de données. Lorsque l'on écrit `42` dans un programme cela signifie « la valeur 42 » si nous voulons écrire un message, c'est-à-dire du texte, nous devons le mettre entre guillemets (*double-quotes*) par exemple :

```
"ceci est un message"
```

On comprend donc que `42` est différent de `"42"`, car `"42"` est une chaîne de caractères, c'est-à-dire une succession de caractères dont le dernier est le caractère de fin de chaîne de valeur 0.

En C et C++, et donc sous *Arduino*, un caractère s'écrit entre apostrophes (*simple-quotes*) de cette façon :

```
'4'
```

Donc `"42"` est la succession de `'4'` `'2'` et `0`.

Chaque caractère est en réalité un entier entre 0 et 127, par exemple `'4'` a la valeur 52 et `'2'` la valeur 50. Donc `"42"` est aussi équivalent à la succession des octets 52, 50 et 0.

La table *ASCII*, c'est-à-dire la table de correspondance des caractères avec leur valeur, est disponible en annexe (source : *asciitable.com*).

Si dans la console série d'*Arduino* nous avons bien du texte alors que nous envoyons un octet c'est parce qu'*Arduino* fait tout seul la conversion. C'est d'ailleurs pour cela que l'on peut spécifier le mode (binaire, hexadécimal, ou décimal) puisque s'il envoyait directement la valeur il n'y aurait aucune différence.

Les fonctions `Serial.println()` et `Serial.print()` peuvent prendre en paramètre aussi bien un entier, qu'un caractère, qu'une chaîne de caractères ou qu'un nombre à virgule.

Exemples :

```
Serial.println ("42");
Serial.println (42);
Serial.print ('4');
Serial.println ('2');
Serial.println (42.0);
Serial.print ("42\r\n");
```

`Remarque` : on peut faire nous-même un retour à la ligne via la séquence d'échappement `\r\n` comme dans la dernière ligne de l'exemple.

Dans certains cas nous avons besoin de transférer directement la véritable valeur de l'octet et non pas la chaîne de caractères correspondantes. Pour cela on doit faire appel à la fonction `Serial.write()` qui transmet sans aucune conversion :

```
Serial.write (42);
```

Vous devriez voir * dans le terminal série d'*Arduino*, car dans la table *ASCII* ce caractère a la valeur 42.

Cette liaison série permet aussi de récupérer des informations venant de l'ordinateur et pas uniquement d'en envoyer.

La fonction `Serial.available()` renvoie le nombre d'octets qui ont été reçus et gardé dans la mémoire tampon (limitée à 64 octets) :

```
int recus;
recus = Serial.available();
```

La fonction `Serial.read()` renvoie le premier octet reçu qui a été gardé dans la mémoire tampon :

```
int octet;
octet = Serial.read();
```

Regardons un exemple concret : nous allons renvoyer, sur la liaison série, tous les octets reçus.

```
int octet;

void setup()
{
    Serial.begin (9600);
}

void loop ()
{
    while (Serial.available() != 0)
    {
        octet = Serial.read();
        Serial.write (octet);
    }
}
```

3.2. Les variables

a. types de variables

Comme nous l'avons vu précédemment les variables sont des espaces mémoires dans lesquels nous pouvons lire et écrire des données. Nous n'avons pour l'instant vu qu'un type de variable, le type `int` qui permet de contenir des entiers. Cependant il existe plusieurs, pour ne pas dire beaucoup, de types d'entiers. Nous allons les voir maintenant.

Le tableau suivant est valable pour toutes les cartes basées sur un *ATMega* (*Arduino Uno, micro, nano, leonardo, esplora* mais pas la *DUE*).

type	description	propriétés	A retenir (environ)
`boolean`	sur 8 bits ne prend que 2 valeurs	false ou true	
`int`	nombre entier (16 bits) signé	de -2^{15} à $2^{15}-1$	-32000 à +32000
`byte`	nombre entier (8 bits) non signé	de 0 à 2^8-1	0 à 255
`char`	nombre entier (8 bits) signé, caractère	de -2^7 à 2^7-1	-128 à +127
`unsigned int`	nombre entier (16 bits) non signé	de 0 à $2^{16}-1$	0 à 65000
`word`	nombre entier (16 bits) non signé	de 0 à $2^{16}-1$	0 à 65000
`unsigned char`	nombre entier (8 bits) signé	de 0 à 2^8-1	0 à 255
`short int`	nombre entier (16 bits) signé	de -2^{15} à $2^{15}-1$	-32000 à +32000
`unsigned short int`	nombre entier (16 bits) non signé	de 0 à $2^{16}-1$	0 à 65000
`long int`	nombre entier (32 bits) signé	de -2^{31} à $2^{31}-1$	-2 milliards à +2 milliards
`unsigned long int`	nombre entier (32 bits) non signé	de 0 à $2^{32}-1$	de 0 à 4 milliards
`float`	nombre à virgule flottante	$1{,}4\times10^{-45}$ à $3{,}4\text{x}10^{38}$	pas de limite, valeur approximative
`double`	nombre à virgule flottante	identique à float	

Ce que nous pouvons constater dans ce tableau c'est que le type `int` est signé par défaut, s'il est précédé par le mot clef `unsigned` il est alors non signé (il va alors de 0 à son double).

Nous pouvons aussi constater que certains types sont en doublons, ainsi :

`unsigned int`, `unsigned short int` et `word` sont identiques,

`unsigned char` est identique à `byte`,

`short int` est identique à `int`.

En réalité ceci n'est vrai que sur les *Arduino* à base d'*ATMega*. Sur l'*Arduino DUE* les `int` et `unsigned int` sont sur 32 bits. On touche là à un problème des langages C et C++ : les types n'ont pas de tailles fixes définies.

De plus c'est généralement très gênant pour ceux qui ont l'habitude de faire de la programmation sur différentes cibles : que ce soit sur PC, ou d'autres familles de microcontrôleurs, les tailles sont encore plus fluctuantes.

Fort heureusement des types de tailles fixes existent :

type	description	propriétés	A retenir (environ)
`int8_t`	nombre entier (8 bits) signé	de -2^7 à 2^7-1	-128 à +127
`uint8_t`	nombre entier (8 bits) non signé	de 0 à 2^8-1	0 à 255
`int16_t`	nombre entier (16 bits) signé	de -2^{15} à $2^{15}-1$	-32000 à +32000
`uint16_t`	nombre entier (16 bits) non signé	de 0 à $2^{16}-1$	0 à 65000
`int32_t`	nombre entier (32 bits) signé	de -2^{31} à $2^{31}-1$	-2 milliards à +2 milliards
`uint32_t`	nombre entier (32 bits) non signé	de 0 à $2^{32}-1$	de 0 à 4 milliards
`int64_t`	nombre entier (64 bits) signé	de -2^{63} à $2^{63}-1$	
`uint64_t`	nombre entier (64 bits) non signé	de 0 à $2^{64}-1$	

Vu la grande quantité de types pour les entiers, comment doit-on choisir ?

Si vous êtes adepte de l'adage « qui peut le plus peut le moins », oubliez-le, c'est la plus mauvaise méthode de choix à adopter.

Il suffit de choisir le type par rapport aux valeurs minimales et maximales à placer dans cette variable.

Par exemple si vous avez besoin de faire une boucle qui va de 0 à 30, il n'est pas nécessaire d'utiliser un `int`, un simple `byte` ou `uint8_t` suffira amplement.

Il faut bien comprendre que la majorité des cartes *Arduino* sont basées sur des microcontrôleurs *ATMega* qui sont 8bits, c'est-à-dire que tous les calculs sur 8 bits sont faits en 1 seul cycle d'horloge alors que tout calcul sur des types de tailles supérieures devra être fait logiciellement et donc prendra beaucoup plus de cycles et donc de temps, et augmentera l'occupation de la mémoire.

Il est donc très important d'adapter le type de variable au besoin que vous en avez.

De même pour les nombres réels, vous pouvez utiliser les types `float` ou `double` mais ceux-ci sont entièrement gérés logiciellement et consomment ainsi beaucoup de temps et de mémoire lors des calculs.

b. opérateurs sur les variables

Nous venons à l'instant de parler de calculs sur les variables, il est donc temps de voir les opérateurs qui permettent de faire ces calculs.

Les opérateurs suivants respectent la syntaxe suivante : valeur1 opérateur valeur2

valeur1 et valeur2 pouvant être des valeurs numériques comme des variables.

Opérateur	Description
`+`	addition
`-`	soustraction
`*`	multiplication
`/`	division
`%`	modulo (reste de la division entière)

D'autres opérateurs existent :

Opérateur	Description
`!`	inversion logique
`++`	incrémentation
`--`	décrémentation

L'opérateur `!` permet d'inverser la valeur logique, pour rappel : est faux ce qui est égal à 0, toutes les autres valeurs sont considérées comme vraies.

`Remarque` : le résultat de l'inversion n'est pas directement assigné à la variable.

Exemple :

```
variable = !variable;
```

L'incrémentation consiste à ajouter 1 :

```
variable++;
```

est équivalent à

```
variable = variable + 1;
```

correspond à

```
variable ← variable + 1
```

La décrémentation consiste à retirer 1 :

```
variable--;
```

est équivalent à

```
variable = variable - 1;
```

correspond à

```
variable ← variable - 1
```

c. déclaration des variables

Lors de nos premiers pas nous avions dit que les variables pouvaient être déclarées en tout début du programme en dehors des fonctions `loop()` et `setup()`.

Regardons maintenant les différentes alternatives qui s'offrent à nous et leurs conséquences.

L'endroit où l'on déclare une variable va définir sur sa portée, c'est-à-dire où dans le programme elle sera accessible. Il existe 4 types de portées : de blocs, de fonctions, de fichier et de programme.

Lorsque nous avons déclaré notre variable tout au début, en dehors des fonctions cela crée une variable à portée de fichier. c'est-à-dire que cette variable sera accessible dans toutes les fonctions de notre fichier *.ino* (croquis), aussi bien dans `setup()`, dans `loop()` que dans les autres fonctions que nous apprendrons par la suite à écrire. On appelle aussi une variable à portée de fichier une variable globale.

Si nous avions déclaré cette variable à l'intérieur de la fonction `setup()` elle n'aurait été accessible que dans cette fonction. C'est ce que l'on appelle la portée de fonction.

Nous avons vu qu'avec un `if` ou un `while` nous pouvions créer des blocs (avec les accolades), si nous avions déclaré cette variable à l'intérieur de ce bloc alors elle n'aurait existé qu'en son sein, elle serait inaccessible en dehors. Ce ce que l'on appelle une portée de bloc.

```
while (condition)
{
     int ma_var;
     …
}
```

Après l'accolade fermée, la variable n'existe plus et donc n'est plus accessible.

Comme nous l'avons vu la déclaration d'une variable se fait toujours ainsi :

```
type nom_variable;
```

Que le type soit `int`, `char`, `float`, etc.

Il est possible de déclarer sur une seule ligne plusieurs variables de même type, on procède ainsi :

```
type nom_variable1, nom_variable2, nom_variable3;
```

Il est aussi possible d'affecter une valeur à une variable dès sa déclaration :

```
type nom_variable1 = valeur, nom_variable2, nom_variable3 = valeur;
```

Il est important de ne pas oublier d'initialiser une variable avant d'utiliser son contenu ! Lorsqu'une variable est déclarée elle ne vaut pas forcément 0, elle contient ce qui était dans l'espace mémoire avant qu'elle l'occupe.

3.3. Les tableaux

Nous avons vu les différents types de bases des variables des langages C/C++. Bien sûr ces langages offrent la possibilité de créer des tableaux, c'est-à-dire plusieurs variables d'un même type qui sont consécutives en mémoire. Ainsi au lieu d'avoir N variables avec N noms, nous avons un seul nom et on accède aux différentes valeurs du tableau par un indice.

Les tableaux sont possibles avec tous les types de variables. Regardons comment déclarer un tableau :

```
type nom_tableau[NOMBRE_DE_VALEURS];
```

Bien que l'on puisse utiliser une variable pour définir `NOMBRE_DE_VALEURS`, il est préférable d'utiliser une constante entière.

Si nous voulons créer un tableau de 10 éléments de type `int` :

```
int tableau_dentiers[10];
```

Pour accéder au Nième+1 élément du tableau :

```
nom_tableau[N]
```

Par exemple pour le 4ième élément :

```
nom_tableau[3]
```

Ce décalage est expliqué par l'indice du premier élément qui est 0 :

```
nom_tableau[0]  // premier élément du tableau
```

Évidemment, contrairement à la déclaration du tableau on peut spécifier une variable pour accéder à l'élément dont l'indice est le contenu de la variable.

Exemple :

```
byte indice = 12;
int tableau_dentiers[14];
tableau_dentiers[indice] = 42;
```

Un tableau, comme une variable, vaut ce qui était en mémoire avant qu'il y soit placé, il n'est pas automatiquement initialisé à 0 (ou autre). Il est donc impératif de vérifier qu'on ne lit pas un élément du tableau qu'on aurait pas, au préalable, affecté de valeur.

Il y a un autre risque bien plus important : il ne faut jamais accéder à un élément dont l'indice est supérieur ou égal à la taille du tableau.

3.4. Les chaînes de caractères

Il existe un type de tableau pouvant avoir un traitement particulier en C/C++. C'est le tableau de type `char`. En effet on peut l'utiliser comme un tableau d'octets mais il est très souvent utilisé comme « chaîne de caractères ». Une chaîne de caractères en C/C++ n'est qu'un tableau d'entiers de type `char` dont le dernier caractère est le caractère nul `'\0'` de valeur décimale 0.

Il est primordial de bien comprendre qu'une chaîne de caractères est toujours terminée par le caractère nul `'\0'`, ou encore appelé caractère de fin de chaîne. C'est le seul moyen de connaître la taille d'une chaîne et donc de pouvoir la traiter.

Une chaîne de caractère se déclare comme un tableau normal :

```
char maChaine[TAILLE_CHAINE+1];
```

Il ne faut surtout pas oublier d'ajouter 1 à la taille que l'on veut, sinon il n'y aura pas de place pour le caractère nul.

Il est possible d'initialiser directement la chaîne lors de la déclaration. Voici plusieurs exemples :

```
char maChaine[10] = "ma chaine";
```

On crée un tableau de 10 caractères qui va être rempli par les 9 caractères de « ma chaine » plus le caractère nul.

```
char maChaine[20] = "ma chaine";
```

On crée un tableau de 20 caractères qui va être rempli par les 9 caractères de « ma chaine » plus le caractère nul.

```
char maChaine[] = "ma chaine";
```

La taille du tableau est automatiquement déterminée par la taille de la chaîne, donc le tableau fera 10 octets (9 caractères pour la chaîne + le caractère nul).

ATTENTION : il est totalement interdit de faire directement une affectation (en utilisant l'opérateur '=') d'une chaîne en dehors de la déclaration.

Pour manipuler des ch'aînes de caractères il existe une panoplie de fonctions que nous allons voir maintenant. Il ne faut pas oublier qu'une chaîne n'est qu'un tableau et qu'il est possible d'y accéder, de modifier son contenu comme pour n'importe quel tableau.

Fonction	Comportement
`strlen (chaine);`	Renvoie la taille de chaîne.
`strcpy (destination, source);`	Recopie la source dans la destination
`strcat (destination, source);`	Ajoute la source à la fin de la destination
`strcmp (chaine1, chaine2);`	Renvoie 0 si les deux chaînes sont identiques
`strcasecmp (chaine1, chaine2);`	Renvoie 0 si les deux chaînes sont identiques sans distinction entre les minuscules et les majuscules.

Nous avons vu qu'avec la fonction `Serial.print()` nous pouvions envoyer des chaînes de caractères sur la liaison série. Cependant, nous constatons rapidement que cette fonction n'est pas très pratique si nous voulons envoyer plusieurs variables, ainsi que du texte. Pour remédier à cela nous pouvons créer une chaîne de caractères qui va contenir tout le texte à envoyer.

La fonction à utiliser pour réaliser cette tâche est la fonction `sprintf()`. Cette fonction est *a priori* difficile à appréhender mais se révèle très efficace.

```
sprintf (chaine, chaineDeFormat, variable1, variable2, ..);
```

où

`chaine` est le tableau qui va être remplie par la fonction `sprintf()`,

`chaineDeFormat` est la chaîne de caractères qui indique le formatage,

`variable1`, `variable2`, ... sont les variables qui vont être converties et placées dans chaîne.

Regardons un exemple avant d'expliquer le fonctionnement :

```
char chaine[40];
int var = 42;
sprintf (chaine, "réponse = %d", var);
Serial.println (chaine);
```

Si on regarde le résultat dans le terminal série nous aurons :

```
réponse = 42
```

La fonction `sprintf()` va recopier la chaîne de formatage qui est en deuxième paramètre jusqu'à tomber sur `%d`, à ce moment là elle remplace `%d` par le contenu de la variable qui est en 3ème paramètre. Il est possible de mettre plusieurs variables, il faudra alors autant de `%d` qu'il y a de variables.

Exemple :

```
char chaine[40];
int var1 = 4, var2 = 2, var3 = 42;
sprintf (chaine, "variables =%d %d %d", var1, var2, var3);
Serial.println (chaine);
```

Le résultat est

```
variables = 4 2 42
```

Dans ces exemples nous n'avons converti que des entiers de type `int`, mais nous pouvons utiliser aussi des chaînes. Il suffit d'adapter la séquence de formatage selon le type de la variable :

Séquence	Type
`%d`	entier signé
`%u`	entier non signé
`%c`	caractère ASCII
`%s`	chaîne de caractères

Exemple :

```
char chaine[30];
char temp[] = "22.3";
sprintf (chaine, "température %s\r\n", temp);
Serial.print (chaine);
```

En réalité la fonction `sprintf()` est bien plus riche et bien plus compliquée que cela, car elle permet de préfixer un nombre entier d'espace ou de 0.

Si nous voulons un entier sur 4 caractères préfixés par des espaces la séquence de formatage sera : `%4d`

Si nous voulons un entier sur 4 caractères préfixés par des 0 la séquence de formatage sera : `%04d`

Exemple :

```
char chaine[7];
sprintf (chaine, "(%4d)", 42);
Serial.println (chaine);
sprintf (chaine, "(%04d)", 42);
Serial.println (chaine);
```

Nous aurons dans le terminal série :

```
(  42)
(0042)
```

En C standard la fonction `sprintf()` permet de convertir des nombres à virgule flottante (type *float*), malheureusement ce n'est pas le cas de l'implémentation de cette fonction dans *Arduino*. Nous devons alors nous rabattre sur une fonction dédiée à cette conversion :

```
dtostrf(valeur, taille, nb_decimales, chaine)
```

où

`valeur` est le nombre à virgule flottante,

`taille` est la longueur de la chaine convertie,

`nb_decimales` est le nombre de décimales désirés,

`chaine` est le tableau qui va être remplie par la fonction.

Il est un peu délicat de saisir le paramètre `taille` : c'est la taille de la chaîne une fois convertie, si le nombre de caractères convertis est plus petit que la valeur de `taille` alors la chaîne sera préfixée d'autant d'espaces que nécessaire pour atteindre cette valeur.

Exemple :

```
      char buffer[10];
      dtostrf(13.666, 10, 2, buffer);
      Serial.println(buffer);
      dtostrf(13.666, 0, 2, buffer);
      Serial.println(buffer);
      dtostrf(13.666, 0, 3, buffer);
      Serial.println(buffer);
```

Nous voyons alors sur le terminal :

```
       13.67
13.67
13.666
```

Et si nous utilisions cela pour envoyer des données à *Labview*. Par exemple nous pourrions lire l'entrée analogique `A2` et envoyer ces données sur la liaison série. Un petit peu de *Labview* permettrait de recevoir ces données et de nous faire un graphique.

Voici le *sketch Arduino* :

```
void setup ()
{
    Serial.begin (19200);
}

void loop ()
{
     char buffer[5];
     int val = analogRead (A2);
     sprintf (buffer, "%04d", val);
     Serial.print (buffer);
     delay (20);
}
```

Voici le diagramme *Labview* :

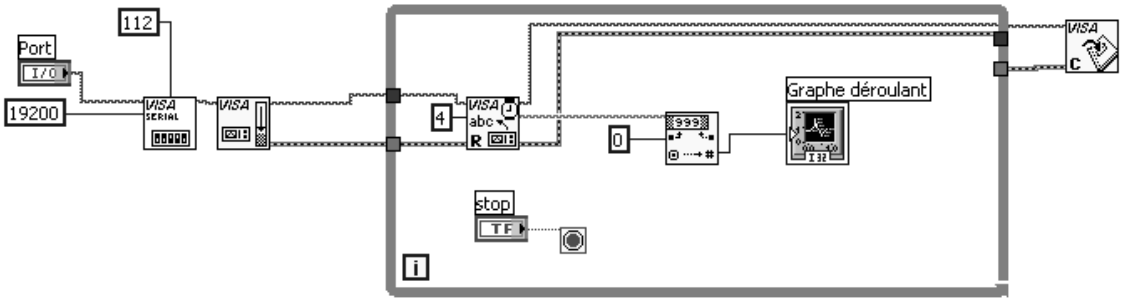


Figure 3: Diagramme Labview

Et voici le résultat :

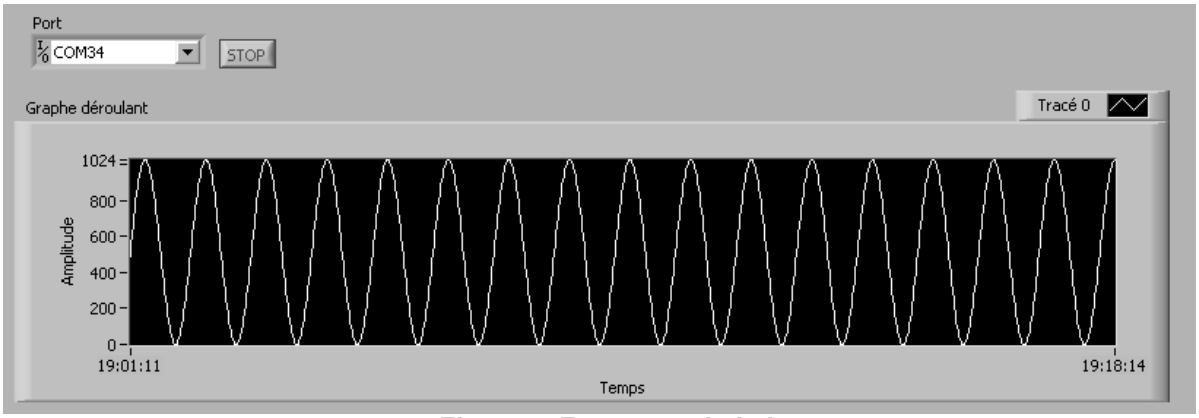


Figure 4: Face avant Labview

Heureusement, il existe une alternative intégrée à *Arduino* pour afficher sous forme de graphique les données renvoyées sur la liaison série. Pour cela il faut cliquer sur *Traceur série* dans le menu *Outils*.

3.5. le if dans toute sa splendeur

La forme que nous avons vu du `if` est la forme la plus simple. Il est possible, d'une part de tester plusieurs conditions en même temps, et d'autre part de chaîner les `if` de manière plus élaborée qu'avec un simple `else`.

Le `if` simple :

```
if (condition)
{
    instructions si la condition est vraie;
}
else
{
    instructions si la condition est fausse;
}
```

Nous avons vu les opérateurs permettant de faire des comparaisons. Il est possible de chaîner les conditions. L'expression des conditions étant les mêmes pour `if` et pour les boucles, ceci s'applique aussi aux conditions des boucles que nous venons de voir.

Opérateur	Condition exprimée
CONDITION1 && CONDITION2	Si CONDITION1 ET CONDITION2 sont vraies
CONDITION1 || CONDITION2	Si CONDITION1 OU CONDITION2 sont vraies

Exemple :

```
if (valeur1 > 0 && valeur2 < 100)
```

Il est possible de spécifier la priorité d'une opérations entre 2 conditions en utilisant les parenthèses.

Exemple :

```
if ((valeur1 > 0 && valeur1 < 100 && valeur2 == 0) || (valeur1 > 50 &&
valeur2 == 1))
```

La condition est vraie dans 2 cas :

si `valeur1` est positif, inférieur à 100 et `valeur2` vaut 0

ou

si `valeur1` est supérieur à 50 et `valeur2` vaut 1

Exercice :

Écrivez un programme qui convertit en minuscule (ou en majuscule) une chaîne de caractères pouvant contenir n'importe quel caractère imprimable de la table *ASCII*. Il faudra donc traiter uniquement les caractères allant de A à Z (ou a à z) et ne pas modifier les autres. (Indice : le caractère a s'écrit en C `'a'`).

Le chaînage des `if` en C est un peu particulier, sa syntaxe est la suivante :

```
if (condition1)
{
  // si condition1 est vraie
}
else if (condition2)
{
  // si condition1 est fausse et condition2 est vraie
}
else if (condition3)
{
  // si condition1 et condition2 est fausse et condition3 est vraie
}
else
{
  // si toutes les précédentes conditions sont fausses
}
```

Il est possible de mettre autant de `else if` que nécessaire, mais attention : un tel code est rapidement illisible.

3.6. Les boucles

a. Boucle while

Nous avons vu brièvement la boucle `while`, mais il existe d'autres types de boucles en langage C/C++.

Toutes les boucles en C/C++ sont exécutées selon la valeur booléenne de la condition exprimée.

Revenons sur la boucle `while`. Sa syntaxe est la suivante :

```
while (condition)
{
  instructions;
}
```

Il faut bien comprendre comment le programme est exécuté. Tout d'abord la condition est évaluée, si elle est vraie, alors on rentre dans la boucle pour exécuter une itération puis la condition est de nouveau évaluée, si elle est toujours vraie une nouvelle itération est exécutée. Cela se répète jusqu'à ce que la condition devienne fausse. Souvent on utilise les boucles pour incrémenter (= ajouter 1) à une variable et exécuter des instructions pour chacune de ses valeurs.

Nous avons alors :

```
int var = 0;
while (var < 10)
{
  /* instructions; */
  var++;
}
```

Rappel : `var++` signifie `var = var + 1;`

b. Boucle for

La boucle `for` est un condensé de la boucle `while`, exemple :

```
int var;
for (var = 0; var < 10; var++)//Pour var de 0 à inférieur à 10 par pas de 1
{
  /* instructions; */
}
```

Ce qui correspond à :

```
for (initialisation; condition; opération de fin de boucle)
```

c. Boucle do while

La dernière boucle est le `do while`.

La différence entre cette boucle et le `while`, est qu'elle exécute d'abord une première itération avant de tester si la condition est vraie ou fausse. Sa syntaxe est la suivante :

```
do
{
  instructions;
} while (condition);
```

d. Opérations sur les boucles

On peut avoir besoin de sortir prématurément d'une boucle (avant que la condition ne soit fausse), on utilise pour cela l'instruction `break`.

Plus rarement, on peut avoir besoin de revenir au début de la boucle, pour cela on utilise le mot-clé `continue`. Ce mot-clé va permettre de revenir aussitôt à la condition de la boucle. Cependant, attention car dans le cas d'une boucle `for`, ce qui était défini par **`opération de fin de boucle`** va être exécuté avant une nouvelle évaluation de la condition.

3.7. Fonctions

Nous avons écrit du code permettant de faire différentes choses depuis le début de cette formation. Mais tout ce code a été écrit dans la fonction `loop()`. Il est grand temps de voir comment architecturer son code en créant des fonctions qui vont réaliser certaines tâches bien précises.

Les fonctions vont permettre d'une part de ne jamais avoir à copier-coller des blocs de code, et d'autres part à créer une organisation par blocs cohérents, comme des sortes de tâches bien distinctes.

Pour l'instant nous avons vu les fonctions `loop()` et `setup()`, et nous avons utilisé des fonctions fournies par *Arduino*. Regardons comment créer une autre fonction.

```
type nom_fonction (type nomPremierParametre, type nomDeuxiemeParametre)
{
  instructions;
}
```

On peut mettre autant de paramètres que l'on veut. On peut aussi ne pas en mettre, on écrit alors `void`, ou laisser vide.

Le type qui précède le nom de la fonction est le type de retour de la fonction. Si la fonction ne retourne rien le type sera `void`.

Les paramètres qui sont déclarés sont des variables qui pourront être utilisées directement dans le corps de la fonction.

Exemple :

```
void setup()
{
    Serial.begin (9600);
}

void loop()
{
    envoiBonjour();
```

```
    delay (200);
}

void envoiBonjour ()
{
    Serial.println ("Bonjour le monde");
}
```

Exemple :

On crée une fonction qui renvoie la valeur absolue du réel qu'on lui passe en paramètre.

```
float valeurAbsolue (float nombre)
{
    if (nombre > 0)
        return (nombre);
    else
        return (nombre * -1.0);
}

void setup ()
{
    Serial.begin (9600);
}

void loop ()
{
    float nombre;
    nombre = -12.0;
    nombre = valeurAbsolue (nombre);
    Serial.println (nombre);
    while (1);
}
```

Il est important de comprendre que la variable `nombre` dans la fonction `valeurAbsolue()` est une nouvelle variable qui ne partage pas l'espace mémoire de la variable `nombre` qui est passée en paramètre. C'est pour cela qu'on est obligé de faire un `return` qui renvoie la valeur que l'on affecte ensuite dans la fonction `loop()` à sa variable `nombre`.

Pour s'en convaincre il suffit d'essayer le code suivant :

```
void setup ()
{
    Serial.begin (9600);
}

void inc (int var)
{
    var++;
}

void loop ()
{
    int valeur = 5;
    inc (valeur);
    Serial.println (valeur);
    while (1);
}
```

A l'exécution on aura dans le terminal série : 5

Faire `inc(valeur)` revient à faire `inc(5)`, or si la valeur était modifiée dans la fonction nous aurions alors : 5 = 5 + 1 ce qui n'est évidemment aucun sens.

Cependant, il y a une exception, ce sont les tableaux. Si vous passez en paramètre un tableau il n'est pas recopié, le tableau qui sera utilisé dans la fonction aura le même espace mémoire. Donc toute modification sur le tableau passé en paramètre sera aussi dans le tableau de la fonction appelante.

Même s'il est possible d'écrire `type nom_tableau[TAILLE]` ou `type nom_tableau[]` dans la déclaration de la fonction, on préférera utiliser la notation suivante : `type *nom_tableau`.

a. Utilisation des références

Pour pallier ce problème nous pourrions créer une variable globale, mais cela nuirait à l'architecture du programme. En C nous n'aurions alors d'autres choix que de faire des pointeurs, cependant les pointeurs sont très difficiles à comprendre lorsqu'on les découvre. Heureusement en C++ il existe un mécanisme beaucoup plus simple qui permet de résoudre ce problème : les références.

La référence va créer une sorte d'*alias* de la variable, la syntaxe de sa déclaration est la suivante :

```
type &nom_variable
```

Évidemment le type de la référence doit correspondre exactement à celui de la variable. Ensuite la référence s'utilise comme n'importe quelle autre variable.

Exemple précédent avec une référence :

```
void setup ()
{
    Serial.begin (9600);
}
void inc (int &var)
{
    var++;
}
void loop ()
{
    int valeur = 5;

    inc (valeur);
    Serial.println (valeur);

    while (1);
}
```

A l'exécution nous aurons bien la valeur 6 dans le terminal série.

Remarque : si la variable `valeur` était de type `char` il y aurait eu une erreur à la compilation.

4. Arduino

Dans le chapitre précédent nous avons surtout vu les éléments des langages C et C++ qui forment les fondations d'*Arduino*. Maintenant que nous disposons des bases nécessaires nous pouvons nous attarder sur les fonctions propres d'*Arduino*.

Il existe davantage de fonctions que celles indiquées dans cette formation, puisqu'il n'était pas possible d'être exhaustif. Je vous invite donc à vous reporter à la documentation d'*Arduino* disponible à l'adresse suivante : http://*Arduino*.cc/en/Reference/HomePage

4.1. Tone

La première fonction que nous allons voir permet de générer un signal carré entre 31Hz et 8MHz. Elle est généralement utilisée avec un petit haut parleur piezzo pour générer des sons, comme celui présent sur le *shield* de la formation (sur la *pin* A2).

Sa syntaxe est la suivante :

```
tone (pin, frequence);
```

ou

```
tone (pin, frequence, duree);
```

`pin` correspond à la *pin* sur laquelle nous voulons générer le signal,

`frequence` est la fréquence en Hz (`unsigned int`)

`duree` est un paramètre optionnel qui permet de définir le temps que doit durer le signal.

`Remarques` : il n'est possible de générer qu'un seul signal à la fois. De plus `tone()` utilise le *timer2* et entre donc en conflit avec les autres fonctions et bibliothèques qui l'utilisent tels que les *PWM* sur 3 et 11.

Pour arrêter le signal nous devons utiliser la fonction suivante :

```
noTone (pin);
```

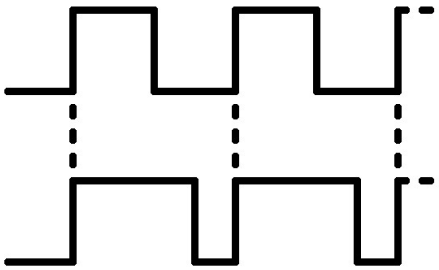
`Exercice :`

Faites varier la fréquence du signal envoyé sur le buzzer (*pin* A2) selon la valeur du potentiomètre, prenez par exemple 100Hz comme minimum et 3KHz comme maximum. Utilisez le bouton sur la *pin* 2 pour arrêter la génération du signal.

4.2. PWM

Nous avons souvent besoin de générer des *PWM* ou MLI en français. *Arduino* possède des fonctions permettant d'en générer sur les *pins* suivantes : 3, 5, 6, 9, 10, 11.

Rappel : une PWM est un signal de fréquence fixe dont on fait varier le rapport cyclique (*duty cycle*) c'est-à-dire le rapport entre le temps à l'état haut et le temps à l'état bas.



Ici le signal a un rapport cyclique de 50%

Ici le signal a un rapport cyclique d'environ 75 %

Figure 5: Rapport cyclique

Nous voyons dans la figure ci-dessus que la fréquence du signal est conservée : le temps entre deux fronts montants, passages de l'état bas à l'état haut, est constant. Ce délai se nomme la période, la fréquence étant l'inverse de la période.

Il faut comprendre que les *PWM* ne sont pas générées logiciellement mais utilisent les *timers hardware* du microcontrôleur. Chaque *timer* du microcontrôleur dispose de deux canaux permettant de générer ainsi deux signaux. Ils sont répartis ainsi :

PWMs	**Timer**
3, 11	timer2
5, 6	timer0
9, 10	timer1

Plusieurs fonctions ou bibliothèques utilisent ces mêmes *timers* pour leur fonctionnement (généralement le *timer2*) et des conflits peuvent donc exister.

Les fonctions `millis()` et `delay()` utilisent aussi le *timer0* ce qui peut entraîner des comportements inexacts des *PWMs* sur les *pins* 5 et 6. Par exemple le rapport cyclique peut être légèrement plus élevé que celui demandé, surtout lorsque le rapport cyclique demandé est faible.

Il faut aussi considérer qu'il n'est pas prévu dans *Arduino* de pouvoir modifier la fréquence de ces *PWMs* qui sont fixées à environ 490Hz pour les *pins* 3, 9, 10 et 11 et de 980Hz pour les *pins* 5 et 6.

Cependant nous verrons un peu plus loin que nous pouvons changer ces fréquences en modifiant les registres du microcontrôleur.

La fonction à utiliser pour générer les *PWMs* est la suivante :

```
analogWrite (pin, rapport_cyclique);
```

où

`pin` est la *pin* à utiliser,

`rapport_cyclique` est le rapport cyclique entre 0 et 255, 0 étant toujours éteint et 255 étant toujours actif.

Pour rappel : le rapport cyclique est le rapport du temps à l'état haut du signal sur la période du signal.

Exercice :

Faites varier l'intensité lumineuse de la LED qui est sur la *pin* 3 selon la valeur du potentiomètre.

4.3. Le temps

Nous avons vu au tout début les fonctions `delay()` et `delayMicroseconds()` mais d'autres fonctions très utiles existent.

Pour rappel `delay()` et `delayMicroseconds()` s'utilisent ainsi :

```
delay (duree);
```

où `duree` est un temps en millisecondes (`unsigned long int`).

```
delayMicroseconds (duree);
```

où `duree` est un temps et microsecondes (`unsigned int`).

Remarque : au delà d'une durée de 16383µs la précision n'est plus garantie, il vaut mieux dans ce cas utiliser la fonction `delay()`.

Nous avons aussi la fonction `millis()` qui retourne le temps écoulé (en `unsigned long int`), en millisecondes, depuis le démarrage du programme.

```
unsigned long int temps;
temps = millis();
```

Remarque : au bout d'environ 50 jours le compteur débordera et reprendra à partir de 0.

Pour plus de précision nous pouvons utiliser la fonction `micros()` qui retourne le nombre (en `unsigned long int`) de microsecondes écoulées depuis le démarrage du programme.

```
unsigned long int temps;
temps = micros();
```

Remarque : au bout d'environ 70 minutes le compteur débordera et reprendra à partir de 0, de plus la résolution est de 4µs.

Exercice : écrire un programme qui mesure le temps d'appui sur un des boutons du *shield*, puis renvoyez le sur la sortie série.

Vous avez du remarquer dans l'exercice que ces fonctions d'*Arduino* sont bien utiles et vous facilitent la tâche. Cependant ce que vous ignorez certainement c'est que ce que vous venez d'écrire existe déjà dans *Arduino* et c'est la fonction `pulseIn()`.

Cette fonction permet de mesurer le temps d'une impulsion, qu'elle soit à l'état bas ou à l'état haut et s'utilise ainsi :

```
unsigned lont int temps;
temps = pulseIn(pin, value, timeout);
```

où

`pin` est le nom de la *pin* sur laquelle doit se produire l'impulsion,

`value` doit être soit `HIGH` pour mesurer le temps de l'état haut, soit `LOW` pour mesurer le temps à l'état bas,

`timeout` est un paramètre optionnel qui définit le délai d'expiration en microsecondes (`unsigned long int`) de l'attente de l'impulsion, si `timeout` n'est pas indiqué il est par défaut d'une seconde.

La valeur retournée est le temps en microsecondes qu'a duré l'impulsion, ou 0 si le délai d'attente a expiré.

Remarque : la fonction est prévue pour fonctionner correctement sur des impulsions entre 10µs et 3 minutes.

Imaginons que vous voulons de nouveau mesurer le temps d'appui sur le bouton, donc l'état bas.

Nous allons faire ceci :

```
unsigned long int temps;
temps = pulseIn (3, LOW, 5000);
```

Puisque nous voulons mesurer le temps à l'état bas (`LOW` dans le code), la fonction va attendre le front descendant, donc le passage de l'état haut à l'état bas, démarrer son compteur, attendre le front montant, arrêter son compteur pour finalement nous retourner le résultat. Si nous n'avons pas appuyé dans les 5s (`5000` dans le code), la fonction retournera 0 pour nous indiquer que le délai d'attente a expiré.

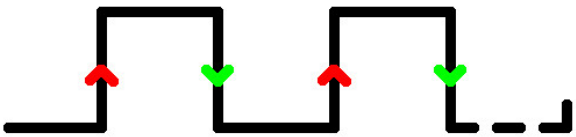


Figure 6: Fronts montants et descendants

Le signal est représenté en noir

La flèche **rouge** indique le front montant

La flèche **verte** indique le front descendant

Les deux figures suivantes montrent le fonctionnement de `pulseIn()`. Le temps mesuré est à chaque fois entre les deux flèches.

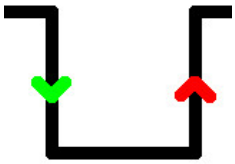


Figure 7: pulseIn LOW

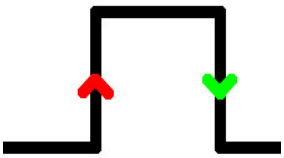


Figure 8: pulseIn HIGH

4.4. Les interruptions

Pour le moment chacun des programmes que vous avez écrit se sont déroulées linéairement, dans l'ordre où vous aviez écrit les instructions. Parfois nous avons besoin de casser le déroulement en cours du programme pour exécuter une tâche lorsqu'un événement survient. Il n'est pas possible sur un microcontrôleur d'effectuer des tâches en parallèle comme sur un PC. Au lieu de cela nous avons la possibilité d'utiliser les interruptions.

En réalité vous avez déjà utilisé les interruptions sans le savoir. La réception des données série, ou encore l'utilisation de `tone()` utilisent des interruptions en arrière plan.

Pour le moment lorsque vous vouliez savoir si un bouton était appuyé ou non vous deviez utiliser périodiquement la fonction `digitalRead()`. Cette fois-ci nous allons utiliser une interruption.

Arduino dispose de deux entrées qui sont interruptibles, c'est-à-dire qu'en cas d'événement, que l'on peut configurer, sur ces entrées cela va entraîner une interruption de votre programme pour exécuter une fonction particulière. Une fois cette fonction terminée le cours du programme reprendra là où il s'était arrêté.

La fonction `attachInterrupt()` se charge de configurer l'interruption :

```
attachInterrupt (interruption, fonction, mode);
```

où

`interruption` vaut 0 pour la *pin* 2 et 1 pour la *pin* 3,

`fonction` est le nom de la fonction qui sera exécutée lors de l'événement,

`mode` est le mode de déclenchement.

Mode	Description
LOW	déclenchement tant que l'entrée est à l'état bas
RISING	déclenchement sur le front montant (passage de l'état bas à l'état haut)
FALLING	déclenchement sur le front descendant (passage de l'état haut à l'état bas)
CHANGE	déclenchement sur front montant et front descendant.

Le nom de la fonction passée en paramètre doit être de type `void` (donc ne rien retourner) et n'avoir aucun paramètre.

Sur le *shield* nous avons placé un bouton sur la *pin* 2 qui est interruptible. Regardons un exemple pour l'utiliser :

```
volatile boolean etatLed = false;
const byte pinLed = 8;

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

void appuiBouton (void)
{
    digitalWrite (pinLed, etatLed);
    etatLed = !etatLed;
}
void loop ()
{
    attachInterrupt (0, appuiBouton, FALLING);
}
```

L'interruption est configurée pour se produire quand on appuie sur le bouton, puisque l'état vaut 0 (`LOW`) quand le bouton est enfoncé nous utilisons le mode `FALLING`.

Notez l'utilisation de l'attribut `volatile` pour la variable utilisée dans la fonction d'interruption.

Cela est due au fait que le compilateur cherche toujours à optimiser le code et à supprimer tout ce qui ne sert à rien, or pour lui l'interruption n'est jamais appelée, car il n'y a pas d'appel direct à cette fonction dans notre programme. Il va donc avoir tendance à supprimer les variables qui ne sont utilisées qu'à l'intérieur. Si la variable est aussi utilisée dans le programme principal, alors il peut avoir tendance à la placer dans un registre la rendant totalement inaccessible à la fonction d'interruption.

Si ce qui est dit précédemment n'est pas très clair, pensez juste à placer l'attribut `volatile` devant toutes les variables que vous utilisez dans une fonction d'interruption.

5. Les bibliothèques

Commençons par définir ce qu'est une bibliothèque. Une bibliothèque est un ensemble de fonctions, suffisamment généralistes, pouvant être utilisées sans modifications dans plusieurs projets. Le but étant d'utiliser les mêmes morceaux de code d'un projet à l'autre sans recourir au copier-coller.

Arduino dispose en natif de plusieurs bibliothèques : *EEPROM* pour la gestion de l'*eeprom* interne au microcontrôleur, *Ethernet* pour le *shield* du même nom, SPI et *Wire* (pour l'*I2C*) sont aussi des bibliothèques, *Stepper* pour les moteurs pas à pas, *SD* pour la gestion de la lecture et de l'écriture sur une carte *SD* (avec gestion de la *FAT*), *SoftwareSerial* pour l'utilisation d'une liaison série sur d'autres *pins* que 0 et 1, et quelques autres encore.

On constate que dès le départ *Arduino* répond à beaucoup d'attentes, mais il faut aussi prendre en compte la communauté : il existe des centaines de bibliothèques pour *Arduino* couvrant tous les domaines. Un grand nombre de capteurs, d'écrans, et autres ont des bibliothèques dédiées.

Arduino a une gestion des bibliothèques qui diffère des autres plateformes. D'habitude une bibliothèque est sous la forme d'un fichier binaire compilé qui sera lié au programme lors de l'édition des liens. Il n'en est rien sous *Arduino* : une bibliothèque est sous forme de code source. Ce qui est une autre preuve de la volonté d'ouverture d'*Arduino*. Vous pouvez lire le code d'une bibliothèque afin de savoir comment elle fonctionne, et même la modifier en cas de besoin.

5.1. Installer une bibliothèque tierce

Utilisons l'écran *LCD* présent sur le *shield* de la formation via la bibliothèque déjà toute faite. Elle est disponible à l'adresse suivante : *https://github.com/Seeed-Studio/Grove_LCD_RGB_Backlight/archive/master.zip*

Il y a trois façons d'installer une bibliothèque :

- **Manuellement**

Vous pouvez installer « à la main » la bibliothèque, c'était la seule façon de faire pendant longtemps.

Décompressez l'archive et vous obtenez un répertoire nommé « Grove_LCD_RGB_Backlight-master ».

Déplacez ensuite ce répertoire dans le répertoire « *librairies* » de votre *sketchbook* (répertoire dans lequel sont stockés vos programmes *Arduino*).

Sous Windows il est par défaut dans *Mes Documents\Arduino*.

Sous Linux il est par défaut dans *~/Arduino*.

- **Semi automatiquement**

Dans l'IDE *Arduino* cliquez dans le menu :

« Croquis » → « Inclure une bibliothèque » → « Ajoutez la bibliothèque .ZIP »

Choisissez le fichier de la bibliothèque que vous venez de télécharger et validez par « OK ».

- **Automatiquement**

Dans l'IDE *Arduino* cliquez dans le menu :

« Croquis » → « Inclure une bibliothèque » → « Gérer les bibliothèques »

Entrez le nom de la bibliothèque dans le champ de recherche, sélectionnez le résultat souhaité et validez par « OK ».

Vous devez ensuite relancer l'IDE *Arduino*, car c'est au démarrage que le programme va analyser le contenu du répertoire « *librairies* » afin de déterminer l'ensemble des bibliothèques disponibles.

Remarque : les bibliothèques sont, en majorité, fournies avec des exemples d'utilisation. Ces exemples sont accessibles dans le menu :

« Fichier » → « exemples » → « nom de la bibliothèque » → « nom de l'exemple »

Ensuite allez dans le menu « Croquis » puis « importer bibliothèque » et sélectionnez « Grove_LCD_RGB_Backlight ». Vous devriez voir apparaître la ligne suivante dans votre code :

```
#include <rgb_lcd.h>
```

Quelques précisions s'imposent. Le fichier « rgb_lcd.h » n'est pas une bibliothèque, c'est un fichier d'en-tête qui contient la liste des fonctions utilisables. C'est une sorte de catalogue référençant tout ce qui est disponible dans la bibliothèque. La bibliothèque est l'ensemble des fichiers « .cpp » contenant les fonctions avec les instructions et du ou des fichiers « .h ».

Que se serait-il passé si vous aviez écrit directement la ligne `#include <rgb_lcd.h>` au lieu d'aller dans le menu ? L'utilisation de la bibliothèque fonctionnerait-elle quand même ?

La réponse est oui. En réalité aller dans le menu pour ajouter la bibliothèque n'est pas nécessaire : avant la compilation du *sketch Arduino* va analyser toutes les directives `#include` et comparer les fichiers « .h » avec ceux des bibliothèques, s'ils correspondent *Arduino* va compiler la bibliothèque associée.

Contrairement à toutes les autres plateformes qui nécessite des options de compilation ou d'éditions de liens, ici tout est fait automatiquement. C'est très pratique pour ceux qui découvrent *Arduino* mais une terrible source de problème en cas de soucis car le processus est assez opaque.

Il y a donc une règle très importante à respecter : ne laissez jamais plusieurs versions, ou variantes, d'une même bibliothèque dans le répertoire « *librairies* ». Car *Arduino* se base sur le nom des fichiers d'en-têtes pour identifier la bibliothèque et si plusieurs bibliothèques ont le même nom de fichier d'en-tête alors il en prend une « au hasard ».

Remarque : attention car ce mécanisme de compilation des bibliothèques n'est vrai que pour le *sketch*. *Arduino* ne fait pas la même chose pour le code des bibliothèques. Ainsi si votre bibliothèque, ou une bibliothèque tierce, utilise des fonctions d'une deuxième bibliothèque vous devez impérativement mettre la directive `#include` concernant la seconde bibliothèque dans votre *sketch* même si vous ne l'utilisez pas directement.

Par exemple pour la bibliothèque « rgb_lcd » qui utilise « Wire » vous devez impérativement avoir, dans cet ordre, dans votre sketch les lignes suivantes :

```
#include <Wire.h>
#include <rgb_lcd.h>
```

5.2. Utilisation de la bibliothèque tierce

Il est temps d'utiliser cette bibliothèque. En réalité son utilisation se fait de la même façon que les bibliothèques natives.

a. bibliothèque LCD I2C Grove

Nous devons commencer par indiquer que nous voulons utiliser la bibliothèque donc nous pouvons aller dans le menu « importer bibliothèque » ou nous pouvons directement mettre

```
#include <Wire.h>
#include <rgb_lcd.h>
```

Ensuite il faut déclarer la « variable » de l'écran :

```
rgb_lcd lcd;
```

A l'instar de `Serial` nous devons ensuite utiliser la fonction `begin()` afin d'initialiser l'écran :

```
lcd.begin(16, 2);
```

Le premier paramètre est le nombre de caractères par ligne, le second paramètre est le nombre de lignes de l'écran.

Pour écrire un message sur l'écran nous devons utiliser la fonction `lcd.print()` ou `lcd.println()`. Ces fonctions s'utilisent très exactement comme les fonctions `Serial.print()` et `Serial.println()`.

Ainsi nous pouvons faire :

```
lcd.println ("Hello world !");
lcd.print (42);
```

Nous pouvons changer la position du curseur via la fonction :

```
lcd.setCursor(colonne, ligne);
```

Les fonctions `lcd.clear()` et `lcd.home()` permettent respectivement d'effacer l'écran et de repositionner le curseur au début de l'écran.

Les fonctions `lcd.scrollDisplayLeft()` et `lcd.scrollDisplayRight()` permettent respectivement de faire défiler le texte vers la gauche et vers la droite.

Les fonctions `lcd.blink()` et `lcd.noBlink()` permettent respectivement de faire clignoter le curseur et de le rendre persistant.

Les fonctions `lcd.cursor()` et `lcd.noCursor()` permettent respectivement d'afficher et de cacher le curseur.

L'écran choisi pour le *shield* a la particularité de disposer d'un rétroéclairage RGB. Il y a donc des fonctions associées à cette fonctionnalité.

Pour changer la couleur :

```
lcd.setRGB(rouge, vert, bleu);
```

où `rouge`, `vert` et `bleu` sont des entiers non signés sur 8 bits (`byte` / `unsigned char`) représentants la teinte à appliquer.

`Exercice` : mesurez la température via le capteur LM35DZ sur l'entrée analogique A0 et affichez la à intervalle régulier sur l'écran avec un petit message défilant.

Indication : le capteur LM35DZ fourni une tension linéaire de 10,0mV/°C, cette tension est amplifiée d'un gain de 3,35. On a donc 33,5mV/°C.

b. bibliothèque MsTimer2

Nous avons vu une manière détournée de générer des interruptions à intervalles réguliers. Il existe une bibliothèque très simple permettant de faire la même chose.

Cette bibliothèque est disponible à l'adresse suivante : *http://playground.Arduino.cc/Main/MsTimer2*

Une fois installée, de la même façon que la bibliothèque *LCD Grove*, nous pouvons passer à son utilisation.

N'oublions pas la directive `#include` pour indiquer à *Arduino* que nous utilisons cette bibliothèque :

```
#include <MsTimer2.h>
```

Cette bibliothèque utilise le *timer2*, comme la fonction `tone()` pour produire une interruption lors de l'expiration d'un délai bien précis.

La fonction `MsTimer2::set()` sert à associer une fonction à l'expiration du délai.

```
MsTimer2::set(delai, fonction);
```

où `delai` (en millisecondes) est un entier 32 bits non signé,

`fonction` est la fonction qui doit être appelée lors de l'expiration du délai. A l'instar des autres interruptions que nous avons vu elle ne doit rien retourner (type `void`) et ne prend aucun paramètre. Son prototype est donc le suivant :

```
void fonction();
```

Ensuite il nous faut autoriser l'interruption par la fonction suivante :

```
MsTimer2::start();
```

Nous pouvons aussi désactiver l'interruption à tout moment par la fonction suivante :

```
MsTimer2::stop();
```

Voici un exemple d'utilisation :

```
#include <MsTimer2.h>

const byte led = 3;

void toggleLed()
{
  static boolean state = HIGH;

  digitalWrite(led, state);
  state = !state;
}

void setup()
{
  pinMode(led, OUTPUT);

  MsTimer2::set(300, toggleLed);

  MsTimer2::start();
}
void loop()
{
}
```

Conclusion

Lors de cette formation nous sommes allés de concepts simples aux concepts complexes. *Arduino* est d'une richesse incroyable grâce à ses fonctions natives et de tout ce que la communauté à pu lui apporter. N'oubliez pas que seule la pratique pourra vous permettre de vous sentir à l'aise dans l'écriture des programmes, et qu'aucune formation, ni aucun livre ne pourra la remplacer.

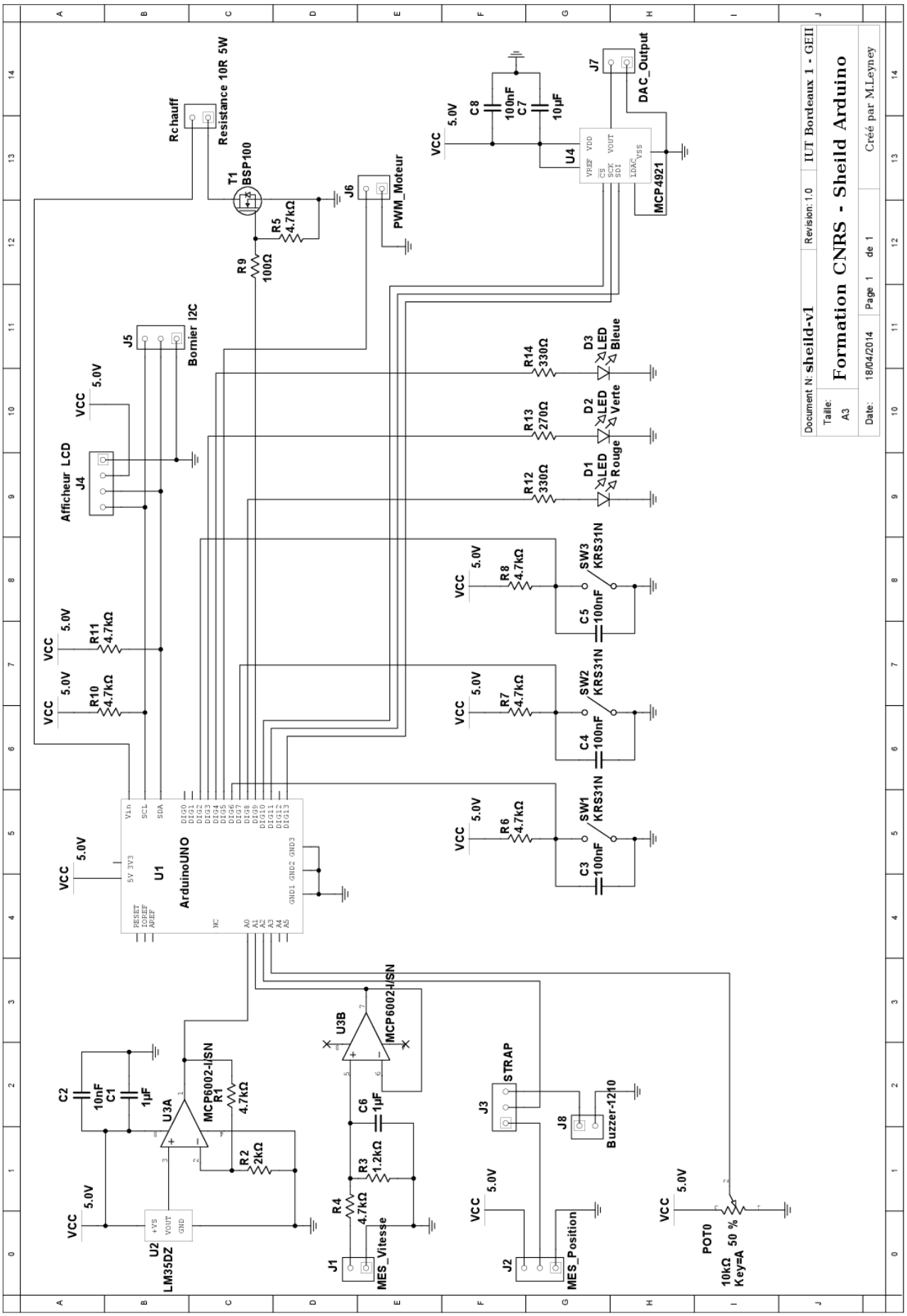
Si vous cherchez un bon exercice amusant qui reprend de nombreuses notions vues dans la formation je vous propose de réaliser jeu dans le style de Simon. Le principe est simple : une séquence aléatoire est jouée sur les LEDs et sur le buzzer. A chaque LED correspond un son et un bouton. Vous devez reproduire la séquence sur les boutons. En cas de succès une nouvelle LED aléatoire est ajoutée à la séquence qui est rejouée depuis le début. Ainsi de suite.

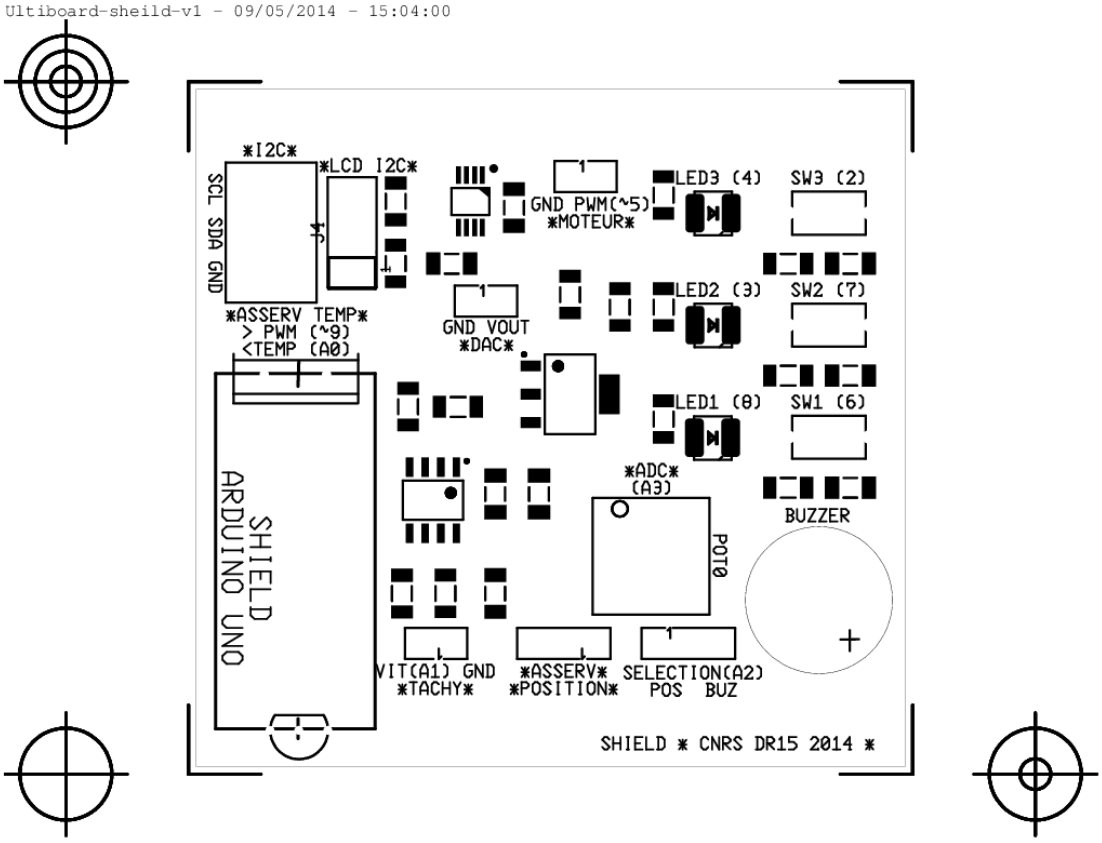
Pour réaliser cela vous aurez besoin :

- des variables, des tableaux,
- des boucles et de `if`,
- de `digitalRead()` et `digitalWrite()`,
- de `tone()`,
- de `delay()`,
- et de créer des fonctions !

Pour la partie aléatoire, faites un tour sur arduino.cc → Learning → Reference → `rand()`.

Annexes


Rchauff
Resistance 10R 5W
T1
BSP100
R5
4.7kΩ
R9
100Ω
J6
PWM_Moteur
J7
DAC_Output
U4
MCP4921
VCC
5.0V
C8
100nF
C7
10µF
J5
Bornier I2C
Afficheur LCD
J4
R14
330Ω
D3
LED
Bleue
R13
270Ω
D2
LED
Verte
R12
330Ω
D1
LED
Rouge
R11
4.7kΩ
R10
4.7kΩ
R8
4.7kΩ
SW3
KRS31N
C5
100nF
R7
4.7kΩ
SW2
KRS31N
C4
100nF
R6
4.7kΩ
SW1
KRS31N
C3
100nF
U1
ArduinoUNO
U3A
MCP6002-I/SN
U3B
MCP6002-I/SN
C2
10nF
C1
1µF
R1
4.7kΩ
R2
2kΩ
C6
1µF
R3
1.2kΩ
R4
4.7kΩ
U2
LM35DZ
J1
MES_Vitesse
J2
MES_Position
J3
STRAP
J8
Buzzer-12H0
POT0
10kΩ
Key=A
50 %
Document N: sheild-v1
Revision: 1.0
IUT Bordeaux 1 - GEII
Taille:
A3
Formation CNRS - Sheild Arduino
Date: 18/04/2014
Page 1 de 1
Créé par M.Leyney


I2C
LCD I2C
SCL SDA GND
J4
J1
GND PWM(~5)
MOTEUR
LED3 (4)
SW3 (2)
LED2 (3)
SW2 (7)
LED1 (8)
SW1 (6)
ASSERV TEMP
> PWM (~9)
<TEMP (A0)
GND VOUT
DAC
ADC
(A3)
POT0
BUZZER
SHIELD
ARDUINO UNO
VIT(A1) GND
TACHY
ASSERV
POSITION
SELECTION(A2)
POS BUZ
SHIELD * CNRS DR15 2014 *

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Figure 9: Table ASCII (asciitable.com)

Atmega328	Arduino	Note
PD0	0	RX
PD1	1	TX
PD2	2	
PD3	3	PWM
PD4	4	
PD5	5	PWM
PD6	6	PWM
PD7	7	
PB0	8	
PB1	9	PWM
PB2	10	SS / PWM
PB3	11	MOSI / PWM
PB4	12	MISO
PB5	13	SCK
PC0	A0	
PC1	A1	
PC2	A2	
PC3	A3	
PC4	A4	SDA
PC5	A5	SCL

Figure 10: Tableau de correspondance ATmega328 et Arduino