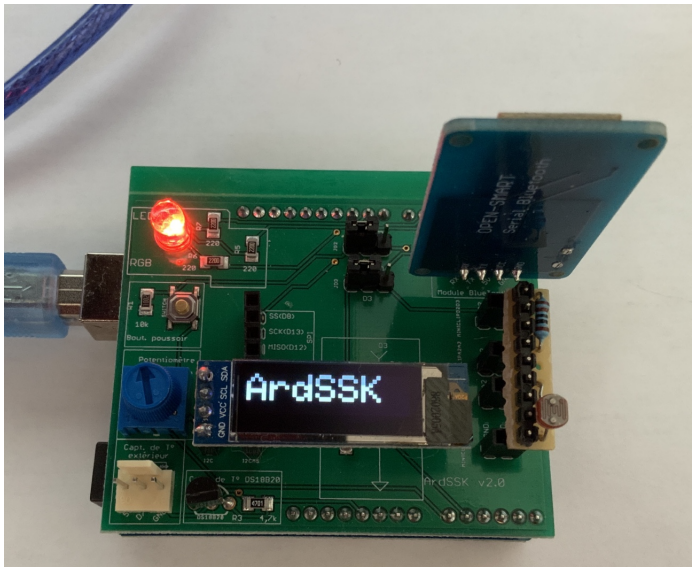
Arduino Scientific Starter Kit
Quelques éléments pour démarrer…



Version 3 février 2019
www.chimsoft.com

Activité 0

Table des matières

Activité 1

Pour commencer !

1 A propos de ce kit pédagogique...

L'ensemble du kit pédagogique que nous proposons se compose :

- du présent document ;
  - dans ce premier chapitre d'introduction, on s'attachera principalement à vérifier que tout est bien installé et que l'on arrive à communiquer à l'aide d'un câble USB entre l'ordinateur et la carte Arduino ;
  - dans le chapitre 2 il s'agit d'apprendre à commander une LED, étape essentielle dans l'utilisation d'un microcontrôleur ;
  - seconde étape non moins essentielle, la récupération de données à partir de la carte Arduino, c'est l'objet du chapitre 3 ;
  - dans le chapitre 4, nous présentons comment réaliser un projet ; en l'occurence la réalisation d'un TP de calorimétrie ;
  - le chapitre 5 est consacré aux différents capteurs proposés dans ce kit, exemple(s) d'utilisation et exercice(s) sont proposés pour chacun des capteurs ;
  - une annexe très volumineuse est proposée, elle contient :
    - une description de la partie matérielle de ce kit ;
    - quelques éléments de programmation de la carte Arduino (ce paragraphe peut vous servir de référence pour la programmation) ;
    - la description de quelques expériences et TP proposés dans ce kit ;
    - la correction commentée de tous les exercices proposés.
- d'un support matériel :
  - pour les chapitres 2 et 3, vous clipserez sur la carte Arduino le module ArdSSK afin d'apprendre à commander une LED (chapitre 2) puis à acquérir une donnée (chapitre 3) ;
  - pour les chapitres suivants soit vous utiliserez les connecteurs disponible sur le module ArdSSK, soit vous vous connecterez directement sur les ports de la carte Arduino, soit vous utiliserez une platine d'expérimentation.
- de programmes Arduino et python. Ces programmes sont présents dans le dossier `Programmes` que vous aurez téléchargé. Les programmes Arduino sont également accessibles directement dans l'environnement de développement Arduino (cf 4.2 page 11.)

Let's go !

2 Le micro-contrôleur Arduino

Sortie en 2005 comme un simple outil pour étudiant, la carte **Arduino** a initié une révolution DIY (Do It Yourself) dans l'électronique à l'échelle mondiale. Pour quelques euros, on dispose d'un système électronique extrêment simple à utiliser pour un néophyte par rapport à de l'électronique traditionnelle. La présence de ports d'entrée/sortie permet ainsi, grâce à quelques instructions que nous découvrirons, de récupérer l'information de capteurs et de commander des systèmes.

Le site officiel **www.arduino.cc** propose de multiples ressources sur les produits Arduino.

Il existe maintenant d'innombrables types de cartes. Nous nous concentrerons dans ce guide sur l'utilisation du modèle de « base », la carte **Arduino Uno**.

Cette carte est fondée sur un micro-contrôleur *ATMega328* cadencé à 16 MHz. Les connecteurs situés sur les bords extérieurs du circuit imprimé permettent d'enficher une série de modules complémentaires (appelés shield). Cette carte peut se programmer avec le logiciel Arduino, grâce à un cordon de connexion USB.

Les entrées/sorties de la carte **Arduino** sont détaillées sur la figure 1.1.

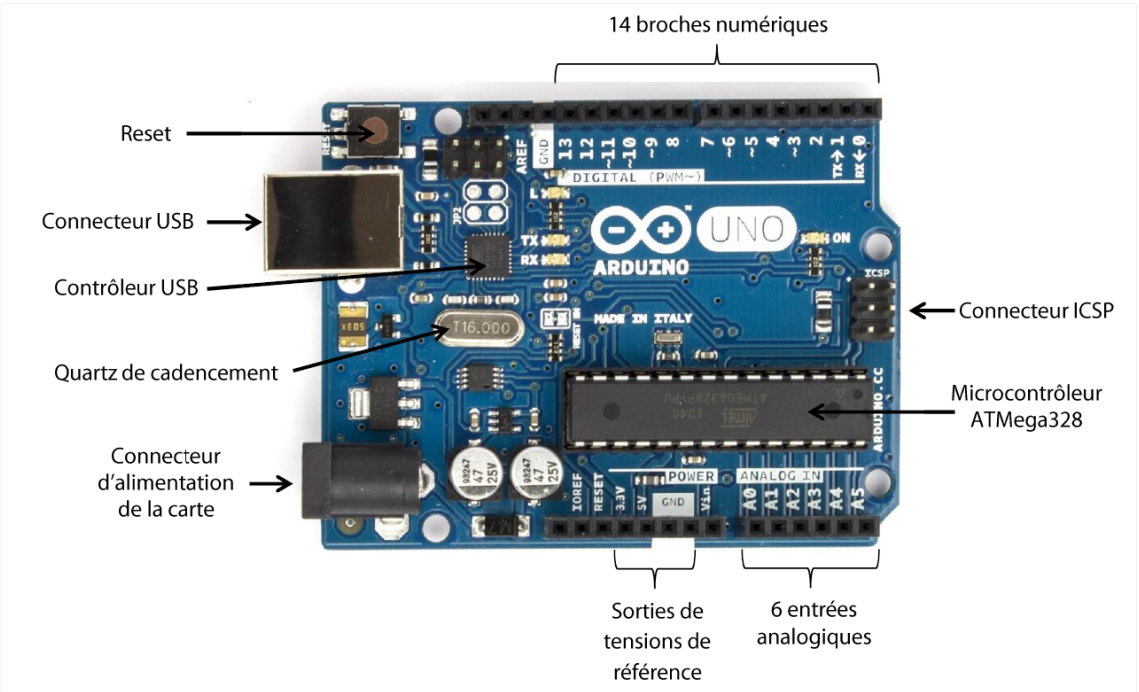


FIGURE 1.1 – Entrées-Sorties de la carte Arduino Uno

Cette carte fonctionne, entre autre, grâce à :

- d'un micro-contrôleur ATMega328 ;
- d'un quartz 16MHz (horloge de cadencement) ;
- d'une connexion USB (avec son contrôleur associé) qui permet la programmation du micro-contrôleur ainsi que l'alimentation de la carte ;

- d'un connecteur d'alimentation jack (nécessaire si le cordon USB est déconnecté après programmation), une tension comprise entre 7 et 12 V est requise ;
- et d'un bouton de réinitialisation (reset).

Cette carte met à notre disposition :

- 14 broches numériques d'entrées/sorties : elles fonctionnent en tout ou rien, offrant en entrée ou en sortie soit une tension de 0 V, soit une tension de 5 V [1] ;
- de 6 entrées analogiques acceptant des valeurs comprises entre 0 V et 5 V ;
- différentes sorties de tension : masse, 5 V (éventuellement 3,3 V)

La tension de référence pour l'**Arduino Uno** est 5 V. D'autres cartes fonctionnent en 3,3 V. Certains capteurs acceptent les 2 tensions, d'autres sont nettement plus susceptibles !

Nous le redirons pas la suite mais... il est HORS DE QUESTION de travailler avec des tensions supérieures à 5 V, de faire circuler des courants de plus de quelques mA, de faire tourner un moteur !

- la sortie + 5 V (dite VCC) que l'on utilisera fréquemment pour alimenter des capteurs ne peut délivrer que 200 mA maximum (c'est encore pire avec la sortie 3,3 V : 50 mA) ;
- les sorties numériques ne peuvent délivrer plus de 40 mA (20 mA recommandé) et 200 mA maximum cumulés sur l'ensemble de ces broches.

Dans la « vraie vie », vous serez certainement très rapidement confrontés à des systèmes nécessitant plus de puissance. Il faudra alors utiliser une alimentation autonome pour le circuit de puissance, l'arduino ne gérant que la partie commande numérique.Savoir commander un transistor, un relais, un circuit de commande d'un moteur... n'est qu'un problème matériel !

A la fin du chapitre 2, vous saurez comment commander une LED et à la fin du chapitre 3 comment récupérer des données de la carte Arduino ; vous aurez donc toutes les clés pour aborder des systèmes plus complexes. Les chapitres 4 et 5 vous montreront comment faire.

3 Quelques installations !

Bon... c'est le plus pénible !

Toutes les informations sont données sur la page : `www.chimsoft.com/licences/ArdSSK/`.
L'identifiant et le mot de passe sont joints à la licence d'utilisation.

L'étape 2 décrite sur la page en question est l'étape indispensable pour la suite.
Pour une utilisation « élève de lycée » c'est dans un premier temps la seule vraiment nécessaire.
Dans un premier temps, si vous n'êtes familiarisés ni avec Arduino, ni avec Python vous pouvez vous contenter de cette étape.

A terme, *a priori*, un couplage Arduino-Python « risque » vite de s'imposer, en particulier pour l'exploitation de données expérimentales ; l'installation de l'environnement de programmation en python s'avèrera alors nécessaire.

En téléchargeant le dossier `CleArdSSK`, vous disposerez de tout !

1. Sauf ruse de sioux pour certaines avec une sortie dite PWM, ce point sera abordé page 17

4 En route vers notre premier programme...

La carte **Arduino** est une carte programmable, à volonté. Le langage de base n'est pas Python, mais le langage C/C++ auquel a été ajouté quelques spécificités propres à l'Arduino (un mix entre le langage C et C++).

Ne jetez pas le bébé avec l'eau du bain ! Ce n'est pas parce que vous ne savez pas programmer en C (ni en python d'ailleurs) que vous ne pouvez pas utiliser l'Arduino.
Nous allons vous proposer dans ce guide tout le code nécessaire pour commencer ; dans un premier temps il « suffira » juste de faire un copier/coller de ce code et de l'adapter à vos besoins. Si on ne cherche pas la virtuosité en programmation, quelques règles simples (résumées dans l'annexe page 101) permettent ensuite de réaliser ce que l'on souhaite !

La démarche de conception d'un programme est donnée ci dessous :

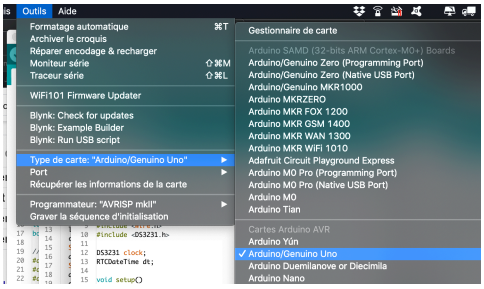
1. on utilise le logiciel Arduino pour écrire son programme ;
2. on télécharge (télé-verse) le programme dans la carte (il y a une étape de compilation intermédiaire qui vous signalera parfois quelques bugs !) ;
3. le programme étant téléchargé, la carte fonctionne en autonomie si elle a sa propre source d'énergie, ou est alimentée via le port USB si elle reste connectée au PC.

4.1 Éditez votre premier programme

a Premiers réglages

Connectez la carte Arduino à l'ordinateur via le cable USB et démarrez le logiciel Arduino (icône ci-contre). En fait, l'ordre importe peu.
Une fois le logiciel démarré :

- allez dans le menu *Outils* puis *Type de carte* et vérifiez que *Arduino Uno* est bien coché (le cocher sinon) ;
- vérifiez que la carte est bien reconnue en allant dans *Outils*, *Port*. La carte est reconnue si un port est affiché, par exemple "Com 3" ou `/dev/cu.usbmodem14201`.



,

b Structure du programme

A l'ouverture du logiciel, on peut remarquer qu'un bout de code est déjà proposé :

Listing 1.1 – Programme vide

```
1 void setup() {
2   // put your setup code here, to run once:
3
4 }
5
6 void loop() {
7   // put your main code here, to run repeatedly:
8
9 }
```

Sans nous intéresser, pour l'instant, à la syntaxe ; on dispose ici de la structure même de tout programme Arduino :

- une fonction (ou procédure) **setup** d'initialisation qui ne sera exécutée qu'une seule fois ;
- une fonction (ou procédure) **loop** qui, comme son nom l'indique également, sera exécutée dans une boucle infinie.

c Le fameux "Hello Word"

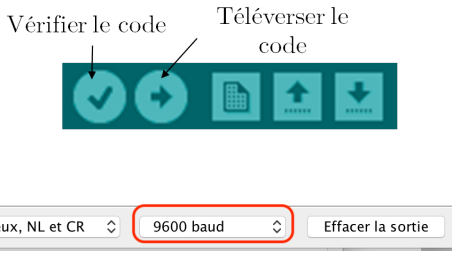
On désire afficher, sur l'écran de l'ordinateur, une seule fois "Hello Word".

Modifiez le programme afin d'obtenir le code suivant (sans oublier les ; et en respectant la casse).

Listing 1.2 – Hello world

```
1 void setup() {
2   Serial.begin(9600) ;
3   Serial.println("Hello world !") ;
4 }
5
6 void loop() {
7   // put your main code here, to run repeatedly:
8 }
```

Une fois le code créé, cliquez sur l'icône de vérification du code (corrigez le code si nécessaire !).



Téléversez ensuite le programme dans la carte, puis sélectionnez *Outils/Moniteur série* ou cliquez sur l'icône loupe à droite de la barre d'icônes. Si tout se passe bien, le message apparaît (assurez vous qu'une vitesse de communication à 9600 bauds est bien sélectionnée dans le moniteur série.)
Que s'est-il passé grâce à ces deux nouvelles lignes de code ?

Lorsqu'un programme Arduino est exécuté, un objet nommé **Serial** est automatiquement créé (même si on ne l'utilise pas). Cet objet émule un port série et permet de communiquer de façon bidirectionnelle entre l'ordinateur et la carte Arduino.

- on précise (ligne 2) que le port série va communiquer à une vitesse de 9600 bauds. On n'est pas pressé, on communique à 9600 bits par secondes (c'est quand même 8 fois plus rapide que le Minitel !) ;
- on demande (ligne 3) d'afficher un message (et d'effectuer un retour à la ligne).

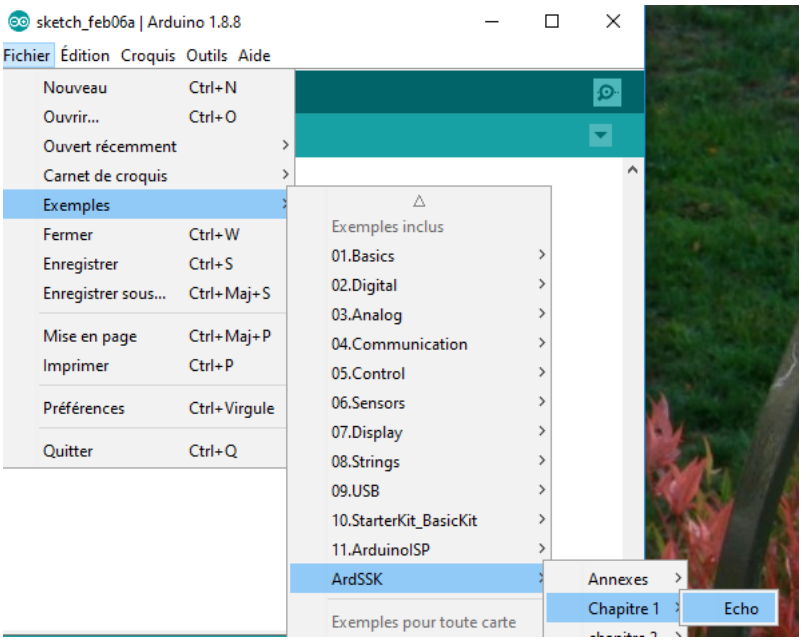
Vous pouvez tenter une variante en plaçant le code de la ligne 3 dans la boucle !

4.2 Un échange d'information bidirectionnel

Le moniteur série de l'Arduino permet non seulement d'afficher des messages reçus de l'arduino mais également d'en envoyer.
Codez le programme suivant (ou, ouvrez le fichier *Echo*.

Dans tout ce document, vous avez deux solutions pour accéder aux programmes qui vous sont fournis :

- soit à partir du dossier *Programmes* dans lequel vous trouverez un dossier par chapitre à l'intérieur duquel se trouvent les différents programmes. Ici, le programme *Echo* se trouve donc dans le sous-dossier *Chapitre 1*.
- soit directement dans le programme de l'IDE d'Arduino en suivant le lien *Fichier* > *Exemples* (ou *Carnet de croquis*) > *ArdSSK* > *Chapitre 1* > *Echo*



Listing 1.3 – Echo

```
1  void setup() {
2    Serial.begin(9600) ;
3    Serial.println("Je suis prêt !") ;
4  }
5
6  void loop() {
7    if (Serial.available())
8    {
9      String message = Serial.readString() ;
10     Serial.println(message) ;
11   }
12 }
```

Dans la zone de texte en haut de l'éditeur série introduisez votre texte et cliquez sur *Envoyer* ; le message apparait alors dans la zone de réponse.

Que ce passe-t-il, cette fois, dans la boucle infinie :

- l'objet Serial scrute en permanence le port série ; si aucune donnée n'arrive, on ne fait rien !
- si par contre une donnée arrive sur le port série, on attend la fin de la ligne et on stocke la chaine de caractères reçue dans la variable *message* (ligne 9) ;

- et on renvoie vers le PC la chaine de caractère en question pour qu'elle s'affiche dans le moniteur série.

Ce que l'on vient de faire peut apparaître tout bête mais, c'est la clé de la communication entre Arduino et le monde extérieur :

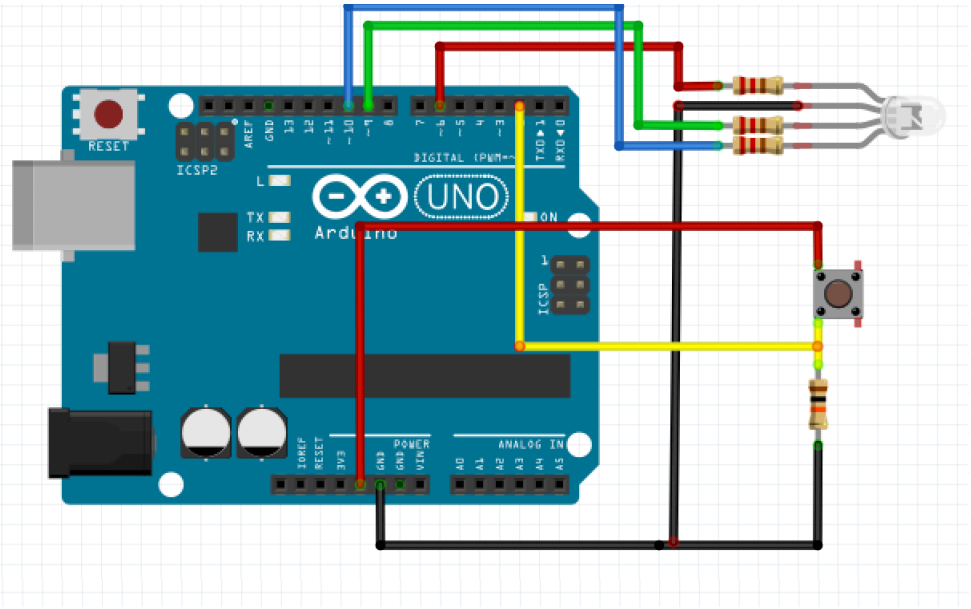
- on peut lui envoyer des instructions ; charge au reste du programme d'interpréter les ordres donnés comme nous le verrons dans le chapitre suivant ;
- on peut recevoir des données, par exemple une acquisition d'un capteur ; si celles-ci sont bien formatées (comme nous le verrons dans le chapitre 3), une exploitation aisée pourra être faite au niveau du PC.

Activité 2

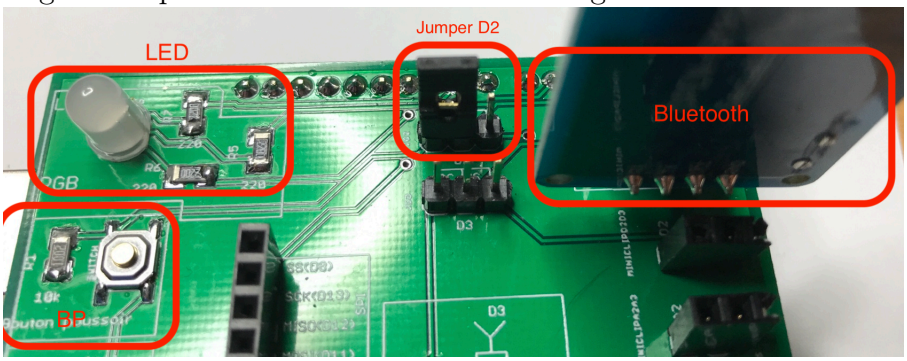
LED Blinking !

Faire clignoter une LED est, classiquement, l'équivalent du « Hello World » dans tout tutoriel sur l'utilisation d'un microcontrôleur, alors commençons par là.

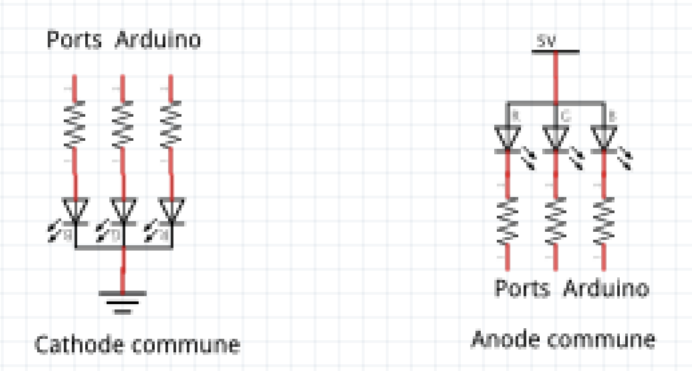
Ce chapitre est long, difficile sur la fin ; ne vous fiez pas à l'apparente facilité de certaines parties... chaque pas compte ! Vous n'êtes pas obligé de tout comprendre et tout maitriser pour utiliser l'Arduino. En particulier, on peut tout à fait se débrouiller sans connaissance du langage C++ utilisé pour programmer la carte. Dans un premier temps, analysez le rôle de chaque instruction et commencez par faire du copier/coller d'un programme que l'on vous propose puis faites les modifications nécessaires pour atteindre l'objectif visé. Quelques éléments de programmation vous sont proposés en annexe page 101.



Nous utiliserons dans ce chapitre, bien sûr, la LED RGB, le bouton poussoir (il faudra positionner le cavalier comme sur la photo ci-dessous) ainsi que le module Bluetooth qui devra être, le moment venu, enfiché comme sur la figure (composants du module vers l'extérieur).
Le câblage correspondant est schématisé sur la figure ci-dessus.



La LED RGB proposée est une LED à cathode commune (reliée à la masse). Appliquer un potentiel positif sur l'une des autres pattes de la LED permet d'allumer la couleur correspondante. Si vous utilisez une LED à anode commune, la logique est inversée !



1 Premières étapes

1.1 L'arnaque du siècle : un programme qui ne fait rien !

Ouvrir l'IDE d'Arduino, taper le programme suivant, enregistrez le puis cliquer sur le bouton **Televerser**

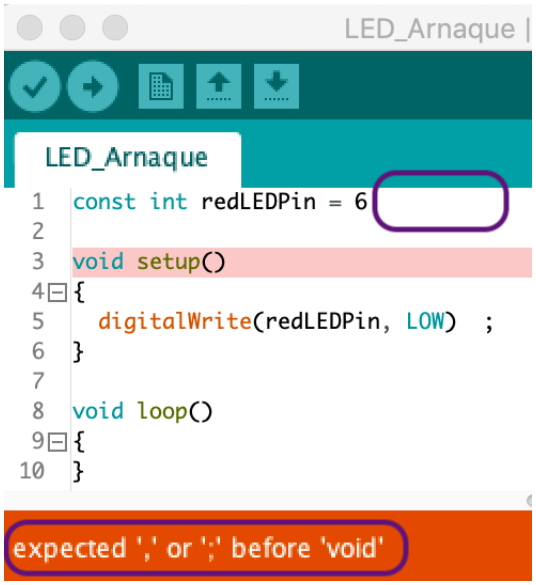
Et bien, c'est magnifique... ce programme ne fait rien, mais le fait bien !
En fait, la situation aurait pu être pire, vous auriez pu, par exemple oublier un point virgule à la fin d'une ligne, ou écrire redLedPin sans respecter la casse, ou autre...

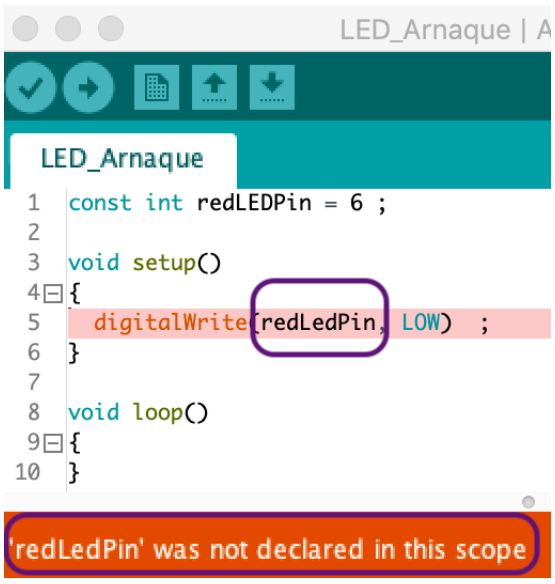
Remplacez maintenant LOW par HIGH dans le programme précédent et *fiat lux* !

Vous pouvez aussi tenter une variante en changeant la ligne 5 par analogWrite(redLEDPin, 125).

Nous pouvons ainsi :

- allumer la LED **digitalWrite(port, HIGH)** ;
- l'éteindre **digitalWrite(port, LOW)** ;
- moduler son éclairement **analogWrite(port, valeur)** avec *valeur* un nombre entier compris entre 0 et 255.

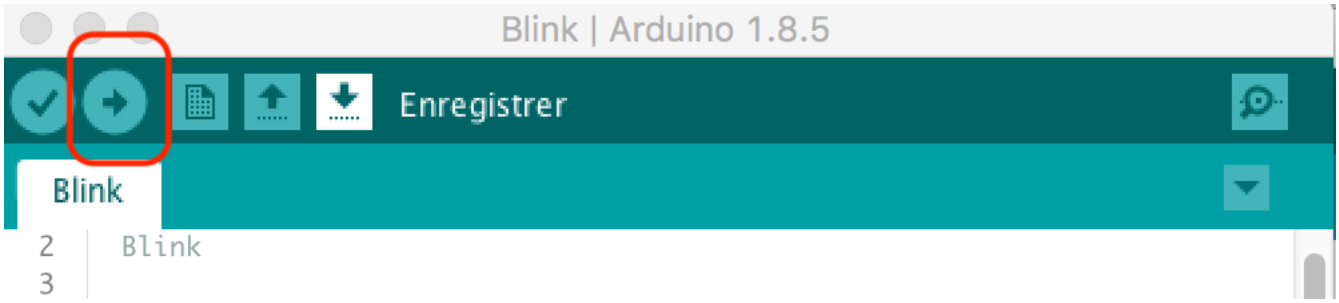




1.2 L'exemple de l'IDE d'Arduino

Un certain nombre de programmes sont proposés dans l'IDE d'Arduino, ceux-ci permettent d'aborder différentes fonctionnalités. Lorsque l'on charge une bibliothèque pour l'utilisation de tel ou tel capteur, quelques exemples d'application sont également proposés.

Sélectionnez le script à l'aide des différents sous-menus de l'IDE d'Arduino :
`Fichier>Exemples>01.Basics>Blink`.
Cliquez ensuite sur le bouton `Téléverser`. Le programme est alors compilé et l'exécutable chargé dans l'Arduino.



Miracle... une LED orange clignote ! Vous ne la voyez pas, regardez en dessous du shield inséré sur l'Arduino. Il s'agit de la LED soudée sur la carte même de l'Arduino et reliée au port **13**.

Regardez de plus près le programme (après les commentaires apparaissant en grisé) :

Listing 2.1 – Programme Blink

```
1 // the setup function runs once when you press reset or power
    the board
2 void setup() {
3   // initialize digital pin LED_BUILTIN as an output.
4   pinMode(LED_BUILTIN, OUTPUT);
5 }
6
7 // the loop function runs over and over again forever
8 void loop() {
9   digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH
      is the voltage level)
```

```
10    delay(1000);                          // wait for a second
11    digitalWrite(LED_BUILTIN, LOW);       // turn the LED off by
          making the voltage LOW
12    delay(1000);                          // wait for a second
13  }
```

Bon... c'est peut-être votre premier vrai programme en C++. Oublions pour l'instant la syntaxe et concentrons nous sur ce que l'on souhaite faire dans ce programme.

Tout programme Arduino comprend :

- une fonction **setup()** qui n'est éxécutée qu'une seule fois et qui permet d'initialiser le système ;
- une fonction **loop()** qui est appelée dans une boucle infinie ; c'est dans cette fonction que l'on devra décrire le fonctionnement du système.

Dans le cas présent :

- on précise dans l'étape d'initialisation à l'aide de la fonction **pinMode** que le port d'entrée/-sortie associé à la LED de la carte (la constante $LED\_BUILTIN$ vaut 13) doit être en mode « sortie »... vous avez tout de suite deviné que si l'on souhaite avoir une entrée sur ce port, il suffira d'écrire INPUT !
- et dans la boucle :
  - grâce à la fonction **digitalWrite** on fait passer le niveau logique du port de la LED à haut (5 V en l'occurence), la LED s'allume ;
  - on attend 1000 ms grâce à l'instruction **delay** ;
  - on fait passer le niveau logique du port de la LED à bas (0 V), la LED s'éteind ;
  - on attend à nouveau 1000 ms.

Et ça clignote ! Changez 1000 en 500 et lancer à nouveau la compilation et le téléversement dans la carte.

Questions syntaxe du langage C++, que remarque-t-on ici :

- les commentaires suivent l'instruction // ou sont compris entre /* et */ ;
- le type de retour de la fonction doit être précisé, ici les fonctions ne retournent rien, on précise le type **void** ;
- tout le code inclus dans une fonction, et plus tard dans un bloc if, while... est inclus entre une parenthèse ouvrante **{** et une fermante **}**.
- une ligne d'instruction doit se terminer par **;**.

1.3 Jouons avec la LED RGB...

Sur la platine du kit a été cablée une LED RGB (avec ses résistances de protection). Les broches rouge, bleue et verte sont respectivement connectées aux ports D6, D9 et D10 de l'Arduino. Ouvrir le programme Blink_Red dans le dossier d'exemples mis à votre disposition, téléverser le programme dans la carte (la compilation est automatique) et, la LED devrait clignoter en rouge. C'est exactement le même programme que précédemment, on a juste associé le port D6 à une variable.

```
1  const int redLEDPin = 6 ;
```

En C, les variables doivent être typée (variable entière de type **int** ici), on précise (mais on pourrait s'en passer) que c'est une constante dans tout le programme. Dans certains programmes fournis, vous pouvez trouver une variante pour définir les constantes du programme (cf ci-dessous), la

syntaxe est un peu particulière ici puisqu'il n'y a pas de ; en fin de ligne (l'erreur est facilement détectée à la compilation).

```
1 #define redLEDPin 6
```

1.4 Utilisation d'une sortie analogique

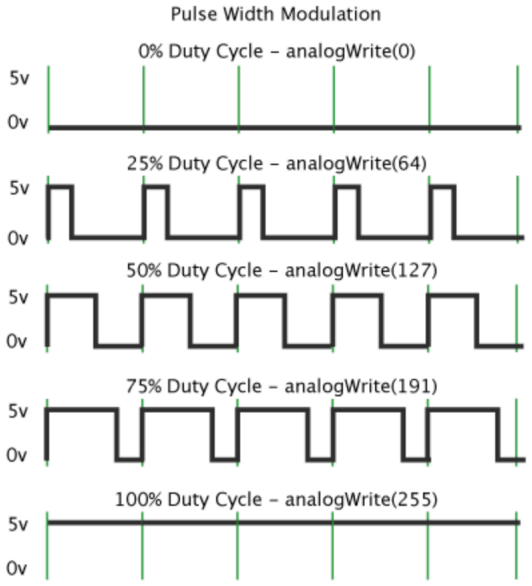
Les ports D3, D5, D6, D9, D10 et D11 de l'Arduino peuvent se comporter comme sortie numérique (instruction **digitalWrite**) ou comme sortie analogique (instruction **analogWrite**). Le programme Blink_Variable met en œuvre cette possibilité.

Listing 2.2 – Programme LED_Variable

```
1  const int redLEDPin = 6 ;
2
3  void setup() {
4    pinMode(redLEDPin, OUTPUT);
5  }
6
7  void loop() {
8    for (int i = 0 ; i < 255 ; i=i+10)
9    {
10     analogWrite(redLEDPin, i) ;
11     delay(200);
12   }
13 }
```

Au lieu d'appliquer une tension soit nulle soit égale à 5 V sur la LED, on impose, grâce à la fonction **analogWrite** une tension « moyenne » égale à $\frac{i \times 5}{255} V$.
En réalité, ce n'est pas vraiment une tension continue qui est appliquée sur le port correspondant. On a une fonction dite PWM (Pulse With Modulation). Si on demande une tension de 5V, on a, en sortie, 5 V en continu. Mais, si, par exemple on demande une tension de 1 V, le système délivre 5 V sur un cinquième du temps.

Le schéma ci-contre est extrait de la documentation d'Arduino.
L'intervalle de temps entre deux traits verts correspond à 2 ms, la fréquence du PWM étant de 500 Hz [a].



a. Les ports 6 et 11 ont une fréquence de 980 Hz. On peut, au besoin, modifier la fréquence...

Notons enfin la syntaxe utilisée pour créer une boucle **for** en langage C. La variable i, de type entier, est incrémentée par pas de 1 entre les valeurs 0 et 255 incluses. Les parenthèses permettent de délimiter le bloc de code exécuté dans la boucle.

Exercice 1 **Programmez un feu tricolore...** la LED rouge doit s'allumer 1000 ms, puis la verte 500 ms, puis la bleue 500 ms et ainsi de suite !
Pour obtenir une couleur orange, une possibilité est d'utiliser RED = 255, GREEN = 165, BLUE = 0 ; modifiez le programme précédent afin d'avoir de l'orange à la place du bleu.

1.5 Peut mieux faire !

Les programmes que l'on vient de voir sont, certes, fonctionnels mais la façon dont ils sont conçus risquent de poser problème lorsque l'on cherchera à aller un peu plus loin. Par exemple, on peut imaginer envoyer des ordres à l'Arduino qui les interprètera et agira sur les sorties en conséquence. Considérons, par exemple, la séquence de consignes suivantes :

- Arduino, allume la lampe rouge ⟶ digitalWrite(redLEDPin, HIGH) ;
- Arduino, éteint la lampe ⟶ digitalWrite(redLEDPin, LOW) ;
- Arduino, réallume la lampe ⟶ digitalWrite(redLEDPin, LOW) ;

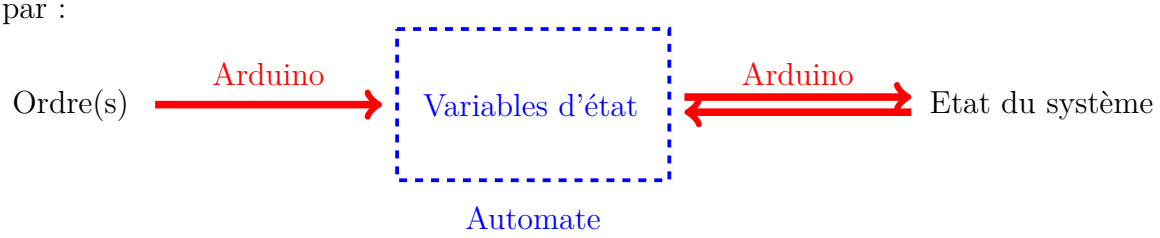
jusque là...tout va bien !

- Arduino, allume la lampe rouge à 50% ⟶ analogWrite(redLEDPin, 128) ;
- Arduino, éteint la lampe ⟶ digitalWrite(redLEDPin, LOW) ;
- Arduino, réallume la lampe ⟶ analogWrite(redLEDPin, ? ? ?) ;

et là, sauf à avoir codé réallume la lampe à 50%, on est bloqué car on ne connait pas l'état de la lampe quand elle est allumée. Il faudrait stocker cette valeur dans une variable ; il en va de même si on joue sur le délai d'une lampe qui clignote.

Tentons de modifier le schéma :

par :



Cette façon de procéder revient, en fait, à créer un automate :

- l'Arduino reçoit des ordres et met à jour un jeu de variables ;
- le moment venu il met à jour l'état du système à partir de ces variables, voire même modifie certaines de ces variables en fonction de mesures réalisées.

.

2 Je suis le maître du temps !

Faire clignoter une LED peut paraître extrêmement simple de prime à bord... si c'est la seule « préoccupation » du microcontrôleur. De tels systèmes ne peuvent exécuter plusieurs tâches en parallèle (comme sur un ordinateur par exemple). Si, par exemple, dans la boucle principale on allume la LED, on attend 500 ms, on éteind la LED, on attend 500 ms, on exécute une autre tâche... forcément la LED restera éteinte plus de 500 ms. Ce n'est pas « grave » si c'est juste pour faire clignoter une LED mais si on donne des ordres pour faire tourner un moteur, on risque d'avoir des saccades !

2.1 Delay. . . faute de mieux

Considérons le programme Blink_Delay.

Listing 2.3 – Programme Blink_Delay

```
1  #define redLEDPin 6
2  int ledState = LOW ;
3  long delai = 1000 ;
4
5  void setup()
6  {
7    Serial.begin(9600) ;
8    pinMode(redLEDPin, OUTPUT);
9  }
10
11 void loop()
12 {
13   if (ledState == LOW)
14   {
15     ledState = HIGH ;
16   }
17   else
18   {
19     ledState = LOW ;
20   }
21   digitalWrite(redLEDPin, ledState);
22   Serial.println(millis()) ;
23   delay(delai) ;
24 }
```

Par rapport au programme `Blink`, nous avons :

- modifié le code de changement d'état de la LED en sauvegardant l'état actuel dans une variable *ledState*. . .notre fameux automate !
- ajouté le code nécessaire pour afficher dans le moniteur série le temps (en ms) correspondant au changement d'état de la LED ;

.
La LED clignote toutes les secondes et les valeurs du temps affichées dans le moniteur série correspondent tout à fait à ce qu'on attend.
Maintenant, supposons que le programme fasse appel à une routine qui prend un « certain » temps. Nous avons dans le programme `Blink_Echec` implémenté une fonction qui bloque le microcontrolleur entre 50 et 800 ms.

Listing 2.4 – Programme Blink_Echec (extrait)

```
1  {
2  void fonctionQuiPrendDuTemps()
3  {
4    long randomDelay = random(50,800) ;
5    delay(randomDelay) ;
6  }
7
8  void loop()
9  {
10   if (ledState == LOW)
```

```
11    {
12      ledState = HIGH ;
13    }
14    else
15    {
16      ledState = LOW ;
17    }
18    digitalWrite(redLEDPin, ledState);
19    Serial.println(millis()) ;
20    delay(delai) ;
21  }
```

Comme on peut s'y attendre, on constate de visu ou sur le moniteur série (356, 1905, 3278, 4487, 5967, 7289...) que la LED ne clignote plus régulièrement ! Le phénomène est peut-être exagéré ici mais le problème existe réellement.

2.2 Mesurons le temps qui passe...

Au lieu de bloquer le système pendant 1000 ms, on pourrait très bien mesurer le temps écoulé depuis le dernier changement d'état de la lampe... et changer à nouveau l'état de la LED lorsque cette durée dépassera 1000 ms.
C'est l'approche proposée dans le programme Blink_MesureDuree

Listing 2.5 – Programme Blink_MesureDuree

```
1  #define redLEDPin 6
2  int ledState = LOW ;
3  long delai = 1000 ;
4  long lastTime = 0 ;
5
6  void setup()
7  {
8    Serial.begin(9600) ;
9    pinMode(redLEDPin, OUTPUT);
10 }
11
12 void fonctionQuiPrendDuTemps()
13 {
14   long randomDelay = random(50, 800) ;
15   delay(randomDelay) ;
16 }
17
18 void loop()
19 {
20   fonctionQuiPrendDuTemps() ;
21   if (millis() - lastTime > delai)
22   {
23     lastTime = millis() ;
24     Serial.println(lastTime) ;
25     if (ledState == LOW)
26     {
27       ledState = HIGH ;
28     }
29     else
30     {
```

```
31        ledState = LOW ;
32      }
33      digitalWrite(redLEDPin, ledState);
34    }
35  }
```

La variable *lastTime* stocke le temps correspondant à un changement d'état de la lampe. Lorsque (ligne 21) l'écart entre le temps courant et *lastTime* devient supérieur au *delai*, on met à jour la variable *lastTime* et on change l'état de la lampe.

Qu'observe-t-on ? C'est mieux mais on n'a quand même pas un top toutes les secondes !
A l'oeil on observe un clignotement régulier mais les sorties du moniteur série correspondent, par exemple, à 1278, 2289, 3793, 5249, 6383. . .. Le délai de la seconde est à peu près respecté mais il y a quelques irrégularités. Le décalage provient du temps mis pour exécuter le bloc `if` entre les lignes 23 et 33 (et en particulier l'instruction println).

2.3 Gloire au timer !

En fait, il y a au cœur du microcontrôleur un circuit (plusieurs en réalité) qui permet de gérer des tâches à intervalle régulier : il s'agit du **Timer**. Celui-ci, à intervalle de temps régulier va bloquer le système et lancer l'exécution d'une fonction que l'on précisera.

Listing 2.6 – Programme Blink_Timer

```
1  #include<MsTimer2.h>
2
3  #define redLEDPin 6
4  int ledState = LOW ;
5  long delai = 1000 ;
6
7  void setup()
8  {
9    Serial.begin(9600) ;
10   pinMode(redLEDPin, OUTPUT);
11   MsTimer2::set(delai,job) ;
12   MsTimer2::start() ;
13 }
14
15 void loop()
16 {
17   fonctionQuiPrendDuTemps() ;
18 }
19
20 void job()
21 {
22   if (ledState == LOW)
23   {
24     ledState = HIGH ;
25   }
26   else
27   {
28     ledState = LOW ;
29   }
30   digitalWrite(redLEDPin, ledState);
31   Serial.println(millis()) ;
```

```
32    delay(delai) ;
33  }
34
35  void fonctionQuiPrendDuTemps()
36  {
37    long randomDelay = random(50,800) ;
38    delay(randomDelay) ;
39  }
```

L'importation d'une bibliothèque (ligne 1) est nécessaire pour l'utilisation du timer. Nous utilisons, ici, MsTimer2. Le code de cette bibliothèque a été placé (cf chapitre 1) dans le dossier libraries du dossier Arduino.
Ligne 12, on initialise ce timer en lui précisant un intervalle de temps (en ms) et le nom de la fonction à exécuter (job ici).
Ligne 13 : on déclanche le Timer... le code contenu dans la fonction job() est alors exécuté tous les *delai* ms. Notons que le changement d'état de la LED ne se fait plus dans la boucle **loop** mais dans la fonction **job**

Et... victoire, ça fonctionne plutôt bien ! Sur la console du moniteur série on lit 998, 1998, 2999, 3999, 5000, 6000, 7001...Ce n'est pas encore une horloge atomique, on se décale environ d'une ms toutes les s, mais c'est déjà pas si mal !

L'utilisation d'un timer est certes séduisante mais ne peut, malheureusement, pas être utilisée systématiquement. En interne, le fonctionnement de ce timer nécessite l'utilisation de certaines ressources machines. Il est donc impossible d'utiliser ces mêmes ressources dans la routine appelée par le timer !
Ainsi :

- on ne peut pas utiliser d'instruction `delay(...)` avec un timer ;
- certaines sorties PWM ne sont pas compatibles ;
- certains capteurs sur port I2C (cf chapitre 5) peuvent ne pas fonctionner.

2.4 Maître du temps... pas tout à fait quand même !

Les solutions proposées précédemment nous permettent de gérer relativement facilement le temps sur une échelle de quelques millisecondes à, disons, une heure. Mais...

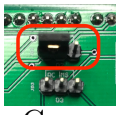
- en dessous de la milliseconde, nous risquons vite d'atteindre les limites, en terme de rapidité, de la carte **Arduino Uno** et des capteurs dont nous disposons facilement. Des solutions pour travailler à l'échelle de la microseconde existent mais, il faut y mettre le prix !
- au-delà de plusieurs heures d'utilisation, la solution proposée n'est plus très efficace. Si on veut, par exemple, suivre l'évolution de la température sur une journée avec une mesure toute les heures, nous risquons une dérive du temps et, surtout, la moindre micro coupure dans l'alimentation conduira à une réinitialisation du temps. Une solution « simple » existe, nous l'aborderons dans la partie 5. Cette solution consiste à associer à la carte Arduino une horloge en temps réel (cf page 81).

3 Commander la LED... version Hardware

C'est bien beau tout ça, mais on aimerait bien pouvoir maintenant interagir avec la LED : l'allumer, l'éteindre, varier l'intensité ou les couleurs par exemple. *A priori*, la solution « intuitive » consisterait, ici, à utiliser une approche électronique « classique » avec interrupteur, boutons poussoir et potentiomètre mais... l'Arduino ne servirait pas à grand chose !
Nous proposons dans un premier temps d'utiliser quand même un bouton poussoir ou un potentiomètre ; nous verrons ensuite qu'il y a plus « simple » : envoyer des ordres au microcontroleur à l'aide de commandes.

3.1 Un bouton marche/arrêt

a Un bouton poussoir en mode continu



Un bouton poussoir est câblé sur le shield. Pour l'utiliser, le cavalier **D2** doit être placé côté bouton poussoir.

L'état du bouton (appuyé ou relaché) peut être obtenu en lecture sur le port **D2**. Comme son état est binaire... vous avez deviné, bravo !, que c'est l'instruction **digitalRead** qu'il faut utiliser. Celle-ci va retourner 0 si le bouton est relâché et 255 s'il est appuyé.

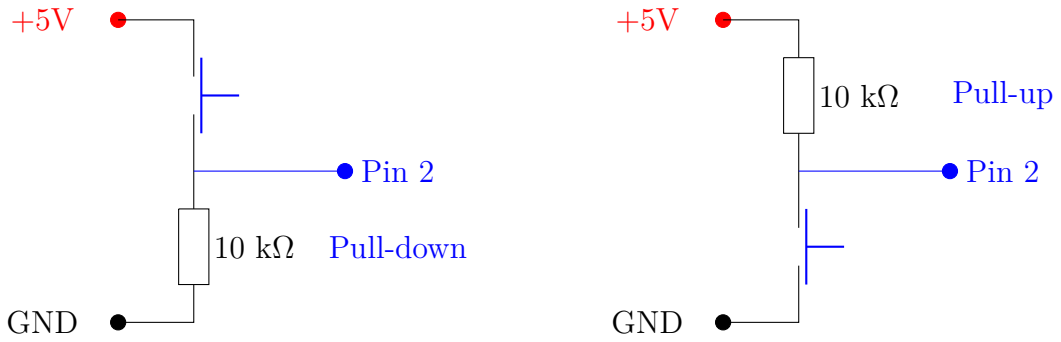
On obtient le programme LED_BoutonAppui.

Listing 2.7 – Programme LED_BoutonAppui

```
1  #define redLEDPin    6
2  #define buttonPin    2
3
4  void setup()
5  {
6    pinMode(redLEDPin, OUTPUT);
7    pinMode(buttonPin, INPUT) ;
8  }
9
10 void loop()
11 {
12   int button = digitalRead(buttonPin) ;
13   digitalWrite(redLEDPin, button) ;
14 }
```

A chaque parcours de la boucle, on lit l'état du bouton poussoir et on gère l'état de la LED en conséquence. La fonction `digitalRead` retourne en fait soit la valeur HIGH, soit la valeur LOW. Elle est stockée ici dans un nombre entier mais on aurait pu également utiliser un booléen (en C++ 1 est interprété comme vrai et 0 comme faux).

Le câblage du bouton poussoir mérite un peu d'attention au risque de réaliser un court-circuit entre la sortie 5V et la masse... sortie qui ne peut rappelons-le supporter que quelques mA.



Le câblage de gauche est celui implanté sur la carte (montage dit pull-down) :

- lorsque le bouton est relâché, le port 2 est relié à la masse via la résistance de 10 k Ω ;
- lorsqu'il est appuyé, il se retrouve lié au +5 V

La logique est « inversée » dans le montage de droite (montage dit pull-up) ou, cette fois, le port est à l'état + 5V lorsque le bouton n'est pas appuyé.

b Bouton poussoir en mode marche/arrêt

On aimerait bien utiliser le bouton poussoir comme une bascule marche/arrêt. Dans le cas précédent, l'état du bouton poussoir n'était pas conservé entre deux parcours de la boucle loop. *A priori*, il suffit de conserver l'état de la LED dans une variable déclarée à l'extérieur de la boucle loop et de modifier cet état lorsque l'on détecte un appui sur le bouton poussoir.

Le programme LED_BoutonMarcheArretEchec implémenterait une telle approche :

Listing 2.8 – Programme LED_BoutonMarcheArretEchec

```
1  #define redLEDPin    6
2  #define buttonPin    2
3  boolean ledState = LOW ;
4
5  void setup()
6  {
7    pinMode(redLEDPin, OUTPUT);
8    pinMode(buttonPin, INPUT) ;
9  }
10
11 void loop()
12 {
13   int button = digitalRead(buttonPin) ;
14   if (button == HIGH)
15   {
16     ledState = !ledState ;
17     digitalWrite(redLEDPin, ledState) ;
18   }
19 }
```

L'algorithme semble correct.

- La LED est initialement éteinte.
- Si on appuie sur le bouton, *button* passe à HIGH, on change l'état de la LED qui passe à HIGH (l'utilisation d'une variable booléenne permet de simplifier quelque peu le code par rapport aux listings 2.3 à 2.6).
- Lorsqu'on lâche le bouton, l'état de la LED n'est plus modifié.

- Lorsqu'on appuie à nouveau sur le bouton, on change à nouveau l'état de la LED qui s'éteint.

Mais... c'est l'enfer ! « Des fois ça marche et des fois ça ne marche pas ! ».
En fait c'est normal ! Notre doigt reste appuyé un certain temps sur le bouton. A chaque parcours de la boucle, la LED change d'état (sans conséquence sur son éclat à cause de la persistance rétinienne) et, au final, on conserve l'état qu'avait la LED la dernière fois que la boucle était parcourue avec le bouton appuyé.

Nous avons mal programmé notre système ; ce qu'il faut détecter ce n'est pas le fait que le bouton soit appuyé à un moment donné mais qu'il passe d'un état non appuyé à un état appuyé. Le code suivant implémente cette approche.

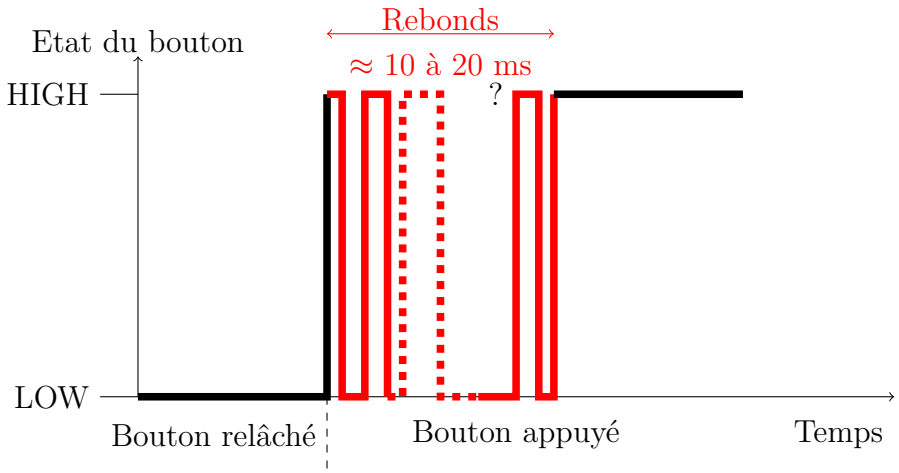
Listing 2.9 – Programme LED_BoutonMarcheArret

```
1  #define redLEDPin    6
2  #define buttonPin    2
3  boolean ledState = LOW ;
4  boolean lastButton = LOW ;
5
6  void setup()
7  {
8    pinMode(redLEDPin, OUTPUT);
9    pinMode(buttonPin, INPUT) ;
10 }
11
12 void loop()
13 {
14   boolean button = digitalRead(buttonPin) ;
15   if (button != lastButton)
16   {
17     if (button == HIGH)
18     {
19       ledState = ! ledState ;
20       digitalWrite(redLEDPin, ledState) ;
21     }
22   }
23   lastButton = button ;
24 }
```

Analysons ce programme :

- L'état du bouton poussoir est stocké dans la variable booléenne *lastButton*.
- A chaque parcours de la boucle, cette variable est actualisée (ligne 23).
- Si on appuie sur la bouton, la variable *button* passe à vrai, devient différente de *lastButton* et on atteint alors la ligne 17.
- Mais, cette ligne sera également atteinte lorsqu'on relâche le bouton poussoir puisqu'on aura alors *button* = faux et *lastButton* = vrai. Il faut tester si c'est effectivement un appui sur le bouton que l'on a fait.
- Dans ce cas, et uniquement dans ce cas, il faut changer l'état de la LED (ligne 19) et l'affichage correspondant (ligne 20).

Exécutez ce programme. Il se peut très bien que tout se passe comme souhaité. On peut aussi avoir un comportement erratique. Tout dépend en fait de la qualité du bouton poussoir !
C'est un problème bien connu des électroniciens, lorsque l'on appuie sur le bouton poussoir, le contact n'est pas immédiat. Des « rebonds » se produisent durant une vingtaine de millisecondes et... l'état logique du bouton est alors indéfini.



Lors du rebond, à chaque fois que l'état va passer de bas à haut, les lignes 19 et 20 seront exécutées et l'état de la LED inversée. Au final, on repère un état haut mais... qu'en était-il la fois d'avant ? Quel était l'état de la LED au point ? du graphe :

- si la LED était éteinte, on va l'allumer ! Victoire.
- si la LED était allumée, on va l'éteindre... d'où un fonctionnement pour le moins aléatoire de notre système.

Une solution matérielle est bien connue, il suffit de brancher un condensateur en parrallèle du bouton poussoir. Ainsi, toutes les rapides fluctuations de tension dues aux rebonds seront filtrées.

Quelle solution logicielle proposer ? Plusieurs solutions sont possibles mais, en gros, il faudrait « oublier » tout ce qui se passe pendant les rebonds. Par exemple, on peut repérer le premier front montant, on attend 20 ms et on vérifie que l'état du bouton poussoir est toujours haut. Dans ce cas, on change l'état de la LED.

La boucle du programme pourrait alors correspondre au code suivant. Les lignes 6 et 7 rajoutées par rapport au programme précédent permettent de régler ce problème.

Listing 2.10 – Programme LED_BoutonMarcheArret avec anti rebond

```
1  void loop()
2  {
3    boolean button = digitalRead(buttonPin) ;
4    if (button != lastButton)
5    {
6      delay(20) ;
7      button = digitalRead(buttonPin) ;
8      if (button == HIGH)
9      {
10       ledState = ! ledState ;
11       digitalWrite(redLEDPin, ledState) ;
12     }
13   }
14   lastButton = button ;
15 }
```

Exercice 2 Au secours ! Réaliser un programme qui envoie un SOS lumineux lorsque l'on appuie sur le bouton poussoir.
Damned, votre sauveur a raté le début du message. On complique l'exercice... on souhaite réémettre

le SOS 5 secondes plus tard tant que l'on n'a pas appuyé à nouveau sur le bouton poussoir (l'appui doit se faire quand la lampe est éteinte).

3.2 Un potentiomètre pour faire varier l'intensité

Cette approche sera abordée dans le chapitre suivant. Il nous faudra, en effet effectuer une lecture analogique d'une tension sur un des ports de l'Arduino.

4 Commander la LED... version Software

4.1 Communication via le moniteur série

Nous avions vu dans le premier chapitre qu'il était possible de communiquer avec la carte Arduino à l'aide du moniteur série. Utilisons là, ici, pour envoyer des ordres à l'Arduino.

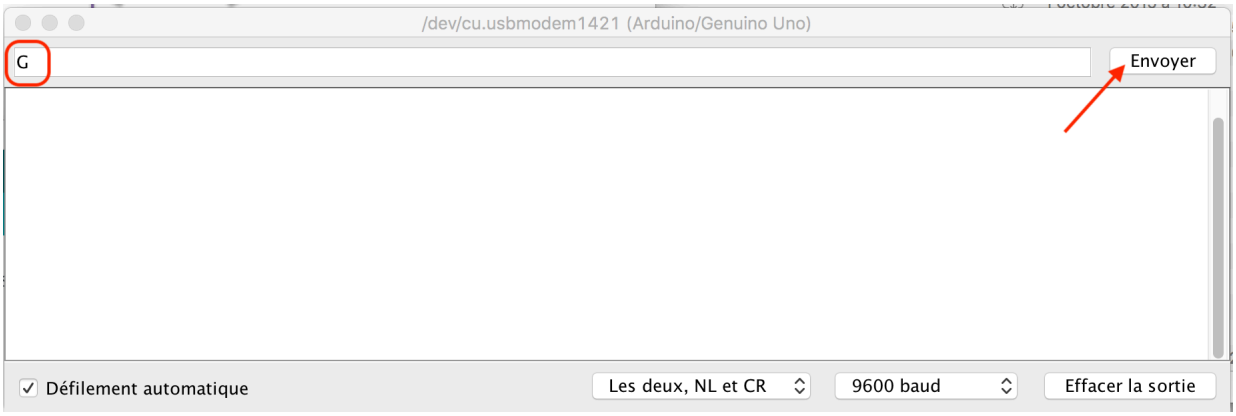
a Définition d'un protocole

Pour communiquer... il faut un langage commun !
Chacun est libre de proposer son propre protocole de communication. Nous allons, ici, utiliser par la suite les conventions suivantes (sans espace entre l'ordre et la valeur).

Ordre envoyé	Signification
G	GO : on exécute en continu
S	STOP : on arrête
O	ONE : on exécute une seule fois le travail souhaité
I	INFO : on récupère une information sur le programme
D : valeur	DELAI : on fixe le délai entre 2 actions

b Côté moniteur série

C'est très simple, il suffit d'introduire l'ordre (G par exemple) dans la zone de saisie de texte et cliquer sur **Envoyer**.



Pour exécuter la tâche souhaitée toutes les 100 ms, par exemple, on enverra `D:100`.

c Côté Arduino !

Le plus dur reste à faire mais... le jeu en vaut la chandelle, cette approche étant utilisée quasi systématiquement par la suite.
N'ayez pas peur... on peut très bien s'en sortir uniquement avec le premier paragraphe !

c.1 Interpréter les ordres GO/STOP

Le programme LED_EditeurGOSTOP propose une solution possible.

Listing 2.11 – Programme LED_EditeurGOSTOP

```
1  #define redLEDPin 6
2
3  void setup()
4  {
5    Serial.begin(9600) ;
6    pinMode(redLEDPin, OUTPUT);
7  }
8
9  void loop()
10 {
11   if (Serial.available())
12   {
13     String message = Serial.readString() ;
14     message.replace("\r", "") ;
15     message.replace("\n", "") ;
16     if (message == "G")
17     {
18       digitalWrite(redLEDPin, HIGH);
19     }
20     else if (message == "S")
21     {
22       digitalWrite(redLEDPin, LOW);
23     }
24   }
25 }
```

Analysons ce programme :

- ligne 11 : on détecte si un ou plusieurs caractères sont accessibles sur le port série (envoie de données de l'ordinateur vers l'Arduino) ;
- ligne 13 : on stocke dans la chaine de caractère (type **String** en C++) la chaîne de caractères envoyée ;
- lignes 14 et 15 : permettent de supprimer les éventuels caractères retour à la ligne et nouvelle ligne ajouté éventuellement à la fin (cf choix du menu déroulant en bas du moniteur série) ;
- lignes 16 et suivantes : on réalise les différents tests et on agit en conséquence.

Attention, ligne 16 et 20 à la syntaxe particulière du test d'une égalité entre deux termes : l'instruction à utiliser est `==` et non `=` qui correspond à une affectation.

Ce programme peut servir de modèle très simple pour passer des ordres à l'Arduino. Il suffit d'ajouter éventuellement d'autres ordres et de modifier les tests en conséquences. Nous proposons, par la suite, quelques « améliorations »... au prix d'un code plus complexe à analyser. En particulier, au paragraphe suivant, nous allons modifier le code afin d'interpréter une valeur numérique.

Exercice 3 Info sur le programme. Ajouter un test pour l'ordre I. Une fois cet ordre interprété, l'Arduino affiche sur le port série une information sur le programme en cours d'exécution. C'est une bonne pratique pour s'assurer que l'on a bien téléchargé le bon programme sur le microcontrôleur.

c.2 Interpréter une valeur numérique On souhaite maintenant passer un ordre du type R :100 ; le programme sur l'Arduino doit interpréter cet ordre et modifier l'éclairement de la LED rouge en conséquence. Lorsqu'on éteint et rallume la lampe, on doit retrouver le même éclairement. C'est l'objet du programme : LED_Editeur

Listing 2.12 – Programme LED_Editeur

```
#define redLEDPin 6
#define Info "Programme LED_Editeur"
int redValue = 255 ;

void setup()
{
  Serial.begin(9600) ;
  pinMode(redLEDPin, OUTPUT);
}

void loop()
{
  if (Serial.available())
  {
    String message = Serial.readString() ;
    message.replace("\r", "") ;
    message.replace("\n", "") ;
    message.replace(":", "") ;
    String ordre = String(message[0]);
    message.replace(ordre, "") ;
    int valeur = 0 ;
    if (message.length() > 0)
      valeur = message.toInt() ;
    if (ordre == "G")
    {
      analogWrite(redLEDPin, redValue);
    }
    else if (ordre == "S")
    {
      digitalWrite(redLEDPin, LOW);
    }
    else if (ordre == "I")
    {
      Serial.println(Info) ;
    }
    else if (ordre == "R")
    {
      redValue = valeur ;
      analogWrite(redLEDPin, redValue);
    }
  }
}
```

Il nous faut cette fois extraire d'une part le premier caractère qui correspond à l'ordre à interpréter et d'autre part l'éventuelle valeur passée après les :. Nous proposons, ici, une version relativement simple mais qui fait toutefois appel à des connaissances sur la gestion des chaînes de caractères en C.

Une simple instruction du type analogWrite(redLEDPin, valeur) ne peut suffir. En effet, la valeur du paramètre doit être sauvegardée afin d'être exploiter lorsqu'on passe un ordre G. C'est le rôle de la variable *redValue*.
Pour faciliter l'utilisation de ce programme, nous proposons une version légèrement modifiée :

- l'interprétation du message se fait dans une fonction dédiée **parse** qu'il suffira de modifier. Comme il y aura toujours les instructions I, G et S, autant écrire le code correspondant dans une fonction dédiée
- au lieu d'avoir une succession de if, on utilise l'instruction **switch** du langage C qui permet une compréhension peut-être plus aisée du programme. Il « suffira » par la suite d'écrire les bons **case**, le code correspondant et terminer par break.

Voici donc le programme :

Listing 2.13 – Programme LED_Editeur2

```
#define redLEDPin 6
#define Info "Programme LED_Editeur2"
byte redValue = 255 ;

void setup()
{
  Serial.begin(9600) ;
  pinMode(redLEDPin, OUTPUT);
}

void loop()
{
  if (Serial.available())
  {
    String message = Serial.readString() ;
    parse(message) ;
  }
}

void parse(String msg)
{
    msg.replace("\r", "") ;
    msg.replace("\n", "") ;
    msg.replace(":", "") ;
    char ordre = msg[0];
    msg.replace(String(ordre), "") ;
    int valeur = 0 ;
    if (msg.length() > 0)
      valeur = msg.toInt() ;
    switch(ordre)
    {
      case 'G' :
        go() ;
        break ;
      case 'S' :
        stop() ;
```

```
37          break ;
38        case 'I' :
39          info() ;
40          break ;
41        case 'R' :
42          redValue = valeur ;
43          go() ;
44          break ;
45      }
46  }
47
48  void info()
49  {
50    Serial.println(Info) ;
51  }
52
53  void go()
54  {
55    analogWrite(redLEDPin, redValue) ;
56  }
57
58  void stop()
59  {
60    digitalWrite(redLEDPin, LOW);
61  }
```

c.3 Sauvegarder l'état du système

Une fonctionnalité supplémentaire de la carte Arduino peut nous être utile afin de résoudre un problème : comment faire pour retrouver le même éclairement lorsque l'on reconnecte la carte ? Dans l'état actuel des choses, lorsque l'on modifie l'éclairement, celui-ci est conservé tant que la carte reste alimentée. Mais si on débranche la carte et si on la rebranche, on retrouve l'état initial correspondant à l'éclairement maximal. Il peut être intéressant, parfois, de conserver les réglages.

La carte Arduino dispose, pour se faire, d'une mémoire non volatile nommée EEPROM (Electrically-Erasable Programmable Read-Only Memory ou mémoire morte effaçable électriquement et programmable).

Listing 2.14 – Programme LED_Editeur3 (Extrait)

```
1  #include <EEPROM.h>
2
3  void setup()
4  {
5    Serial.begin(9600) ;
6    pinMode(redLEDPin, OUTPUT);
7    EEPROM.get(0, redValue);
8  }
9
10         case 'R' :
11           redValue = valeur ;
12           EEPROM.update(0, redValue) ;
13           go() ;
14           break ;
```

Afin de sauvegarder une variable (ici $redValue$), on modifie le programme LED_Editeur2 de la façon suivante :

- On importe la bibliothèque **EEPROM** (ligne 1).
- Lorsque l'on modifie la valeur de $redValue$, on stocke en mémoire la valeur correspondante à l'adresse 0 (ligne 11).
- Cette valeur peut-être récupérée lors de l'initialisation du système. Ligne 7, on lit dans la mémoire EEPROM à l'adresse 0 la valeur de $redValue$.
- Ainsi, lorsque l'on lance l'ordre G, la LED a un éclairement correspondant à la valeur de $redValue$ sauvegardée.

L'EEPROM a une taille de 1 Ko. Elle ne supporte qu'environ 100000 cycles d'effacement/écriture et ne doit donc pas être mise à jour dans une boucle pour stocker des variables temporaires. Plusieurs valeurs peuvent être stockées dans l'EEPROM, à condition de bien gérer les adresses mémoire.

c.4 Un programme modèle

Le programme précédent (ou LED_Editeur2) semble fonctionnel pour peu que l'on envoie des ordres du type allumer la LED, éteindre, obtenir telle ou telle intensité... si on voulait faire clignoter la LED, on pourrait passer et interpréter un paramètre *delai* mais, où placer le code permettant de changer l'état de la LED. Dans la boucle loop ?

Nous nous retrouvons dans une situation quasi similaire à celle décrite au paragraphe 2.1 19. La solution la plus élégante est d'utiliser un timer appelant une fonction **job**. On peut même rajouter un test, si on ne souhaite aucun délai (en passant comme paramètre une valeur nulle), on fait appel à cette fonction directement dans la boucle **loop** afin d'obtenir un fonctionnement en continu.

Le programme LED_Modele fonctionne ainsi.

Listing 2.15 – Programme LED_Modele

```
1  #include <MsTimer2.h>
2
3  // variables utilisées pour tous les programmes :
4  #define BAUD 9600  // vitesse d'échange pour le port série
5  #define Info "Programme LED_Modele"
6  long delai = 1000 ; // Delai en ms entre 2 appels de la
       fonction job
7  bool continu = false ; // à vrai si delai = 0
8
9  // variables spécifiques au système étudié
10 #define redLEDPin 6 // le port de la led rouge
11 byte redValue = 255 ; // éclairage maximum
12 bool ledState = false ; // vrai pour allumé
13
14 void setup()
15 {
16   Serial.begin(BAUD) ;
17   pinMode(redLEDPin, OUTPUT);
18   MsTimer2::set(delai, job) ;
19 }
20
21 void loop()
22 {
23   if (Serial.available())
```

```
24    {
25      String message = Serial.readString() ;
26      parse(message) ;
27    }
28    if (continu)
29    {
30      job() ;
31    }
32  }
33
34  void parse(String msg)
35  {
36    msg.replace("\r", "") ;
37    msg.replace("\n", "") ;
38    msg.replace(":", "") ;
39    char ordre = msg[0];
40    msg.replace(String(ordre), "") ;
41    int valeur = 0 ;
42    if (msg.length() > 0)
43      valeur = msg.toInt() ;
44    switch (ordre)
45    {
46      case 'I' :
47        info(Info) ;
48        break ;
49      case 'G' :
50        go() ;
51        break ;
52      case 'O' :
53        job() ;
54        break ;
55      case 'S' :
56        stop() ;
57        break ;
58      case 'D' :
59        if (valeur > 0)
60        {
61          delai = valeur ;
62          MsTimer2::set(delai, job) ;
63          continu = false ;
64        }
65        else
66        {
67          continu = true ;
68        }
69        break ;
70      default :
71        parse(ordre, valeur) ;
72        break ;
73    }
74  }
75
76  void info(String msg)
77  {
```

```
78    Serial.println(msg) ;
79  }
80  // traitements spécifiques au système
81
82  void go()
83  {
84    if (continu)
85    {
86      ledState = true ;
87    }
88    else
89    {
90      MsTimer2::start() ;  // c'est parti !
91    }
92  }
93
94  void stop()
95  {
96    MsTimer2::stop() ; // on stoppe le timer
97    ledState = false ; // on éteind la LED
98    digitalWrite(redLEDPin, LOW) ;
99  }
100
101 void parse(char ordre, long valeur)
102 {
103   switch (ordre)
104   {
105     case 'R' :
106       redValue = valeur ;
107       break ;
108   }
109 }
110
111 void job()
112 {
113   ledState = !ledState ;
114   if (ledState)
115   {
116     analogWrite(redLEDPin, redValue) ;
117   }
118   else
119   {
120     digitalWrite(redLEDPin, LOW) ;
121   }
122 }
```

Vous pouvez vous en servir comme modèle pour quasiment tous vos programmes. Il vous suffit de modifier les tests de la fonction **parse** (ligne 101 et suivantes) et le cœur de la fonction **job** (lignes 112 et suivantes).

4.2 Communication Bluetooth via un smartphone

Chargez sur l'Arduino le programme : LED_ModeleBT et connectez vous à la carte à l'aide du programme implémentant un moniteur série sur votre smartphone.
Une fois lancé, le programme vous propose de vous connecter. Sélectionnez « JDY-32-LE » ; la diode verte sur le module Bluetooth doit rester allumée. Vous vous trouvez alors avec un moniteur série. En bas de l'écran vous pouvez envoyer des ordres à l'Arduino et en haut de l'écran s'afficheront les messages reçus depuis l'Arduino.

Comment cela fonctionne-t-il ? La différence entre LED_Modele et LED_ModeleBT consiste en quelques lignes permettant de mettre en œuvre un second port série sur l'Arduino. Il s'agit d'un port série commandé en soft (mais c'est complètement transparent pour nous). À l'aide de ce second port série, on pourrait faire communiquer deux Arduino entre eux. Ici, on établit une liaison série via le module BlueTooth. La logique du code est, ceci étant, tout à fait semblable.

Listing 2.16 – Programme LED_ModeleBT (Extrait)

```
#include <SoftwareSerial.h>
...
#define softTX 4
#define softRX 5
SoftwareSerial BT(softRX, softTX) ;
...

void setup()
{
  Serial.begin(BAUD) ;
  BT.begin(BAUD) ;
  ...
}

void loop()
{
  if (Serial.available())
  {
    String message = Serial.readString() ;
    parse(message) ;
  }
  if (BT.available())
  {
    String message = BT.readString() ;
    parse(message) ;
  }
  if (continu)
  {
    job() ;
  }
}

void affiche(String msg)
{
  Serial.println(msg) ;
  BT.println(msg) ;
}
```

Que fait-on de plus ?

- Ligne 1 : on fait appel à la bibliothèque SoftwareSerial.
- Ligne 5 : on initialise ce port série que l'on nomme **BT** : les données seront reçues sur le port 4 de l'Arduino (RX) et transmises sur le port 5 (TX).
- Ligne 22 : après avoir testé l'arrivée de caractères sur le port série traditionnel, on fait de même sur le port série BT. L'interprétation se fait ensuite de la même façon.
- Ligne 36 : après avoir affiché les données sur le port série traditionnel, on fait de même sur le port BT.

4.3 Communication à partir d'un programme python

En fait, les deux exemples précédents montrent qu'il « suffit » d'établir une connexion série avec la carte pour pouvoir communiquer. Il existe, en python, une bibliothèque permettant de communiquer via le port série : la bibliothèque **serial**.
On suppose, par la suite que le programme LED_RGBBT ou LED_RGB est téléchargé sur l'Arduino.

a A partir de la console ou d'un programme python

a.1 Première tentative

Voici un programme qui allume successivement la led en rouge, en vert puis en bleu.

Listing 2.17 – Programme python envoiOrdre.py

```
1  import serial as serial
2  import serial.tools.list_ports as lp
3  import time as time
4
5  ports = lp.comports() # ports est une liste de listes
6  print([ports[i][0] for i in range(len(ports))]) # liste des
       ports
7
8  port = ports[len(ports)-1][0] # choix du bon port série
9  baud = 9600
10
11 ser = serial.Serial(port, baud) # ouverture du port série
12 time.sleep(2) # une petite pause de 2 secondes !
13 print("Connexion à "+port+" à la vitesse : "+str(baud)+"
       BAUDS")
14
15 ser.write(bytes("G", "UTF-8"))
16 time.sleep(1.2)
17 ser.write(bytes("R:255", "UTF-8"))
18 print("Rouge")
19 time.sleep(10)
20 ser.write(bytes("E", "UTF-8"))
21 time.sleep(1.2)
22 ser.write(bytes("V:255", "UTF-8"))
23 print("Vert")
24 time.sleep(10)
25 ser.write(bytes("E", "UTF-8"))
26 time.sleep(1.2)
27 ser.write(bytes("B:255", "UTF-8"))
```

```
28 print("Bleu")
```

Ce programme fonctionne :

- les lignes 5 à 8 permettent de sélectionner le port série (à modifier éventuellement compte tenu de la liste des périphériques connectés sur les ports USB) et, ligne 9, on précise la vitesse de communication ;
- ligne 11 on établit une connexion avec la carte Arduino ; la variable *ser* est un objet (de type **serial**) nous permettant de communiquer avec l'Arduino ;
- pour envoyer un ordre à l'arduino, on utilise la syntaxe (quelque peu complexe) `ser.write(bytes(ordre, "UTF-8"))` où *ordre* est une chaîne de caractère ;
- mais... on ne peut envoyer 2 ordres de suite ; il faut une pause (1,2 seconde ici) !

Une unique connexion série est autorisée. Si la fenêtre du moniteur série du logiciel Arduino n'est pas fermée, une erreur se produira lors de l'exécution de la ligne 9. Le message d'erreur qui apparait alors dans la console est reproduit ci-dessous.

```
SerialException: [Errno 16] could not open port /dev/
cu.usbmodem1421: [Errno 16] Resource busy: '/dev/cu.usbmodem1421'
```

Ce n'est donc pas si compliqué que ça d'envoyer des ordres à la carte Arduino à partir d'un programme écrit en python (ou d'instruction dans la console) ; par contre... recevoir des informations de la carte est une tâche beaucoup plus complexe ! Par exemple, si on envoie l'instruction `ser.write(bytes("I", "UTF-8"))` pour avoir des informations sur la carte en cours d'exécution il faut attendre en retour un certain nombre de caractères. Attendre mais... combien de temps ? et, pendant ce temps, l'ordinateur est bloqué ! Dans ce cas de figure, on pourrait toutefois s'en sortir car bloquer le système 1 ou 2 secondes n'est pas préjudiciable. Mais, dans d'autres cas de figure, l'attente de la fin d'une tâche pourrait être beaucoup plus longue et on aimerait éventuellement pouvoir interrompre cette tâche. Pour ce faire, il faudrait implémenter du « multithreading ». Tout se passe alors comme si le programme pouvait exécuter deux processus en même temps : l'un permettant ici d'envoyer des instructions à l'Arduino via le port série ; l'autre scrutant le port série afin de récupérer les informations envoyées par l'Arduino. Ce type de programmation n'est pas évident à concevoir. Nous mettons donc à votre disposition une classe nommée Arduino qui implémente une connexion série avec l'Arduino et l'envoi/réception de messages. On dispose ainsi des mêmes instructions que celles présentes dans un moniteur série.

a.2 Utilisation de la bibliothèque Arduino mise à votre disposition Si vous avez installé le dossier **ArdTools** à l'emplacement spécifié (cf **??** page **??**), vous pourrez accéder à la bibliothèque Arduino à partir de n'importe quel programme python sur votre ordinateur.

Le programme python suivant permet de se connecter à l'Arduino et envoyer un ordre.

Listing 2.18 – Programme python testBibliothequeArduino.py

```
1  import ArdTools.ArdTools as ard
2  import time
3
4  ser = ard.Arduino()
5
6  ports = ser.ports()
7  print(ports)
8
9  ser.connexion(ports[len(ports)-1], 9600)
10
```

```
11 ser.envoyer("E")
12 ser.envoyer("D:100")
13 ser.envoyer("B:50")
14 ser.envoyer("G")
15
16 time.sleep(10)
17
18 ser.envoyer("D:0")
19 for i in range(5) :
20     ser.envoyer("E")
21     ser.envoyer("R:255")
22     time.sleep(2)
23     ser.envoyer("E")
24     ser.envoyer("V:255")
25     time.sleep(1)
26     ser.envoyer("E")
27     ser.envoyer("B:255")
28     time.sleep(3)
29
30 ser.envoyer("V:255")
31 ser.envoyer("R:255")
```

Comment utiliser cette bibliothèque ? On se reportera également au paragraphe 2.5 page 104.

- Ligne 1 : on importe la bibliothèque sous l'alias `ard`.
- Ligne 2 : comme dans le programme précédent, on instancie un objet *ser* de type `Arduino` permettant de communiquer avec le port série.
- Lignes 6 et 7 : on récupère dans la liste *ports* la liste des ports séries ouverts sur votre ordinateur et on les affiche dans la console.
- Ligne 9 : on établit une connexion avec le bon port série. Un message d'erreur est affiché dans la console en cas de problème de connexion (le plus souvent soit parce que le port série est déjà utilisé par le moniteur de l'IDE d'Arduino). Dans le cas contraire vous pouvez envoyer des ordres à l'arduino soit dans un programme comme sur l'exemple ci-dessus, soit à partir de la console.
- L'importation de la bibliothèque `Time` permet de faire une pause dans le programme python grâce à l'instruction `time.sleep(valeur)` ; *valeur* est en seconde.

b A partir d'une interface dédiée

L'utilisation de **PyQT** nous permet de concevoir une interface utilisateur. La tâche n'est pas simple... nous vous fournissons quelques exemples dans ce guide. Les « experts » pourront se baser sur le code que nous proposons pour construire leur propre interface.

b.1 Un moniteur série Nous allons utiliser une interface appelée PyToArdQt : elle permet d'envoyer des ordres du PC vers l'Arduino, et de recevoir des informations provenant de l'Arduino à afficher sur le PC. C'est une sorte de moniteur série, avec quelques fonctionnalités supplémentaires. Chacune des applications PyQt que nous vous proposons est présentée de la même manière :

- un dossier ayant le nom de l'application
- dans ce dossier : un fichier .py ayant le nom de l'application
- un fichier .pyw ayant le nom de l'application

- d'autres éléments utiles pour le fonctionnement de l'application. Pour un pur utilisateur, elles ne vous intéresseront pas.

Pour lancer l'application, vous avez deux possibilités : soit vous ouvrez le script *PyToArdQt.py* avec l'ide de votre choix (par exemple Idle), puis vous exécutez le script (pour Idle, appuyez sur F5), soit vous double cliquez sur *PyToArdQt.pyw* : le script se lance automatiquement et l'interface s'affiche.
Une fois l'interface lancée, vérifiez que :

- la carte Arduino est bien branchée à l'ordinateur
- le moniteur série d'Arduino est bien fermé : la liaison série ne peut être utilisée que par un seul protocole à la fois (soit le moniteur série Arduino, soit Python, mais pas les deux !)
- vous avez bien repéré le numéro de port com de votre carte Arduino (dans le logiciel Arduino Outils/Ports) et la vitesse de communication.

Vous êtes prêt à utiliser l'interface !
En cliquant sur *Vérifier ports*, la carte doit apparaître, sélectionnez la dans le menu déroulant, puis choisir la bonne vitesse de connexion. Il ne reste plus qu'à cliquer sur *Connexion*. Patientez quelques instants le temps que la connexion soit établie. Le bouton doit maintenant être vert, avec *Connecté* écrit dessus.
Pour envoyer une commande, c'est simple : écrivez la commande dans le champ intitulé *Commande à envoyer*, puis cliquez sur *Envoyez*. Si un retour de l'arduino est attendu, il s'affichera dans le champ intitulé *Informations venant de l'Arduino*. Vous pouvez à tout moment effacer ces informations, ou les enregistrer. L'enregistrement ne tient compte que des données affichées dans le champ des informations provenant de l'Arduino, sous la forme d'un fichier *csv*. Le répertoire par défaut est celui de l'interface, mais vous pouvez le changer à votre souhait. Vérifiez tout de même que vous avez bien les droits d'écrire dans le répertoire de sauvegarde choisi.
Pour que l'affichage des informations reçues fonctionne correctement, voici quelques règles (peut contraignantes) à respecter du côté de votre programme Arduino :

- Chaque informations que vous voulez envoyer du côté du PC doit se terminer par un retour à la ligne : en effet, l'interface PyToArdQt, surveille le port série toutes les 10 millisecondes. Si des données sont reçues, elle attendra de recevoir le retour à la ligne avant de les afficher dans le champ d'informations. Donc : utilisez, du côté Arduino, *Serial.println("...")* et non *Serial.print("...")*
- si vous prévoyez d'envoyer périodiquement des informations vers l'ordinateur, veillez à ce que l'envoi ne soit pas "trop" rapide. Il n'est pas conseillé de descendre en dessous de 10 millisecondes entre deux envois. Ces performances dépendent de la taille de la chaîne de caractère à envoyer, de la vitesse de connexion de la liaison série et des performances de votre machine (notamment les performances de votre ports USB en lecture/écriture).
  Cette interface simple est très générique, elle conviendra pour un certain nombre de vos programmes.

b.2 Une interface dédiée Nous vous proposons une interface dédiée à la commande d'une LED de type RGB : LEDRGBQt. Son l'arborescence de dossier qui la contient est similaire à celle de PyToArdQt. Démarrer l'application en lançant *LEDRGBQt.py* ou *LEDRGBQt.pyw*. Connectez votre carte en choisissant le bon port et la bonne vitesse de connexion.
Dans la zone *Pilotage LED*, nous vous proposons plusieurs possibilités pour commander votre LED :

- un widget proposant différentes couleurs de base avec une palette de couleurs : cette partie est intéressante car en maintenant clic gauche appuyé et en parcourant la palette, vous voyez automatiquement les niveaux RGB s'ajuster. Ces niveaux sont aussi réglables manuellement en rentrant les niveaux RGB souhaités (partie inférieure à la palette. Enfin vous avez la

possibilité d'utiliser la fonction *Pick Screen Color* qui joue le rôle d'une pipette venant prélever une couleur où vous voulez sur votre écran (sur une photo par exemple).

- trois sliders que nous avons ajouté pour régler chaque niveau indépendamment.

Suivant votre action, la commande à envoyer à l'Arduino s'actualise pour vous informer de la future couleur demandée. Évidemment, vous ne pourrez pas reproduire avec cette simple LED toutes les couleurs demandées, mais cette interface vous laisse entrevoir toutes les possibilités pédagogiques s'offrant à vous !
Une fois votre couleur choisie, cliquez sur le bouton envoyer. Vous allez voir progressivement le niveau rouge, puis vert et enfin bleu s'activer. L'opération prend quelques secondes pour s'exécuter avant de pouvoir lancer une autre commande.

5 Conclusion

Ne sous-estimons pas le travail réalisé. Certes, le Diable se cache dans les détails, mais... à quelques « détails » près,si on sait commander une LED, on sait commander n'importe quoi.

Les problèmes que l'on peut rencontrer sont principalement de deux types :

- on ne peut pas espérer, avec la platine Arduino dont nous disposons, commander des systèmes à une fréquence supérieure à 100-1000 Hz.
- les sorties de l'Arduino fonctionne entre 0 et 5V et ne peuvent délivrer plus de 20 mA (200 mA en tout) ; ce qui INTERDIT toute alimentation de système type moteur.... Un montage électronique sera donc nécessaire pour utiliser un circuit « de puissance ». Ce montage sort des objectifs de ce kit de formation mais sera bientôt proposé par ailleurs !

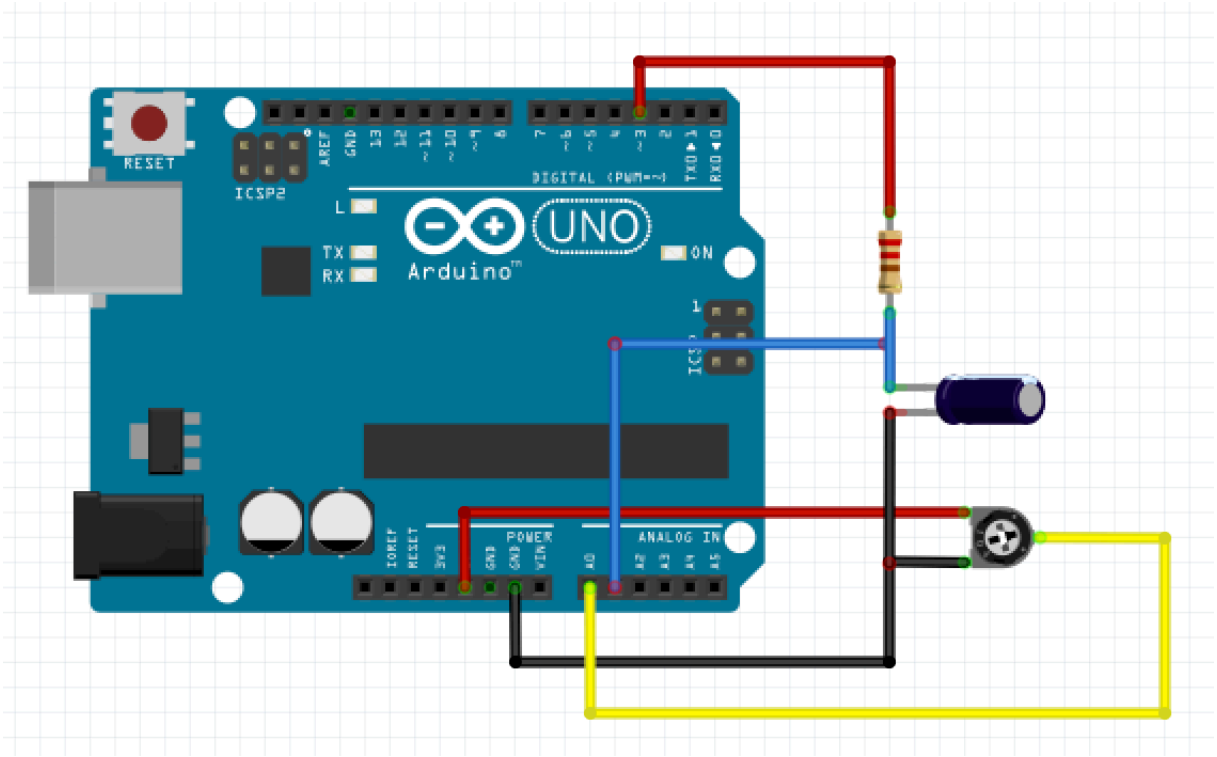
Activité 3

Lecture analogique

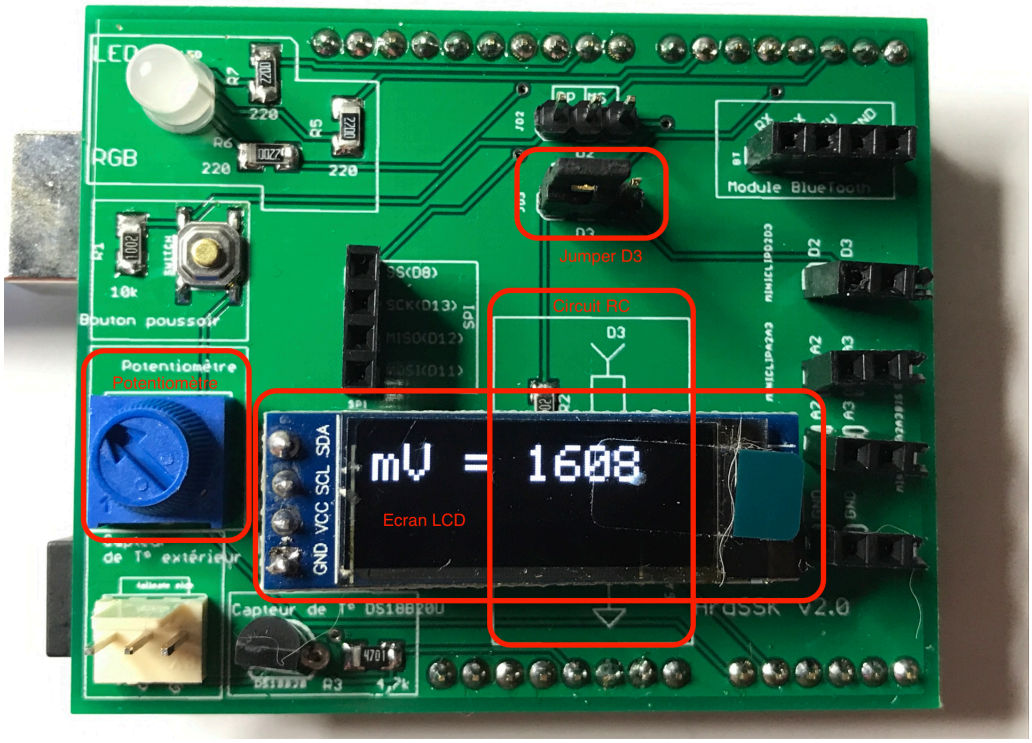
La platine Arduino Uno dont vous disposez comporte 6 ports dédiés à une lecture analogique (nommés A0, A1… () A5). Il s'agit de convertisseurs analogique-numérique (ADC) sur 10 bits. $2^{10} = 1023$. On peut lire une tension, en entrée, comprise entre 0 et 5 V, ce qui donne une résolution de 5 mV. La conversion demande 100 microsecondes. Pour aller au-delà, il faudra, soit connecter un convertisseur d'une résolution plus importante (16 bits par exemple), soit plus rapide, soit… utiliser une carte Arduino plus puissante (et plus chère !).
Deux circuits sont câblés sur la carte et seront exploités dans ce chapitre :

- un diviseur de tension (potentiomètre rotatif de 10 kΩ) alimenté entre 0 et 5 V et dont le point milieu est connecté à l'entrée analogique **A0** ;
- un circuit RC alimenté par le port **D3** et dont le point milieu est connecté à l'entrée analogique **A1** R = 10 kΩ et C = 10 μF.

Le schéma du circuit, utile dans ce chapitre est donné ci-dessous.



Il faudra penser à connecter le cavalier comme sur la figure ci-dessous afin de pouvoir commander le circuit RC. Vous utilisez également l'écran LCD pour afficher, dans un des exemples d'application, vos résultats.



1 analogRead... bien sûr !

On s'en serait douté, après **digitalWrite** (pour imposer une valeur 0/5V sur un port), **digitalRead** (pour lire une valeur 0/5V sur un port), **analogWrite** (pour imposer une valeur entre 0 et 5 V) dans le chapitre précédent... on veut lire, ici, une valeur comprise entre 0 et 5 V sur un port, en toute logique, c'est l'instruction **analogRead** qu'il faut utiliser !

1.1 Lecture sur un port analogique

Le programme minimal ADC_Minimum se contente de lire toutes les 100 ms une donnée sur le port **A0** et l'afficher dans le moniteur série.

Listing 3.1 – Programme ADC_Minimum

```
1  #define analogPin A0
2
3  void setup()
4  {
5    Serial.begin(9600);
6  }
7
8  void loop()
9  {
10   int sensorValue = analogRead(analogPin);
11   Serial.print("sensor = ");
12   Serial.println(sensorValue);
13   delay(100);
14 }
```

Exercice 1 Éclairage variable. C'était l'objectif attendu au paragraphe 3.2 page 27. Moduler l'intensité de la LED à l'aide du potentiomètre rotatif. Attention, analogRead retourne un entier

compris entre 0 et 1023 (10 bits) alors que analogWrite attend un entier compris entre 0 et 255 (8 bits). Faites la conversion, ou utilisez la fonction **valeur = map(entier,min1,max1,min2,max2)** qui convertit un entier compris entre $min1$ et $max1$ en un entier $valeur$ compris entre $min2$ et $max2$.

1.2 Sortie sur écran LCD

Connectez l'écran LCD sur la platine d'essai (cet écran affiche 128×32 pixels). Exécutez le programme ADC_LCD. Cette fois la valeur de la tension lue sur la broche **A0** s'affiche à l'écran.

Listing 3.2 – Programme ADC_LCD

```
1  #include <Adafruit_GFX.h>
2  #include <Adafruit_SSD1306.h>
3
4  #define analogPin A0
5  #define OLED_RESET 4
6  Adafruit_SSD1306 lcd(OLED_RESET);
7
8  void setup()
9  {
10   lcd.begin(SSD1306_SWITCHCAPVCC, 0x3C);
11   lcd.setTextSize(2);
12   lcd.setTextColor(WHITE);
13   lcd.clearDisplay() ;
14   lcd.display();
15 }
16
17 void loop()
18 {
19   int sensorValue = analogRead(analogPin);
20   int ddp = map(sensorValue, 0, 1023, 0, 5000) ;
21   lcd.clearDisplay();
22   lcd.setCursor(0,0);
23   lcd.print("mV = ") ;
24   lcd.println(ddp) ;
25   lcd.display() ;
26   delay(100) ;
27 }
```

Nous utilisons, pour ce faire, une bibliothèque externe. On se contente, ici, de recopier le code de l'exemple fournit avec la bibliothèque. On instancie un objet **lcd** ligne 6. L'initialisation se fait dans la fonction `setup`. La valeur anologique est lue (ligne 19) et on effectue une conversion entre 0 et 5000 mV. L'affichage est ensuite mis à jour. L'instruction **setCursor** (ligne 22) permet de préciser le coin supérieur gauche du premier caractère (on aurait écrit (0,16) pour obtenir le texte sur la seconde ligne).

1.3 Mon premier objet...

a Mon premier objet embarqué

Exercice 2 Un millivoltmètre autonome. Exploitez tout ou partie des codes fournis dans des programmes précédents pour réaliser un voltmètre respectant le cahier des charges suivant.

- Au début, l'écran LCD affiche « Appuyez sur le bouton ! ».

- Quand on appuie sur le bouton poussoir, la différence de potentiel lue sur le port **A0** s'affiche sur l'écran.
- Quand on appuie à nouveau sur le bouton poussoir, le premier message s'affiche à nouveau.
- Si on dispose d'une pile 9V, le système doit fonctionner sans connexion à l'ordinateur.

b Mon premier object connecté

Exercice 3 Un millivoltmètre connecté. Exploitez tout ou partie des codes fournis dans des programmes précédents pour réaliser un voltmètre respectant le cahier des charges suivant.

- On se connecte à la carte via le programme moniteur série sur son smartphone.
- On commande l'acquisition à partir de son smartphone, on peut modifier le délai entre 2 mesures (si vous avez un petit peu de mal, considérez que le délai ne peut pas être nul (on ne gère alors pas l'acquisition en continu).
- L'affichage de la valeur doit se faire sur l'écran du smartphone.
- Si on dispose d'une pile 9V, le système doit fonctionner sans connexion à l'ordinateur.

2 Sortie formatée

Mais... finalement, nous disposons maintenant d'un système d'acquisition (et en plus, nous savons le contrôler !).

Il nous reste deux étapes, beaucoup plus faciles, à franchir :

- dans la suite de ce chapitre : formater les données afin de pouvoir les exploiter graphiquement ;
- dans les chapitres 4 et 5 : voir comment adapter un capteur « du commerce » à notre système : température, pression, lumière, champ magnétique, accélération... un « jeu d'enfants » si on a compris comment utiliser les bouts de code dont on dispose pour concevoir la trame de notre système d'acquisition.

2.1 Choisir un protocole pour le println !

Nous nous retrouvons quasiment face au même problème que celui rencontré paragraphe a page 27 : choisir un formatage des données. Chacun est libre de choisir le sien. Nous avons opté ici pour le format suivant (similaire à celui utilisé pour l'envoie d'instruction à l'Arduino) :
DATA :id1 :valeur1 :id2 :valeur2...

- chaque information est séparée par **:** ;
- on commence par **DATA** ;
- id est une chaine de caractère pour identifier la donnée : TIME, T0, U, I...
- valeur est... la valeur correspondante !

Le programme ADC_DATA met en œuvre ce protocole. On modifie le programme ADC_Minimum en envoyant les données sur le port série avec le format requis.

Listing 3.3 – Programme ADC_DATA

```
1 #define analogPin A0
2
3 void setup()
4 {
5   Serial.begin(9600);
```

```
6  }
7
8  void loop()
9  {
10   int sensorValue = analogRead(analogPin);
11   String msg = "DATA" ;
12   msg = msg + ":TIME:" + String(millis()) ;
13   msg = msg + ":A0:" + String(sensorValue) ;
14   Serial.println(msg) ;
15   delay(100);
16 }
```

C'est tout bête en fait, mais extrêmement fonctionnel comme nous allons le voir.
Les instructions lignes 11 à 13 permettent de formater les données à notre guise. L'instruction `String(nombre)` permet de convertir ce nombre en chaîne de caractères.

2.2 Un programme modèle

Le programme précédent est fonctionnel mais rudimentaire.
Dans un cas plus général, on aimerait bien pouvoir commander l'acquisition et que celle-ci ne débute effectivement qu'à partir du moment où on envoie l'instruction 'G'. Il faudrait également que l'origine des temps corresponde à cet instant.
Le programme LED_Modele (ou VoltmetreContinu_Connecte) peut nous servir de base de travail. Si vous avez résolu l'exercice « millivoltmètre connecté » avec acquisition continue (cf programme VoltmetreContinu_Connecte page 43), il y a juste quelques instructions à modifier ; sinon la solution consiste à reprendre le programme LED_ModeleBT (page 32) et de remplacer tout ce qui concerne la LED par la lecture analogique.

Nous remettons ici le programme ADC_Modele dans son intégralité afin d'insister sur l'importance d'un modèle efficace que l'on peut adapter à ses besoins : on fait un copier/coller du fichier et on change ce qui nous intéresse.

L'ajout d'une variable *start* mise à 0 lors de l'exécution de la fonction `go()` permet d'avoir le tempsé écoulé depuis le début de l'acquisition grâce à l'instruction millis()-start.

Listing 3.4 – Programme ADC_Modele

```
1  #include <MsTimer2.h>
2  #include <SoftwareSerial.h>
3
4  // variables utilisées pour tous les programmes :
5  #define BAUD 9600  // vitesse d'échange pour le port série
6  #define Info "ADC_Modele"
7  #define softTX 4
8  #define softRX 5
9  SoftwareSerial BT(softRX, softTX) ;
10 unsigned long start = 0 ;
11 long delai = 1000 ; // Delai en ms entre 2 appels de la
       fonction job
12 bool continu = false ; // à vrai si delai = 0
13 bool doJob = false ;
14 // variables spécifiques au système étudié
15 #define analogPin   A0
16
```

```
17 void setup()
18 {
19   Serial.begin(BAUD) ;
20   BT.begin(BAUD) ;
21   MsTimer2::set(delai, job) ;
22 }
23
24 void loop()
25 {
26   if (Serial.available())
27   {
28     String message = Serial.readString() ;
29     parse(message) ;
30   }
31   if (BT.available())
32   {
33     String message = BT.readString() ;
34     parse(message) ;
35   }
36   if (continu && doJob)
37   {
38     job() ;
39   }
40 }
41
42 void parse(String msg)
43 {
44   msg.replace("\r", "") ;
45   msg.replace("\n", "") ;
46   msg.replace(":", "") ;
47   char ordre = msg[0];
48   msg.replace(String(ordre), "") ;
49   int valeur = 0 ;
50   if (msg.length() > 0)
51     valeur = msg.toInt() ;
52   switch (ordre)
53   {
54     case 'I' :
55       affiche(Info) ;
56       break ;
57     case 'G' :
58       go() ;
59       break ;
60     case 'O' :
61       job() ;
62       break ;
63     case 'S' :
64       stop() ;
65       break ;
66     case 'D' :
67       if (valeur > 0)
68       {
69         delai = valeur ;
70         MsTimer2::set(delai, job) ;
```

```
71          continu = false ;
72        }
73        else
74        {
75          continu = true ;
76        }
77        break ;
78      default :
79        parse(ordre, valeur) ;
80        break ;
81    }
82  }
83
84  void affiche(String msg)
85  {
86    Serial.println(msg) ;
87    BT.println(msg) ;
88  }
89  // traitements spécifiques au système
90
91  void go()
92  {
93    if (continu)
94    {
95      doJob = true ;
96    }
97    else
98    {
99      MsTimer2::start() ;  // c'est parti !
100   }
101   start = millis() ;
102   job() ;
103 }
104
105 void stop()
106 {
107   MsTimer2::stop() ; // on stoppe le timer
108   doJob = false ; // on éteind la LED
109 }
110
111 void parse(char ordre, long valeur)
112 {
113   switch (ordre)
114   {
115     default :
116       break ;
117   }
118 }
119
120 void job()
121 {
122   int sensorValue = analogRead(analogPin);
123   String msg = "DATA" ;
124   msg = msg + ":TIME:" + String(millis()-start) ;
```

```
125    msg = msg + ":A0:" + String(sensorValue) ;
126    affiche(msg) ;
127  }
```

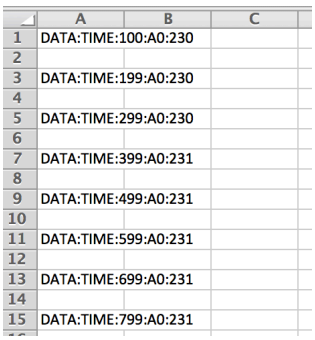
Qu'aurons nous à faire pour adapter ce programme à nos besoins ?

- Compléter les lignes 1 et 2 par l'importation éventuelle d'autres bibliothèques.
- Définir ligne 15 et suivantes les différents ports utilisés sur l'arduino, instancier différents objets associés aux capteurs également.
- Compléter la fonction `parse(char ordre, long valeur)` ligne 111 et suivantes si l'on doit passer d'autres paramètres ou informations au système.
- Et enfin, modifier la fonction **job**.
- Et c'est tout !

Ce programme utilise MSTimer2 ; de ce fait, on ne pourra pas utiliser les sorties **3** et textbf11 comme sortie PWM. Afin, par exemple, de pouvoir étudier le circuit RC par la suite, un programme ADC_ModelNoTimer est proposé.

2.3 Copier dans Excel

Faire une acquisition sur quelques secondes en tournant le bouton du potentiomètre rotatif.

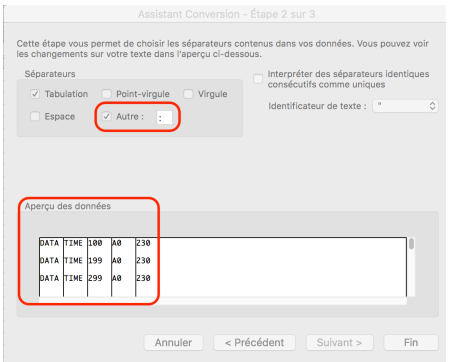


	A	B	C
1	DATA:TIME:100:A0:230		
2			
3	DATA:TIME:199:A0:230		
4			
5	DATA:TIME:299:A0:230		
6			
7	DATA:TIME:399:A0:231		
8			
9	DATA:TIME:499:A0:231		
10			
11	DATA:TIME:599:A0:231		
12			
13	DATA:TIME:699:A0:231		
14			
15	DATA:TIME:799:A0:231		

Faire un copier coller de l'intégralité des données (ou de celles qui nous intéressent) du moniteur série d'Arduino vers la première case d'une feuille Excel.

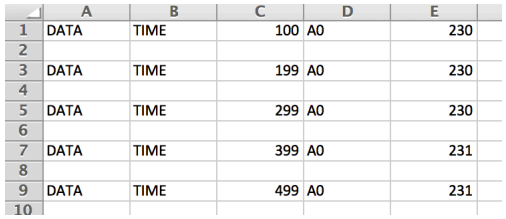
Les données sont maintenant dans la première colonne, éventuellement séparées d'une ligne blanche.

Sélectionnez la première colonne puis, dans le menu **Données** d'Excel, cliquez sur l'item **Convertir...**.



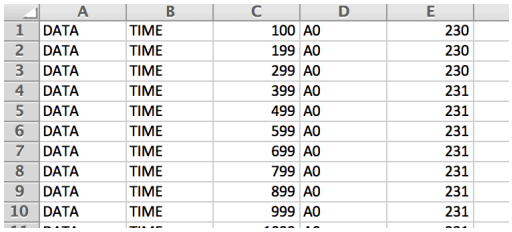
Une boite de dialogue s'ouvre, passez à la seconde page et préciser que le séparateur est **:** (ou autre si vous avez choisi un autre format de sortie).

Les données apparaissent alors dans chacune des premières colonnes.



	A	B	C	D	E
1	DATA	TIME	100	A0	230
2					
3	DATA	TIME	199	A0	230
4					
5	DATA	TIME	299	A0	230
6					
7	DATA	TIME	399	A0	231
8					
9	DATA	TIME	499	A0	231
10					

Sélectionnez l'ensemble des données et cliquez sur le bouton **Trier** .

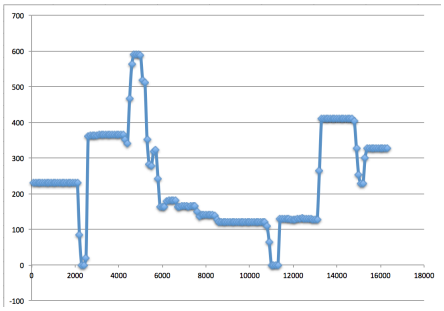


	A	B	C	D	E
1	DATA	TIME	100	A0	230
2	DATA	TIME	199	A0	230
3	DATA	TIME	299	A0	230
4	DATA	TIME	399	A0	231
5	DATA	TIME	499	A0	231
6	DATA	TIME	599	A0	231
7	DATA	TIME	699	A0	231
8	DATA	TIME	799	A0	231
9	DATA	TIME	899	A0	231
10	DATA	TIME	999	A0	231

Nous avons ainsi nos données prêtes à être traitées.

Si l'on a des nombres réels, il faudra penser à convertir les . en , ! .

Reste alors à faire la représentation graphique...



2.4 Exploiter les données en python

On peut très bien également copier les données du moniteur série et les sauvegarder dans un fichier texte (data.txt par exemple) .

Le programme acquisition.py permet de lire les données dans un fichier texte que l'on vient de créer, de les interpréter et de tracer la courbe.

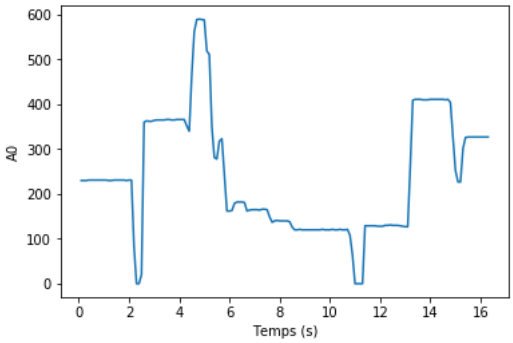
Listing 3.5 – Programme acquisition.py

```
1  import matplotlib.pyplot as plt
2
3  fichier = open('data.txt')
4  lignes = fichier.readlines()
5  fichier.close()
6
7  Temps=[] # temps en s
8  A0=[]    # analogRead sur port A0
9
10 for ligne in lignes :
11     ls = ligne.split(':') # on découpe la ligne
12     if len(ls) >= 5 :
13         Temps.append(float(ls[2])/1000) ;
14         A0.append(int(ls[4]))
15
16 plt.plot(Temps, A0)
17 plt.xlabel('Temps (s)')
18 plt.ylabel('A0')
19 plt.show()
```

Ce programme n'est absolument pas générique mais permet d'être facilement adaptable :

- On ouvre le fichier data.txt et on extrait dans la variables *lignes* l'ensemble des lignes du fichier. Cette variable est une liste de chaines de caractères. Le premier élément de la liste correspond à 'DATA :TIME :100 :A0 :230'.
- On crée 2 listes vides pour stocker les données.
- L'instruction clé est la ligne 11 qui permet de couper chacune des lignes à l'occurence du caractère **:**. On crée ainsi une variable *ls* qui est à nouveau une liste de chaines de caractère. Pour la première ligne *ls* correspond à ['DATA','TIME','100','A0','230'].
- l'élement d'indice 2 correspond à la valeur du temps, celui d'indice 4 à la lecture sur A0 ; reste à convertir la chaine de caractère en réel ou entier et stocker les valeurs.

On dispose alors de deux tableaux de valeurs que l'on peut exploiter en python.



Et si on voulait tracer la tension plutôt que la lecture du port analogique. Deux solutions s'offre à nous :

- faire la conversion dans le programme de l'Arduino (à l'aide de la fonction `map` par exemple) comme dans le programme ADC_Minimum ;
- faire la conversion dans le programme python.

Dans la mesure du possible, on s'arrange plutôt pour faire exécuter les calculs par l'ordinateur. L'Arduino ne dispose pas d'unité arithmétique, de ce fait, les calculs réalisés sur le microcontrôleur demande beaucoup de temps.

Pour éviter de retaper toujours les mêmes lignes dans le programme (ou pour des élèves, ou professeurs, débutants en python), on peut grandement simplifier le programme en utilisant une bibliothèque.
Stockons dans un fichier **util.py** la fonction suivante :

Listing 3.6 – Fonction de lecture des données dans le fichier util.py

```
1  def lecture2Data(nomFichier) : # lecture de 2 données X, Y
2      fichier = open(nomFichier)
3      lignes = fichier.readlines()
4      fichier.close()
5      X=[]
6      Y=[]     # analogRead sur port A0
7      for ligne in lignes :
8          ls = ligne.split(':') # on découpe la ligne
9          if len(ls) >= 5 :
10             X.append(float(ls[2])/1000) ;
11             Y.append(int(ls[4]))
12     return X,Y
```

Dès lors, pour peu que le fichier `util.py` soit dans le même dossier que le programme **acquisition2.py**, l'extraction des données devient transparente pour l'utilisateur.

Listing 3.7 – Programme acquisition2.py

```
1  import matplotlib.pyplot as plt
2  import util as util
3
4  Temps, A0 = util.lecture2Data('data.txt')
5
6  plt.plot(Temps, A0)
7  plt.xlabel('Temps (s)')
8  plt.ylabel('A0')
9  plt.show()
```

3 Acquisition/exploitation à partir d'un programme python

A la fin du chapitre précédent, nous avons vu (paragraphe 4.3 page 36) comment envoyer des données à une carte Arduino soit à partir de la console soit en construisant une interface dédiée. *A priori*, on pourrait s'attendre à faire de même ici.

Pour nous simplifier la tâche, nous allons utiliser deux programmes python distincts :

- un premier programme chargé du contrôle de l'acquisition, la sauvegarde des données dans un fichier texte (éventuellement, cerise sur le gâteau, l'existence d'une fenêtre graphique permettant de suivre de façon dynamique l'évolution temporelle du système) ;
- un second programme chargé du traitement numérique du système étudié.

Le second programme, nous l'avons déjà puisque nous venons de voir dans le paragraphe précédent comment exploiter les données. Reste le premier...
Les difficultés ont été évoquées page 37. Comme dans le chapitre précédent, nous allons vous proposer deux programmes « clés en main » permettant de gérer l'acquisition.

3.1 A partir de la console

Dans le chapitre précédent nous avions uniquement des ordres à envoyer à la carte Arduino. L'utilisation de l'objet `Arduino` de la bibliothèque `ArdTools` était pertinente (cf paragraphe a.2 page 37). Cette fois, une fois l'ordre 'G' passé, on attend de nombreuses réponses. Deux solutions vous sont proposées. Dans le premier cas, les résultats sont affichés dans la console python, dans le second une représentation graphique des valeurs mesurées est dynamiquement mise à jour.

On suppose que le programme ADC_ModeleNoTimer est chargé sur la carte Arduino.

a Acquisition sans représentation graphique

La classe à utiliser est alors **ArduinoAcq**.

Listing 3.8 – Programme acquisitionArduinoAcq.py

```
1  import ArdTools.ArdTools as ard
2
3  a = ard.ArduinoAcq()
4  ports = a.ports()
5  a.connexion(ports[len(ports)-1], 9600)
6
7  a.sauvegarde("potar.txt")
8  a.envoyer("D:100")
9  a.go()
10
11 # taper a.stop() dans la console pour stopper l'acquisition
```

- On instancie, cette fois un objet de type `ArduinoAcq` (ligne 3).
- Toutes les fonctions de la classe `Arduino` sont accessibles. Le protocole de connexion est le même (lignes 4 et 5) tout comme la possibilité d'envoyer des ordres à la carte Arduino (lignes 8 et 9.
- Si on souhaite enregistrer les données dans le fichier, on le précise (ligne 7) avant de lancer l'acquisition ! Les données sont, dans le cas présent, sauvegardées dans un fichier texte dont le nom est spécifié dans le même dossier que votre programme. Ce fichier peut ensuite exploité par exemple par le programme acquisition2.py précédent.

- L'instruction `a.go()` (ligne 9, on aurait aussi pu écrire a.envoyer('G')) envoie effectivement l'ordre 'G' à la carte Arduino mais se charge, en plus, de gérer les données reçues de la carte via le port série. Ces données sont affichées dans la console python et éventuellement stockées dans un fichier.

On ne peut stopper l'acquisition que dans la console. Il faut y introduire l'instruction `a.stop()` ! Si on connait la durée de l'acquisition, on peut automatiser la tâche en faisant une pause dans le programme python ! Dans le programme ci-dessous, on fait une pause de 10 s et l'acquisition est automatiquement stoppée.

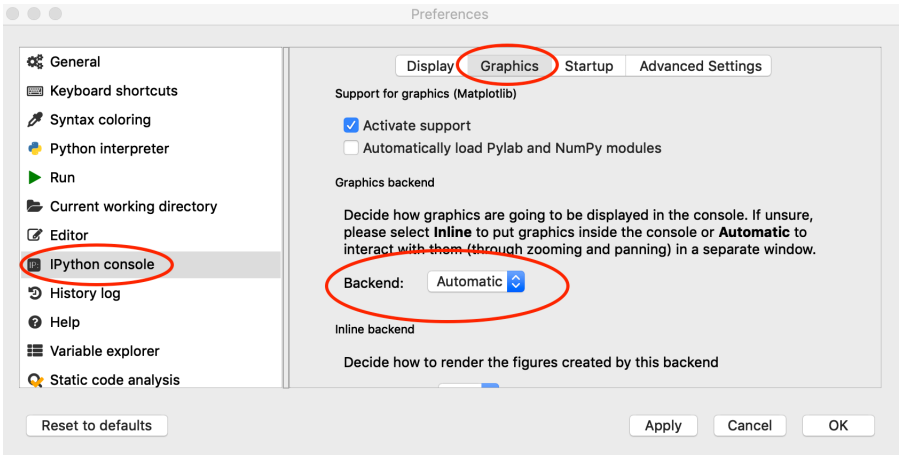
Listing 3.9 – Programme acquisitionArduinoAcq2.py

```
1  import ArdTools.ArdTools as ard
2  import time
3
4  a = ard.ArduinoAcq()
5  ports = a.ports()
6  a.connexion(ports[len(ports)-1], 9600)
7
8  a.sauvegarde("potar.txt")
9  a.envoyer("D:100")
10 a.go()
11
12 time.sleep(10)
13
14 a.stop()
```

b Acquisition avec représentation graphique dynamique

Cette fois, c'est la classe `ArduinoPlot` qu'il convient d'utiliser.

Si vous utilisez Spyder comme environnement de développement python, il faut préciser que vous ne souhaitez pas avoir les sorties graphiques dans la console !
Ouvrez le dialogue Préférences de Spyder :



Sélectionnez l'item **IPython console** dans la liste de gauche puis l'onglet **Graphics**. Le menu déroulant **Automatic** doit être sélectionné.

Le programme suivant, très similaire à acquisitionArduinoAcq.py permet de lancer l'acquisition.

Listing 3.10 – Programme acquisitionArduinoPlot.py

```
1 import ArdTools.ArdTools as ard
2
3 a = ard.ArduinoPlot()
4 ports = a.ports()
5 a.connexion(ports[len(ports)-1], 9600)
6
7 a.sauvegarde("potar.txt")
8 a.envoyer("D:100")
9 a.go()
```

Une fenêtre s'ouvre et l'évolution de la grandeur mesurée en fonction du temps apparaît. Dans le programme actuel, il faut faire tourner le bouton du potentiomètre rotatif.

La nouvelle fenêtre peut, parfois être cachée par la fenêtre principale !

L'acquisition ne peut s'arrêter qu'en tapant `a.stop()` ou `a.envoyer('S')` dans la console ! Vous ne pouvez pas utiliser `time.sleep(valeur)` ici.

Après avoir stoppé l'acquisition, une seconde fenêtre s'ouvre. L'évolution de la grandeur mesurée en fonction du temps est tracée. Un curseur permet de se déplacer sur l'écran pour repérer abscisse et ordonnée.

3.2 A partir d'une interface dédiée

En fait, devrions nous dire, deux interfaces ! Vous en avez déjà utilisé une : PytoArdQt qui remplace le moniteur série d'Arduino. Si vous désirez profiter d'un affichage dynamique, nous vous proposons TraceQt.

a Interface sans représentation graphique

- Démarrez l'interface PyToArdQt.
- Connectez votre carte.
- Envoyez un ordre pour le timer. Exemple : *D :100*.
- Envoyez simplement *G* pour lancer votre Acquisition.
- Les données reçues défilent dans la zone *Informations venant de l'Arduino*.
- Envoyez l'ordre *S* pour arrêter votre acquisition.
- Vous pouvez sauvegarder les données reçues, ou effacer le contenu du moniteur pour recommencer une acquisition.

Quelques recommandations :

- Ne soyez pas trop gourmand en période d'échantillonnage. Nous avons mené quelques essais et pour l'envoi de 6 mesures analogiques, il n'est pas conseillé de descendre en dessous de 10 ms. En dessous de cette valeur, vous risquez de saturer le port série et l'interface graphique également !
- Nous vous conseillons de bien cliquer sur *Déconnexion* avant chaque fermeture de l'application pour libérer proprement le port série en vue d'une utilisation via une autre application par exemple.

b Interface avec représentation graphique

Cette application vous permet de bénéficier d'un affichage dynamique pour votre acquisition, et d'un graphe synthétisant l'ensemble de votre acquisition une fois terminée. Démarrez *TraceQt.py* ou *TraceQt.pyw*
Cette fois-ci, elle semble un peu plus complexe que les autres interfaces, mais pas de panique : nous vous expliquons tout. Déjà, notons la présence de deux onglets :

- *Acquisition* : il contient tout ce que dont vous avez besoin pour configurer et lancer votre acquisition. Le tracé dynamique s'affiche dans cet onglet.
- *Graphique* : Bien plus simple, il contient l'ensemble de votre acquisition, une fois terminée.

Pour le moment, restez sur l'onglet *Acquisition* : vous reconnaissez déjà la zone habituelle de connexion : connectez donc votre carte. Rentrez une période d'échantillonnage, en millisecondes. Vous apercevez également un label *Rafraichissement du graphe*. Voici son fonctionnement :

- si vous rentrez une valeur strictement positive, par exemple 5 , le graphe dynamique se rafraichira toutes les 5 secondes. Cette option évite d'afficher trop de points à l'écran et garantit une certaine fluidité à l'acquisition.
- si vous rentrez 0, tout l'acquisition sera affichée, de l'instant initial de la mesure, jusqu'à ce que vous décidiez de l'arrêter : très pratique pour avoir l'ensemble de l'acquisition sous les yeux. Attention cependant aux éventuels ralentissements !

Une fois que vous avez choisi votre période d'échantillonnage et la période de rafraichissement du graphe, cliquer sur *Valider*. Si vous le souhaitez, vous pouvez envoyer une commande personnalisée, qui ne sera transférée que si vous appuyez sur le bouton *Envoyer*. **Attention, ne pas envoyer d'ordre où un retour vers le PC est attendu !**. Par exemple, imposer une couleur à une LED ou faire tourner un moteur est accepté, par contre, envoyer *I* fera planté l'application !
Pour sauvegarder votre acquisition vous avez deux choix :

- Choix 1 : Vous voulez enregistrer vos données au fur et à mesure, pendant l'acquisition. Cliquez sur le bouton *Préparer sauvegarde*, choisir un nom de fichier et un répertoire. Décochez *Sauvegarde différée*. Vous êtes prêt pour lancer une acquisition ! Attention toutefois, la sauvegarde en temps réel, i.e. en même temps que l'acquisition sauvegarde les données reçues dans le fichier stipulé au fur et à mesure que des nouvelles données sont reçues sur le port série. Pour les acquisition nécessitant une certaine rapidité, cela peut nuire aux performances !
- Choix 2 : Vous voulez enregistrer vos données une fois l'acquisition terminée. Comme précédemment, cliquez sur *Préparer Sauvegarde* mais laissez coché *Sauvegarde différée*. La sauvegarde ne se sera qu'une fois l'acquisition terminée, en cliquant sur *Sauvegarder*.

Une fois que vous avez configuré votre acquisition, vous pouvez cliquer sur ... *Acquisition* ! Le graphe déroulant apparaît. Une fois votre mesure terminée, cliquez sur *Stop*. Si vous avez choisi le mode différé, n'oubliez pas de sauvegarder !
Quand votre acquisition est terminée, vous pouvez utiliser le deuxième onglet intitulé *Graphique*. Vous y verrez la totalité de l'acquisition, sous la forme de deux graphiques interagissant ensemble pour zoomer sur une zone particulière par exemple. Le bouton *Sauvegarder* de cet onglet à strictement la même fonction que celui vu précédemment.

4 Conclusion : « Je suis le maître du monde ! »

Vos efforts pour arriver à ce point ne sont pas vains car, à quelques « détails » [1] près, vous savez tout faire.

Ainsi, par exemple, vous disposez dès maintenant d'un voltmètre numérique dont vous maitrisez toute la conception avec exploitation possible des données dans Excel ou à l'aide d'un programme python.

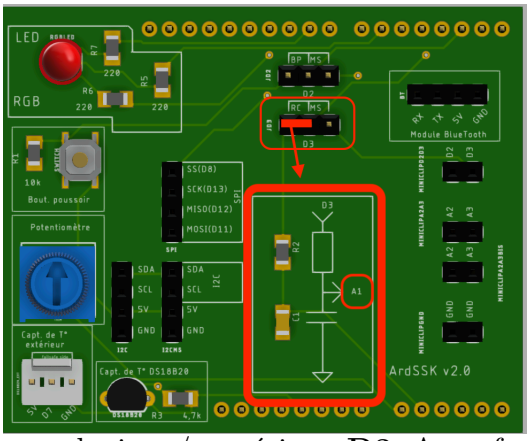
Vous savez :

- élaborer un programme pour contrôler les ports de sortie de l'Arduino ;
- élaborer un programme pour lire sur les ports d'entrée ;
- communiquer à partir du moniteur série ou d'un programme en python avec l'Arduino afin de lui envoyer des ordres et récupérer des données ;
- récupérer les données dans Excel ou dans un programme python pour un traitement ultérieur.

Ne soyez pas pessimistes si ce dernier point vous intéresse et vous semble, pour l'instant, insurmontable... python est un langage simple à appréhender si on cherche à faire des choses simples !
Pour preuve, nous vous proposons la mise en œuvre expérimentale d'un TP sur la charge ou la décharge d'un circuit RC, circuit déjà câblé sur la platine ArdSSK.

5 Un premier TP : étude et application d'un circuit RC

Nous allons utiliser, ici, le circuit électrique câblé sur la platine (un circuit RC alimenté par le port **D3** et dont le point milieu est connecté à l'entrée analogique **A1** comme reporté sur le schéma en tête de ce chapitre).



Pour cette partie, il faut mettre le cavalier D3 de la platine du kit vers la gauche, comme sur la figure ci-contre.

Nous exploitons ici l'entrée analogique **A1** et la sortie analogique/numérique **D3**. A un facteur multiplicatif près $\left(\frac{5000}{1023}\right)$, **A1** permet de lire la ddp (en mV) aux bornes du condensateur.
Supposons que l'on souhaite étudier la décharge du condensateur. Que souhaite-t-on faire ?

1. Imposer sur l'entrée **3** une tension de 5V et attendre (par exemple 2 ou 3 s) que le condensateur soit chargé.
2. A l'instant t=0, on applique une tension nulle sur l'entrée **3**.
3. On mesure la tension sur l'entrée **A1** et on envoie les informations sur le port série.
4. On arrête au bout d'un certain temps !

Côté Arduino, on sait programmer chacune de ces tâches individuellement :

1. digitalWrite(HIGH) et delay(3000) feront l'affaire ;

1. Il faut avoir conscience que notre système d'acquisition a ses limites, la première d'entre elle est sa fréquence d'échantillonage. Avec le matériel de base, on ne peut espérer dépasser le kHZ mais... est-ce vraiment le problème ?

2. digitalWrite(LOW) et start = millis() ;
3. valeur = analogRead(A1) et println('DATA :TIME :str(millis()-start)) :A1 :valeur') bien sûr ;
4. on peut, par exemple, s'arrêter quand l'entrée analogique devient inférieure à 10 (la tension atteint alors environ le centième de sa valeur initiale).

5.1 Une solution « simple »...

Le programe RC_Minimum propose une implémentation des 4 étapes précédentes.

Listing 3.11 – Programme RC_Minimum

```
1  #define analogPin    A1
2  #define commandePin 3
3  long start = 0 ;
4  int sensorValue = 0 ;
5
6  void setup()
7  {
8    Serial.begin(9600);
9    pinMode(commandePin, OUTPUT) ;
10   digitalWrite(commandePin, HIGH) ;
11   delay(3000) ;
12   sensorValue = analogRead(analogPin);
13   digitalWrite(commandePin, LOW) ;
14   start = millis() ;
15 }
16
17 void loop()
18 {
19   if (sensorValue > 10)
20   {
21     sensorValue = analogRead(analogPin);
22     String msg = "DATA" ;
23     msg = msg + ":TIME:" + String(millis() - start) ;
24     msg = msg + ":A0:" + String(sensorValue) ;
25     Serial.println(msg) ;
26   }
27 }
```

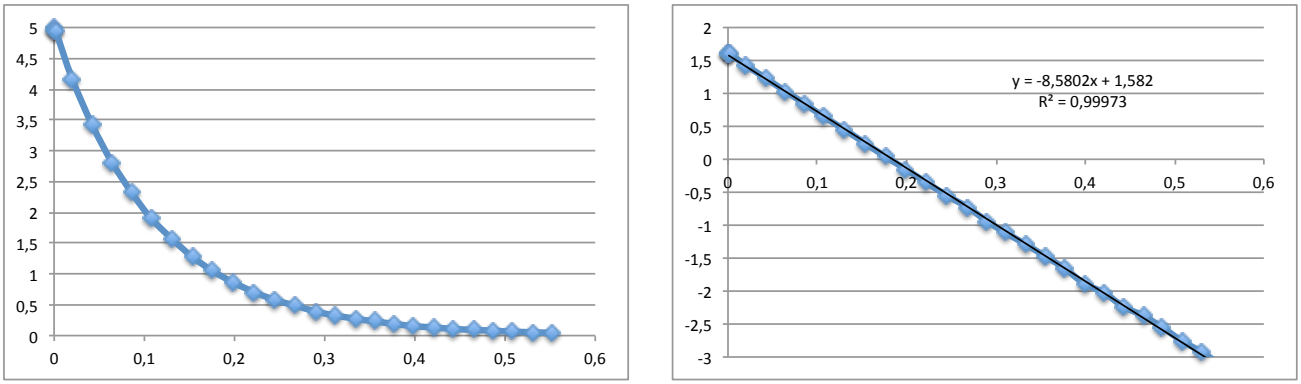
On va :

- Faire toutes les initialisations dans la fonction **setup**, y compris la charge du condensateur. Un premier appel d'`analogRead` permet d'initialiser la variable *sensorValue* à une valeur proche de 1023.
- Lignes 13 et 14, on impose 0 V sur le port **3** et on note le temps.
- Dans la boucle, si la valeur lue est supérieure à 10, on lit à nouveau la valeur et on envoie les résultats formatés sur le port série.

Ensuite, on récupère les données dans la console de l'IDE d'Arduino :

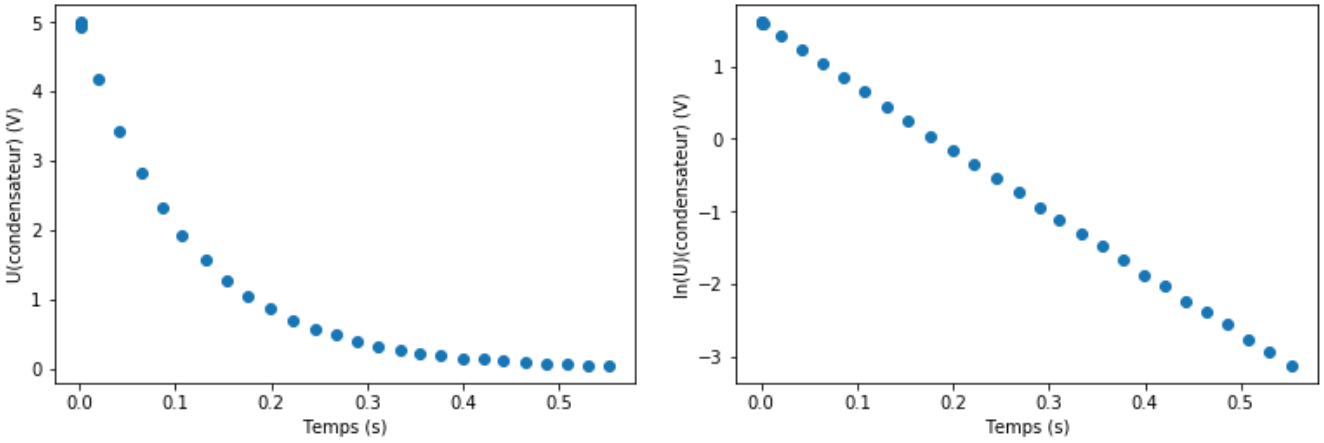
1. que l'on copie/colle dans Excel ;
2. ou que l'on copie dans un fichier texte 'dataRC.txt'.

Après, vous savez faire !



On trace (à gauche) U_C (en V) et fonction de t (en s), et à droite $\ln(U_C)$ et fonction de t. On obtient une droite de pente -8,58 avec un excellent coefficient de corrélation (0,99973) !
On en déduit $RC = 0,12s$, en accord avec les composants utilisés $R = 10k\Omega$ et $C = 10\mu F$ compte tenu de la tolérance de 10% du fabricant sur C en particulier.

Ou en python :



Pour traiter les données, nous avons ajouté dans le fichier **util.py** une routine de régression linéaire [2] (avec quelques variantes !).

Listing 3.12 – Programme acquisitionRC : acquisitionRC.py

```
1  import matplotlib.pyplot as plt
2  import util as util
3  from math import log
4
5  Temps, A0 = util.lecture2Data('dataRC.txt')
6  UC = []
7  lnUC = []
8  for a in A0 :
9      u = a*5/1023
10     UC.append(u)
11     lnUC.append(log(u))
12
13 plt.plot(Temps, UC, 'o')
14 plt.xlabel('Temps (s)')
15 plt.ylabel('U(condensateur) (V)')
16 plt.show()
17
18 plt.plot(Temps, lnUC, 'o')
```


2. L'explication de la mise en oeuvre de ces routines est faite dans le chapitre Régression linéaire du document sur les méthodes numériques en python

```
19 plt.xlabel('Temps (s)')
21 plt.ylabel('ln(U)(condensateur) (V)')
22 plt.show()
23
24 a,b,r = util.regrelin(Temps, lnUC)
25 print("a = ",a,"b = ",b,"r^2 = ",r*r)
26
27 print("RC = ", round(-1/a, 2), "s")
```

5.2 Bien sur, on peut commander à partir du moniteur série

Exercice 4 Temps de demi décharge
Modifier la fonction `job` du programme ADC_ModeleNoTimer (ainsi que l'attribution des ports) pour déterminer, lors d'une décharge, le temps au bout duquel la tension aux bornes du condensateur atteint la moitié de la valeur initiale. Afficher la valeur et comparer au résultat précédent. On rappelle qu'envoyer un ordre **'O'** à l'Arduino permet de n'appeler qu'une seule fois la fonction `job`.

Exercice 5 Étude en fonction de la tension de charge
La broche **3** qui permet d'alimenter le circuit RC que l'on étudie est une sortie PWM [3] (cf page 17). Modifier le programme de l'exercice précédent afin d'afficher dans le moniteur série tension et temps de demi décharge pour une dizaine de valeurs de la tension. Pour chaque valeur de la tension, on pourra éventuellement faire la moyenne de 10 mesures du temps de demi décharge.

Exercice 6 Générateur de tension variable
On souhaite réaliser un générateur de tension à peu près constante (à 10 % près) sans utiliser la fonction PWM. On prend comme modèle le programme ADC_Modele ou ADC_ModeleNoTimer. Le programme doit :

- interpréter l'ordre `'T:valeur'` ou *valeur* est une tension en mV, inférieure à 5000 bien sûr !
- délivrer sur le port **A1** une tension la plus constante possible voisine de *valeur* ;
- afficher la valeur lue sur **A1** afin d'en faire une représentation graphique de l'évolution temporelle.

5.3 Ou en python...

Tout a été abordé dans la première partie de ce chapitre.

a A partir de la console

Exercice 7 En guise de révision !
Faire une acquisition sans représentation graphique (cf a page 51) de la décharge d'un condensateur. Stocker les valeurs dans un fichier et exploiter les à l'aide du programme acquisitionRC.py.

Exercice 8 En guise de challenge !
Modifiez le programme réalisé dans l'exercice Générateur de tension ci-dessus afin de concevoir un générateur de signaux triangulaires (sans trop se soucier de la valeur précise de la fréquence souhaitée dans un premier temps). Validez votre système à l'aide d'une acquisition avec représentation graphique (cf paragraphe b page 52).

3. Comme précisé plus haut, la sortie **3** n'est pas compatible avec l'utilisation d'un timer, c'est pourquoi il faut partir du progamme modèle ADC_ModeleNoTimer !

b A partir d'une interface dédiée

... mais, rêve-je ? N'ai-je pas sous les yeux un programme qui simule un oscillo numérique (pour une fréquence inférieure à 100 Hz, il faut rester modeste !).

6 Pour gagner quelques millisecondes !

a Jouer sur le débit

Prenons, par exemple une sortie formatée : `DATA:TIME:1666:A1:360` de l'exercice générateur de tension. Celle-ci compte 22 caractères (avec le caractère de fin de ligne). Le protocole de communication nécessite, par caractère 8 bits de données (code ASCII) et 1 bit de start et 1 bit de stop. Environ 220 bits sont nécessaires.

À un débit de 9600 bauds, il faut donc environ 20 ms. On comprend mieux le relatif échec de notre générateur de tension (cf page 123).
Passer à débit de 115200 bauds (par exemple) diviserait, naturellement ce temps par 12 ! Soit un temps de transfert d'1 à 2 ms qui ne serait plus forcément le processus limitant.
Un petit effort et pour une application dédiée seule l'information /verbatim1666 :360 suffirait. On peut ainsi espérer un temps de communication sur le port série de l'ordre de la milliseconde.

Alors, pourquoi ne pas mettre systématiquement 115200 bauds voire plus (le moniteur série propose 2 millions de bauds !) ?
Lorsque le débit augmente, le signal se déforme, la détection des fronts montant ou descendant devient de plus en plus difficile et des erreurs de communication peuvent intervenir. 9600 bauds nous permet d'assurer une bonne communication dans pratiquement tous les cas de figure. Libre à vous, ensuite d'augmenter ce débit pour des applications nécessitant une fréquence de traitement plus importante. La plupart des programmes de test dans les bibliothèques que nous aborderons par la suite utilisent également ce débit de 9600 bauds.

b Changer le protocole de communication

En réalité, l'instruction `Serial.println('bla bla bla')` n'est pas la solution la plus rapide pour communiquer entre l'Arduino et l'ordinateur. Les utilisateurs expérimentés préfèreront l'instruction `Serial.write(valeur)` permettant un transfert octet par octet. Nous n'en dirons pas plus ici, mais la solution existe !

c Changer de carte Arduino, ou de microcontrôleur...

La carte que vous utilisez est la carte standard : **Arduino Uno**. Son processeur est cadencé à 16 MHz. Mais il en existe d'autres qui sont compatibles et se programment exactement de la même façon (il suffit de changer le type de carte dans l'IDE d'Arduino). Par exemple les cartes **Teensy** qui peuvent atteindre 72 et même 180 MHz sont des alternatives possibles.

Au prix d'efforts importants dans l'apprentissage d'un nouvel environnement de développement, on peut même tenter l'aventure des pics, ou STM32 ou autre...

Quels qu'ils soient, ces cartes restent des microcontrôleurs... certaines permettent de travailler à l'échelle du méga Hertz !

Activité 4

Réalisation pratique d'un projet

Nous allons voir, ici, sur un exemple simple à mettre en oeuvre quelles étapes permettent de passer d'un projet « Mesurer une température » à sa réalisation pratique.

La première idée est de taper « **Arduino** température » sur Google... plus d'un million de résultats apparaissent !

Parions que sur un sujet aussi classique, d'innombrables solutions tant matérielles que logicielles peuvent convenir. On pourrait se focaliser sur certains sites, « Carnet du Maker » par exemple. Mais, sans y avoir aucune action, il y a un site que nous pouvons que vous recommander : c'est **Adafruit** : www.adafruit.com. Vous y trouverez d'innombrables capteurs, actionneurs, systèmes... et, avantage ÉNORME, pour chacun d'eux (ou presque) vous disposerez :

- d'une description du capteur ;
- de son câblage ;
- et surtout d'une bibliothèque fiable pour son utilisation... immense, immense avantage !
- d'exemples d'applications clé en main.

Il vous restera :

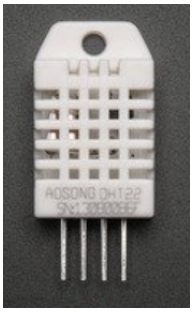
- à acheter le capteur (Mouser, par exemple commercialise les produits Adafruit en France, et quelques grossistes chinois en propose des clones), câbler, installer les bibliothèques, tester et... le moment venu recopier le code nécessaire dans vos programmes fétiches développés aux chapitres précédents.
- ou... regarder sur notre site si nous ne proposons pas un kit d'utilisation dudit capteur avec le capteur et un guide pédagogique d'utilisation. Petit à petit, nous espérons pouvoir vous proposer quelques solutions pour les systèmes les plus « classiques ».

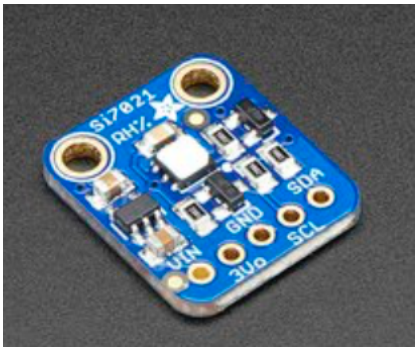
Analysons les étapes les unes après les autres. C'est, en fait, le chapitre qu'un diffuseur de matériel ne devrait pas publier !

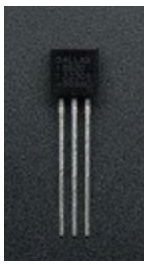
1 Mise en oeuvre d'un capteur de température

1.1 Commençons par Adafruit

Allons voir sur le site **Adafruit.com** et effectuons une recherche sur **temperature**. Parcourons les quelques premières pages...

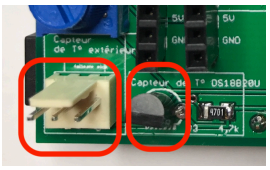







Cliquons sur le lien du premier (un capteur température et humidité : DHT22 ou son petit frère DHT11 que l'on trouve dans quasiment tous les kits). Pas mal mais impossible de plonger ce capteur dans une solution pour une étude de calorimétrie par exemple.
Il en va de même pour le second. Cette fois on nous propose un capteur sur une carte électronique... peut-être une solution, un jour pour concevoir un système plus complexe ! On voit écrit SCL, SDA ; c'est donc un module qui communique par I2C, il « suffit » d'avoir la bibliothèque et c'est extrêmement simple d'utilisation (cf notre afficheur LCD !).
Aïe, pour le troisième, va falloir sortir son fer à souder. D'un autre côté, ça ne coute quasiment rien et on pourrait en fixer quelques-uns sur une plaque métallique pour étudier la diffusion de la chaleur par exemple. Son petit nom est DS18B20.
Oh, pas mal le quatrième. DS18B20 comme le précédent et étanche. On nous dit que ce n'est pas le plus simple à utiliser mais que la bibliothèque est fournie, que l'on peut en connecter plusieurs sur le même port et qu'il a une précision de 0,5 degré c'est pas mal. Ce sera donc notre choix !

En tout cas, ce fût le nôtre et deux capteurs peuvent être utilisés sur notre platine d'essai. Le premier est soudé et un connecteur est disponible pour le second.



1.2 La bibliothèque DallasTemperature

Cette bibliothèque a été installée sur votre ordinateur lors de la copie des bibliothèques (cf **??** page **??**).
Prenons le premier exemple directement dans l'IDE d'Arduino :

- sélectionner l'item **Exemple** du menu **Fichier** de l'IDE d'Arduino ;
- cliquer ensuite **DallasTemperature** vers le bas du sous menu déroulant qui vous est proposé ;
- sélectionner enfin l'exemple **Simple** dans le nouveau menu.

Le programme se charge alors et apparait dans l'éditeur.
Il faut changer l'identificateur du port ONE_WIRE_BUS : pour la carte configurée, l'identificateur est **7** (et non 2 !) (ligne 6).

Téléversez le programme sur la carte et ouvrez le moniteur série. Mettez la main sur le capteur, ça marche !

Le programme est le suivant :

Listing 4.1 – Programme Simple de la bibliothèque Dallas

```
1 // Include the libraries we need
2 #include <OneWire.h>
3 #include <DallasTemperature.h>
```

```
4
5  // Data wire is plugged into port 2 on the Arduino
6  #define ONE_WIRE_BUS 7
7
8  // Setup a oneWire instance to communicate with any OneWire
       devices (not just Maxim/Dallas temperature ICs)
9  OneWire oneWire(ONE_WIRE_BUS);
10
11 // Pass our oneWire reference to Dallas Temperature.
12 DallasTemperature sensors(&oneWire);
13
14 /*
15    The setup function. We only start the sensors here
16 */
17 void setup(void)
18 {
19   // start serial port
20   Serial.begin(9600);
21   Serial.println("Dallas Temperature IC Control Library Demo");
22
23   // Start up the library
24   sensors.begin();
25 }
26
27 /*
28    Main function, get and show the temperature
29 */
30 void loop(void)
31 {
32   // call sensors.requestTemperatures() to issue a global
       temperature
33   // request to all devices on the bus
34   Serial.print("Requesting temperatures...");
35   sensors.requestTemperatures(); // Send the command to get
       temperatures
36   Serial.println("DONE");
37   // After we got the temperatures, we can print them here.
38   // We use the function ByIndex, and as an example get the
       temperature from the first sensor only.
39   Serial.print("Temperature for the device 1 (index 0) is: ");
40   Serial.println(sensors.getTempCByIndex(0));
41 }
```

Que doit-on retenir de ce programme pour l'exploiter par la suite ?

- les lignes 2 et 3 pour l'importation des bibliothèques ;
- les lignes 9 et 11 pour l'instanciation du bus OneWire et du (ou des éventuels) capteur(s) sur ce bus ;
- ligne 24 l'initialisation du (ou des) capteur(s) ;
- ligne 35 on attend la lecture d'une température ;
- ligne 40 ou lit la température (la première, et la seule ici).

Exercice 1 Afficher la température. Concevoir un système embarqué (autonome) qui affiche sur l'écran LCD la température donnée par le capteur soudé sur le module ArdSSK

Exercice 2 Capteur de température BlueTooth Concevoir un système connecté permettant d'accéder à la température donnée par le capteur soudé sur le module ArdSSK.Tester dans votre frigo ou votre congélateur (pas dans le four !).

Exercice 3 Une LED pour indiquer s'il fait chaud ou froid. Se fixer deux températures limites $TMin$ et $TMax$. Si la température est inférieure à $T1$ la LED RGB s'éclaire en bleu, si elle est comprise entre $T1$ et $T2$ en vert et si elle est supérieure à $T2$ en rouge. Faire varier la température avec la main et observer.

Exercice 4 Rupture de la chaine du froid. Se fixer une température limite $T1$. Le programme, sur la carte **Arduino**, doit indiquer sur l'écran LCD le temps (en seconde) passé au-dessus de la température $T1$ ainsi que la température maximale atteinte. Faire varier la température avec la main plusieurs fois et observer.

2 Un TP de calorimétrie

2.1 Vérifions notre système d'acquisition

a Côté Arduino

Reprendre le programme ADC_Modele, en faire une copie en le renommant, par exemple, Temperature_Modele. Remplacer, comme dans l'exercice Temperature_Embarque tout ce qui concernait l'entrée analogique sur la port $A0$ par les instructions relatives au capteur DS18B20. Penser, quand même, à formater convenablement la sortie sur le moniteur série !

Téléverser le programme et ouvrir le moniteur série. Taper **G** et l'acquisition doit commencer (conserver un délai supérieur à 1000 ms).

b Côté ordinateur

Vous avez maintenant l'embarras du choix en fonction du public, du temps dont on dispose, des objectifs pédagogiques...

- recopier les données du moniteur série vers Excel ou dans un fichier texte pour l'exploiter en python ;
- controler directement l'acquisition dans la console python, ou dans un programme avec interface dédié.

Les bases étant posées dans les chapitres précédents, vous pouvez utiliser n'importe laquelle des solutions qui doit être quasiment fonctionnelle du premier coup !

Par exemple, le programme suivant permet une visualisation de l'évolution de la température en fonction du temps. Les données seront sauvegardées dans un fichier texte et on demande qu'il n'y ait pas de défilement sur l'axe des abscisses (ligne 3).

Listing 4.2 – Programme arduinoPlotT.py

```
1 import ArdTools.ArdTools as ard
2
3 a = ard.ArduinoPlot()
4 ports = a.ports()
```

```
5  a.connexion(ports[len(ports)-1], 9600)
6
7  a.sansDefilement()
8  a.sauvegarde("dataT.txt")
9  a.delay(100)
10 a.go()
```

c Côté calorimètre

Une solution simple et peu couteuse et qui, de plus, vous donnera des résultats certainement meilleurs que bon nombre de calorimètres « officiels » !

Vous trouvez une boite en plastique de 12 à 15 cm environ de côté (boite de congélation par exemple). Vous trouvez un bout de polystyrène (ou mousse de polyuréthane), vous la découpez pour qu'elle rentre dans la boite. Vous faites un trou pour introduire un bécher de 100 mL forme haute. Une petite plaque en plastique pour protéger des projections éventuellement, une encoche pour faire passer les fils de la sonde de température et... le tour est joué !

Et vous disposez d'un calorimètre :

- dans lequel vous allez pouvoir manipuler avec 25 mL de produit environ ;
- vous allez pouvoir agiter car le bareau magnétique n'est pas trop loin de l'agitateur ;
- last but not least, vous avez un calorimètre d'une capacité thermique de l'ordre de 30 $J.K^{-1}$. [1]

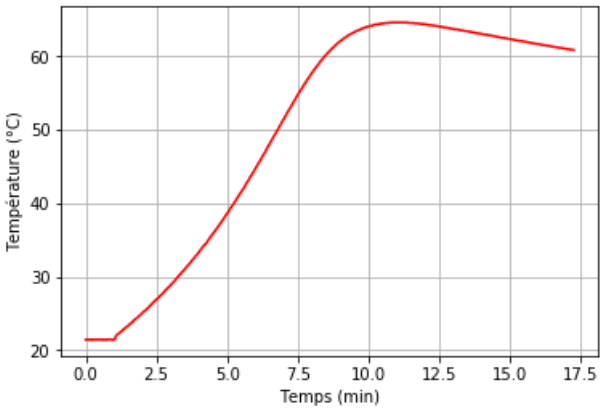
.

d Côté réaction chimique

Versez, par exemple 20 mL d'eau oxygénée commerciale diluée 4 fois. Commencez l'acquisition de la température en fonction du temps, ajoutez 5 mL d'une solution de KI à 0,1 $mol.L^{-1}$ au bout de 60 s, fermez le calorimètre et attendre environ 5 minutes après l'obtention du maximum de température.

La courbe température en fonction du temps a alors l'allure représentée ci-contre.

Récupérer les fichiers de points dans un programme python permet ensuite un traitement numériques des données afin de déterminer les caractéristiques de cette réaction.



1. En gros vous avez une capacité thermique de $30 + 25 \times 4 = 150$ $J.K^{-1}$. Du coup avec 0,01 mole d'un réactif pour lequel l'enthalpie standard de réaction vaut 100 $kJ.mol^{-1}$, on peut s'attendre à une augmentation de température d'environ 7 degrés.

3 Plusieurs données identiques à gérer

Le composant utilisé, **DS18B20**, offre l'avantage de pouvoir utiliser plusieurs capteurs de température sur le même port de l'**Arduino** (port **12** en l'occurence). On peut ainsi « facilement » mettre en œuvre quelques expériences intéressantes :

- diffusion de la chaleur ;
- étude d'un échangeur thermique ;
- détermination d'une température de fusion ou de ramollissement (inférieure à 120 degrés) ;
- ...

La principale difficulté réside dans l'identification des sondes. Cette fois encore, consulter un programme exemple proposé dans la bibliothèque va nous permettre de trouver une solution.

3.1 Exemple *Multiple* de la bibliothèque

Connecter la seconde sonde de température à la platine (il faudra peut-être débrancher et rebrancher la carte pour que la sonde soit prise en compte). Ouvrir et téléverser le programme **Multiple** (Fichier > Exemple > DallasTemperature > Multiple dans l'IDE d'Arduino) . Modifier la ligne 6 du programme afin de préciser le bon numéro pour le port OneWire : `#define ONE_WIRE_BUS 7` (modifier éventuellement la ligne 7 `#define TEMPERATURE_PRECISION 12` afin de ne pas changer la résolution du capteur par défaut) puis lancer l'exécution du programme.

En sortie on obtient, dans la console de l'IDE :

```
Dallas Temperature IC Control Library Demo
Locating devices...Found 2 devices.
Parasite power is: OFF
Device 0 Address: 2820D7459208026B
Device 1 Address: 2883B27791110258
Device 0 Resolution: 9
Device 1 Resolution: 9
Requesting temperatures...DONE
Device Address: 2820D7459208026B Temp C: 22.00 Temp F: 71.60
Device Address: 2883B27791110258 Temp C: 24.50 Temp F: 76.10
...
```

Nos deux capteurs ont bien été reconnus. Chaque puce DS18B20 possède une unique adresse gravée dans sa ROM. Cet identificateur ne peut-être modifié. On peut alors accéder à un capteur donné soit par son indice dans la liste des capteurs détectés sur le port OneWire, soit par son adresse. On peut très bien se passer des adresses, les résultats sont toujours donnés dans le même ordre.

Nous n'allons pas imprimer ici tout le programme mais noter les lignes qui nous intéressent maintenant :

Listing 4.3 – Programme Multiple (extrait)

```
1 #define ONE_WIRE_BUS 7
2 OneWire oneWire(ONE_WIRE_BUS);
3 DallasTemperature sensors(&oneWire);
4
5 DeviceAddress insideThermometer, outsideThermometer;
6
```

```
7  void setup(void)
8  {
9  ...
10   Serial.print(sensors.getDeviceCount(), DEC);
11 ...
12   if (!sensors.getAddress(insideThermometer, 0))
        Serial.println("Unable to find address for Device 0");
13   if (!sensors.getAddress(outsideThermometer, 1))
        Serial.println("Unable to find address for Device 1");
14 ...
15   printAddress(insideThermometer);
16 ...
17 }
18
19 void printTemperature(DeviceAddress deviceAddress)
20 {
21   float tempC = sensors.getTempC(deviceAddress);
22   Serial.print("Temp C: ");
23   Serial.print(tempC);
24   Serial.print(" Temp F: ");
25   Serial.print(DallasTemperature::toFahrenheit(tempC));
26 }
27
28 void printData(DeviceAddress deviceAddress)
29 {
30   Serial.print("Device Address: ");
31   printAddress(deviceAddress);
32   Serial.print(" ");
33   printTemperature(deviceAddress);
34   Serial.println();
35 }
36
37 void loop(void)
38 {
39 ...
40   sensors.requestTemperatures();
41 ...
42   printData(insideThermometer);
43   printData(outsideThermometer);
44 }
```

Que fait-on ?

- L'initialisation de la variable *sensors* se fait comme dans les programmes précédents (lignes 1 à 3).
- On sait ici que l'on a 2 capteurs de températures, on stocke leurs adresses dans les variables *insideThermometer* et *outsideThermometer* (la bibliothèque permet de gérer jusqu'à 8 capteurs).
- L'instruction `sensors.getDeviceCount()` ligne 10 est importante ; elle permet de connaître le nombre de capteurs DS18B20 sur le port utilisé. La valeur de retour n'est pas exploitée ici mais elle permettrait de nous assurer que les deux capteurs sont effectivement bien connectés.
- Lignes 12 et 13 : on accède aux adresses des capteurs par leur indice, les valeurs sont stockées dans les variables *insideThermometer* et *outsideThermometer*.

- Une routine non explicitée ici permet d'afficher l'adresse d'un capteur (il faudra penser à copier la routine `printAddress` si on veut utiliser les adresses par la suite !).
- La ligne 49 est essentielle : l'instruction `sensors.requestTemperatures()` force chaque capteur sur le port utilisé à lancer une acquisition de température.
- On appelle ensuite `printData` avec comme paramètre l'adresse de chaque capteur, routine qui affiche effectivement l'adresse du capteur puis appelle la routine `printTemperatures` toujours avec l'adresse du capteur comme paramètre.
- L'instruction `sensors.getTempC` permet alors d'accéder, via son adresse, à la température sur chaque capteur.

3.2 Simplifions tout ça !

Bon, essayons maintenant de simplifier ce programme afin de l'adapter à nos besoins, en particulier disposer d'une sortie formatée. On se contentera, dans un premier temps d'un programme « basique » avec un délai dans la boucle `loop`

Bien souvent, la gestion des adresses n'est pas nécessaire. Nous avions vu ligne 40 du programme **Simple** (listing 4.1) l'instruction `getTempCByIndex(0)` qui permettait d'accéder à la température d'un capteur par son indice. Qu'a-t-on à modifier par rapport au listing précédent ?

- Au lieu de stocker les adresses, il faut savoir combien de capteurs sont effectivement connectés : on remplacera les variables *insideThermometer* et *outsideThermometer* par la variable *nbTemperature*.
- Les lignes 12, 13 ne nous servent plus à rien, tout comme la routine `printAddress` puisque l'on peut tout gérer par les indices.
- Il nous reste alors à modifier la boucle principale pour l'adapter à notre format de sortie.

Le programme suivant implémente ces modifications.

Listing 4.4 – Programme Temperatures

```
1  #include <OneWire.h>
2  #include <DallasTemperature.h>
3
4  #define BAUD            9600
5  #define ONE_WIRE_BUS    7
6  OneWire oneWire(ONE_WIRE_BUS);
7  DallasTemperature sensors(&oneWire);
8
9  byte nbTemperature ;
10
11 void setup(void)
12 {
13   Serial.begin(BAUD);
14   sensors.begin();
15   nbTemperature = sensors.getDeviceCount() ;
16   Serial.println("Nombre de capteurs de température : " +
       String(nbTemperature)) ;
17 }
18
19 void loop(void)
20 {
21   sensors.requestTemperatures();
22   String msg = "DATA:TIME:" + String(millis()) ;
```

```
23    for (int i = 0 ; i < nbTemperature ; i++)
24    {
25      msg = msg + ":T" + String(i) + ":" +
            String(sensors.getTempCByIndex(i));
26    }
27    Serial.println(msg);
28  }
```

Exercice 5 Deux températures sur l'écran LCD
Dans le premier exercice de ce chapitre (page 62), il fallait concevoir un système embarqué affichant la température du capteur soudé à la carte sur l'écran LCD. Modifier le programme afin d'afficher maintenant les deux températures.

3.3 Et pour accéder aux données ?

Les modifications précédentes ont été implémentées dans le programme **Temperatures_Modele** que vous pouvez charger sur votre carte Arduino.

a A partir d'un programme python

Jusqu'à maintenant, nous n'avions qu'une seule donnée en fonction du temps. La routine `lecture2Data` dans le fichier `util.py` permettait d'extraire ces données. Dans ce même fichier, nous proposons une fonction **lectureData(nomFichier)** permettant d'extraire un nombre quelconque de données. L'instruction `titre, data = lectureData('monFichier.txt')` donne, en retour :

- comme premier paramètre une liste de chaînes de caractères correspondant aux libellés des grandeurs transmises (ici, titre correspond à ['TIME', 'T1', 'T2'] puisque ce sont les noms que l'on a donné dans le programme sur la carte Arduino) ;
- comme second paramètre une liste de listes : on accède à la ligne i par `data[i]`, cette ligne est une liste qui contient, dans l'ordre la valeur du temps, de la température T1 et de la température T2.

Le programme suivant montre comment exploiter ces données à partir du fichier '2temperatures.txt' créé, par exemple, à l'aide du programme arduinoPlotT.py.//

Avec Spyder il faut choisir `Automatic` dans l'onglet `Graphics` des réglages `IPython console` des préférences de spyder pour le tracé dynamique avec matplotlib !

Listing 4.5 – Programme 2temperatures.py

```
1  import matplotlib.pyplot as plt
2  import util as util
3
4  Titre, Data = util.lectureData('2temperatures.txt')
5
6  Temps = []
7  T1 = []
8  T2 = []
9  for i in range(len(Data)):
10     Temps.append(Data[i][0])#temps en premier
11     T1.append(Data[i][1])#puis T1
12     T2.append(Data[i][2])#puis T2
13
14 plt.plot(Temps, T1, label = 'T1', color = 'r')
15 plt.plot(Temps, T2, label = 'T2', color = 'b')
```

```
16 plt.legend()
17 plt.xlabel("Temps (s)")
18 plt.ylabel("Température (C)")
19 plt.show()
```

b A partir du programme TraceQt

On branche, on se connecte, on lance l'acquisition et... l'évolution des deux températures en fonction du temps s'affiche ! //

Avec Spyder il faut choisir `Inline` dans l'onglet `Graphics` des réglages `IPython console` des préférences de spyder pour le tracé dynamique avec matplotlib !

Activité 5

Pour aller plus loin !

1 Faire et refaire, c'est toujours travailler !

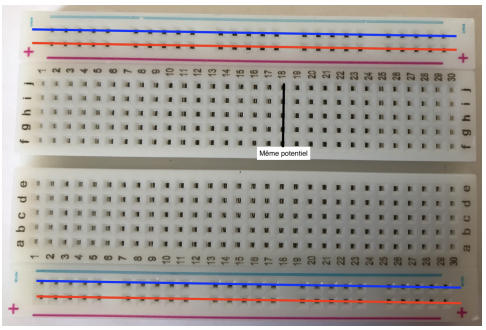
Dans les premiers chapitres de ce guide, vous n'aviez aucun câblage à réaliser puisque tout était déjà soudé sur la platine de l'**ArdSSK**.

Vous disposez, par ailleurs, des principaux composants et du matériel vous permettant de réaliser les câblages vous mêmes...
A l'aide de la platine d'expérimentation, vous pouvez, sans oublier les éventuelles résistances de protection :

- refaire le montage de la LED RGB ;
- refaire le montage du potentiomètre et du bouton poussoir ;
- refaire le montage du circuit RC (vous disposez de différentes résistances et différents condensateurs) ;
- ...

La platine d'expérimentation mise à votre disposition propose :

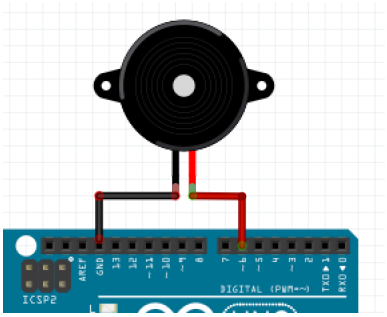
- deux lignes (-), en bleu, non connectées entre elles ; relier au moins l'une d'entre elle à la masse de l'Arduino ;
- deux lignes (+), en rouge, non connectées entre elles ; *a priori* l'une d'entre elle sera reliée au + 5V de l'Arduino ;
- 60 lignes permettant de réaliser votre montage. Les connecteurs sont reliés entre eux perpendiculairement à l'axe de la plaque et ce trouvent donc au même potentiel.



2 Et vous pouvez aussi vous faire la main avec. . .

2.1 Un buzzer

Ce composant est particulièrement simple à cabler puisqu'il suffit de relier l'une des bornes du buzzer à la masse et l'autre sur une entrée PWM (broche 6 par exemple). On peut éventuellement rajouter une résistance ($\approx 100\ \Omega$) en série.



Une fonction Arduino existe clé en main pour jouer un son, il s'agit de la fonction **tone(port, fréquence, durée)** qui accepte comme paramètres le port utilisé, la fréquence en Hz (supérieure à 32 Hz) et, éventuellement, la durée (en ms). L'appel de cette fonction n'est pas bloquant ; l'exécution du code continue sans délai.

Cette fonction modifie le comportement de la sortie PWM (cf page 17) puisqu'ici, on a automatiquement un duty cycle de 50% (la sortie est 50% du temps à un niveau haut et 50% à un niveau bas) mais, par contre, on peut faire varier la fréquence.

Voici un exemple de code qui, toutes les secondes émet une note `la` pendant 100 ms.

Listing 5.1 – Programme Buzzer

```
1  #define buzzPin 6
2  #define la      440
3
4  void setup() {
5    pinMode(buzzPin, OUTPUT) ;
6  }
7
8  void loop() {
9    tone(buzzPin,la,100) ;
10   delay(900) ;
11 }
```

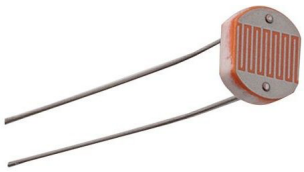
Exercice 1 SOS Câbler sur la platine d'essai un bouton poussoir (et sa résistance de rappel) ainsi qu'un buzzer. Quand on appuie sur le bouton, on doit entendre un SOS en Morse !
Remarque : à défaut de bouton poussoir. . . contentez vous de faire un contact temporaire entre 2 connexions.

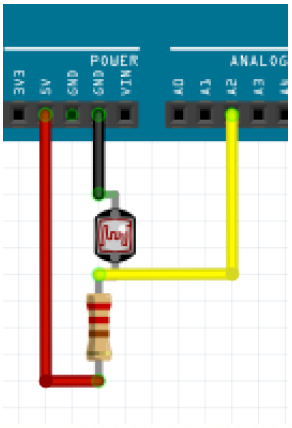
2.2 Étude d'un capteur résistif

Un grand classique également des kit Arduino. L'étude d'un tel capteur est au programme de première.

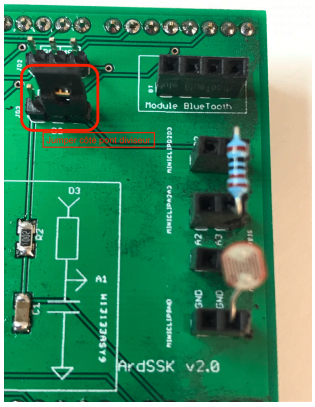
On branche la résistance variable (photorésistance ou thermistance) en série avec une résistance de protection (typiquement 1 à 10 kΩ) afin de créer un diviseur de tension. On mesure la tension aux bornes de la photorésistance ou de la thermistance à l'aide d'une entrée analogique.

a Une photorésistance

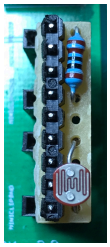




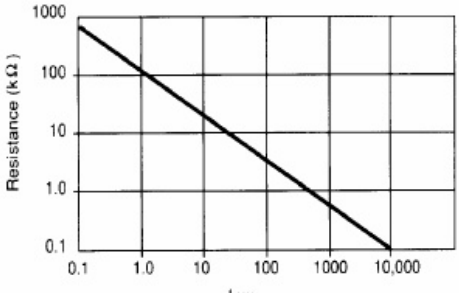
Une première solution consiste à câbler sur la platine d'essai le pont diviseur entre la broche 5V et la masse.



Une seconde solution consiste à utiliser le pont diviseur câblé sur la partie droite du module ArdSSK. Il faut penser à mettre le cavalier **D3** côté pont diviseur. Avec le montage sur la photo, la lecture analogique se fait sur le port **A3**. Ce montage permet, par exemple en TP, une étude du capteur en minimisant les risques !



Ces risques sont encore minimisés si vous fabriquez une mini carte qui vient se clipser sur les connecteurs `D3-A3-A3-GND` cf paragraphe 1.1 page 98.



La résistance de la photorésistance diminue quand l'intensité lumineuse augmente.

Le programme minimal d'utilisation est alors très simple.

Listing 5.2 – Programme LDR2

```
1  #define sensorPin   A2
2
3  void setup() {
4    Serial.begin(9600) ;
5  }
6
7  void loop() {
8    int sensorValue = analogRead(sensorPin);
9    Serial.println(sensorValue) ;
10 }
```

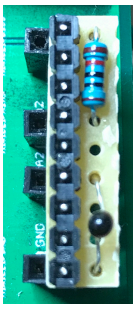
En alimentant par la broche **D3**, il faut modifier quelque peu ce programme.

Listing 5.3 – Programme LDR

```
#define sensorPin   A3
#define alimPin 3

void setup() {
  Serial.begin(9600) ;
  pinMode(alimPin, OUTPUT);
  digitalWrite(alimPin, HIGH) ;
}

void loop() {
  int sensorValue = analogRead(sensorPin);
  Serial.println(sensorValue) ;
  delay(1000) ;
}
```

b Une thermistance

La thermistance proposée est de type MF52-B-3950.
C'est une résistance CTN (ou NTC en anglais) à coefficient négatif de température. A 25 ° C, la valeur de la résistance vaut $R_{25} = 10k\Omega$ (à 5% près) et la résistance R_T diminue quand la température augmente.



Température (° C)	0	10	15	20	25	30	35
R (kΩ)	98,96	20,00	15,76	12,51	10,00	8,05	6,52

L'équation de Steinhart-Hart permet de lier la température absolue T à la résistance R_T. Sur une plage limitée de température autour de 25 ° C, on peut utiliser la formule : $\frac{R_T}{R_{25}} = \exp\left(B \times \left(\frac{1}{T} - \frac{1}{T_{25}}\right)\right)$. Pour notre capteur, la valeur de B vaut 3950 K.

Le programme Thermistance permet d'afficher la température mesurée par la thermistance (on utilise le même montage que pour le programme LDR : résistance de 10 kΩ). On a comparé cette valeur à celle donnée par le capteur DS18B20 (cf chapitre 4)... capteur réputé être plus précis qu'une thermistance.

Listing 5.4 – Programme Thermistance

```
#include <OneWire.h>
#include <DallasTemperature.h>

#define ONE_WIRE_BUS 7
#define sensorPin   A3
#define alimPin 3
OneWire oneWire(ONE_WIRE_BUS);
DallasTemperature sensors(&oneWire);

float R0 = 10000 ; // résistance du pont diviseur
```

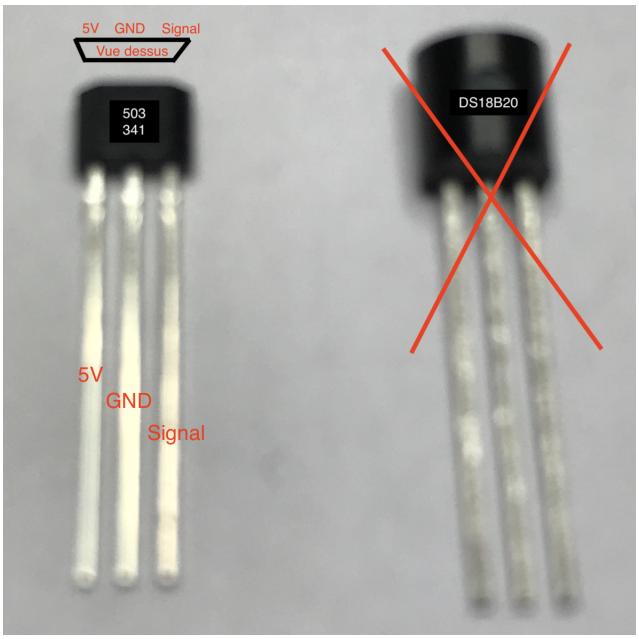
```
11 float R25 = 10000; // résistance de référence de la
       thermistance
12 float B = 3950 ; // coefficient de température de la
       thermistance
13
14 void setup()
15 {
16   Serial.begin(9600) ;
17   sensors.begin();
18   pinMode(alimPin, OUTPUT);
19   digitalWrite(alimPin, HIGH) ;
20 }
21
22 void loop()
23 {
24   sensors.requestTemperatures(); // Send the command to get
       temperatures
25   float T = sensors.getTempCByIndex(0);
26   float T2 = getT() ;
27   Serial.println("Température : DS18B20 : " + String(T) + "
       RTC : " + String(T2)) ;
28   delay(1000) ;
29 }
30
31 float getT()
32 {
33   int sensorValue = analogRead(sensorPin);
34   float RT=R0/((1023.0/sensorValue)-1) ;
35   float T = 1/(log(RT/R25)/B+1/298.15);
36   return T-273.15 ;
37 }
```

La routine intéressante est `getT()` :

- *sensorValue* correspond à la lecture analogique (comprise entre 0 et 1023) ;
- la formule du diviseur de tension s'écrit : $\frac{u(lu)}{u(5V)} = \frac{R_T}{R_0 + R_T}$; on en déduit l'expression de R_T ;
- reste ensuite à calculer la température.

2.3 Un capteur à effet Hall

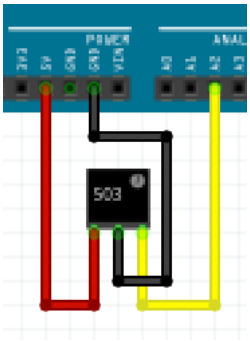
Son petit nom de baptême est **AH3503** (à ne pas confondre, dans le kit avec le capteur de température DS18B20).



Le plus difficile est d'identifier les sorties. Le capteur présente une grande face plate et une plus petite sur laquelle est gravée 503-341. C'est la face visible sur la photo, dans ces conditions la broche +5V est à gauche et la sortie à droite.

C'est un capteur linéaire susceptible de mesurer des champ magnétiques positifs ou négatifs. En l'absence de champ magnétique la broche signal se trouve à un potentiel de 2,5 V et la sensibilité est voisine (d'après les indications du Datasheet) de 1,30 mV par Gauss (entre -900 et 900 G au centre de la face imprimée).

Une fois les sorties identifiées, le câblage est simple.



Le code ne devrait pas non plus poser de problème.
Pour complexifier un peu, on souhaite afficher la valeur du champ magnétique estimé (en considérant une sensitivité du capteur de 1,30 mV/G). Pour cela on cherche d'abord la valeur lue sur le convertisseur en absence de champ magnétique. La valeur est stockée dans la variable globale *milieu*. La conversion est ensuite réalisée (ligne 15). On a également pris la valeur moyenne de la lecture sur 10 mesures afin de minimiser les fluctuations du signal.

Listing 5.5 – Programme Magnetique

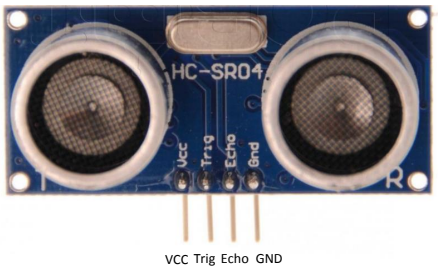
```
1  # define sensorPin     A2
2  # define nbIterMesure  10
3  int milieu ;
4
5  void setup()
6  {
7    Serial.begin(9600);
8    findMilieu();
9    Serial.println("Calibration : point milieu pour AnalogRead =
       " + String(milieu)) ;
10 }
11
12 void loop()
13 {
14   int valeur = mesure() ;
15   Serial.println("Estimation champ magnétique (G) : " +
```

```
      String((valeur - milieu) * 1.3 * 5000 / 1023)) ;
  delay(1000);
}

void findMilieu()
{
  unsigned long calibration = 0 ;
  int nbIter = 10  ;
  for (int i = 0; i < nbIter; i++)
    calibration = calibration + mesure() ;
  milieu = int(calibration / nbIter) ;
}

int mesure()
{
  unsigned long somme = 0 ;
  for (int i = 0 ; i < nbIterMesure ; i++)
    somme = somme + analogRead(sensorPin);
  return int(somme / nbIterMesure) ;
}
```

Une application comme compte-tour est présentée dans les expériences possibles (cf chapitre 6-3.6 page 111).

2.4 Le « célèbre » capteur à ultrason

Son petit nom de baptême est **HC-SR04**. On le retrouve dans de nombreux kit d'introduction puisqu'il permet de modéliser un radar de recul à moindre frais. Il s'agit d'un capteur de distance possédant un émetteur et un récepteur. En présence d'un obstacle, les ondes émises sont réfléchies. La mesure de l'écart de temps entre émission et réception permet de remonter à la distance à l'obstacle.



Le principe du programme est le suivant :

- on envoie une impulsion (HIGH) de 10 μs sur la broche `Trigger` du capteur ;
- le capteur émet alors une série de 8 impulsions ultrasoniques (40 kHz) et place l'état de la broche `Echo` à HIGH ;
- les ultrasons se propagent dans l'air, rencontrent un obstacle et reviennent vers le capteur ;
- le capteur détecte l'écho, clôture la prise de mesure et fait passer la broche `Echo` à LOW ;
- une fonction `pulseIn(pin, etat, timeout)` de la bibliothèque Arduino permet de connaitre (en microsecondes) la durée d'une impulsion dans un état donné (HIGH ou LOW). La fonction retourne 0 si aucun écho n'a été détecté pendant la durée *timeout*.

Une étude complète dans le carnet du Maker donne une incertitude de l'ordre de 5% entre 10 et 50 cm ; 1 % au delà. Ils préconisent une surface de 50 cm^2 en face du capteur ; le cas défavorable étant le capteur posé sur une table. (.carnetdumaker.net/articles/mesurer-une-distance-avec-un-capteur-ultrason-hc-sr04-et-une-carte-arduino-genuino)

a Détermination d'une distance

Aucune bibliothèque n'est nécessaire. On connecte, en adéquation avec le code proposé, les broches `trig` et `echo` respectivement aux ports 2 et 3 de la carte.

Le code du programme proposé est le suivant :

Listing 5.6 – Programme Radar (cf Carnet du Maker)

```
{
/* Constantes pour les broches */
#define TRIGGER_PIN     2          // Broche TRIGGER
#define ECHO_PIN        3          // Broche ECHO

/* Constantes pour le timeout */
#define MEASURE_TIMEOUT 25000UL    // 25ms = ~8m à 340m/s

/* Vitesse du son dans l'air en cm/us */
#define SOUND_SPEED     0.034

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

  /* Initialise les broches */
  pinMode(TRIGGER_PIN, OUTPUT);
  digitalWrite(TRIGGER_PIN, LOW); // La broche TRIGGER doit ê
     tre à LOW au repos
  pinMode(ECHO_PIN, INPUT);
}

void loop()
{
  /* 1. Lance une mesure de distance en envoyant une impulsion
     HIGH de 10us sur la broche TRIGGER */
  digitalWrite(TRIGGER_PIN, HIGH);
  delayMicroseconds(10);
  digitalWrite(TRIGGER_PIN, LOW);

  /* 2. Mesure le temps entre l'envoi de l'impulsion
     ultrasonique et son écho (si il existe) */
  long measure = pulseIn(ECHO_PIN, HIGH, MEASURE_TIMEOUT);

  /* 3. Calcul la distance à partir du temps mesuré */
  float distance_cm = measure * SOUND_SPEED / 2.0 ;

  /* Affiche les résultats*/
  Serial.println("Distance (cm):" + String(distance_cm,1) ) ;

  /* Délai d'attente pour éviter d'afficher trop de résultats à
      la seconde */
  delay(500);
}
```

Exercice 2 Radar de recul. Réaliser à l'aide du buzzer et du module HC-SR04 un montage permettant de modéliser un radar de recul automobile : en plaçant un obstacle au delà de 50 cm, aucun son n'est émis ; lorsque l'obstacle se rapproche, un bip sonore de plus en plus long est émis.

b Créons notre propre bibliothèque

Profitons de cet exemple pour montrer comment créer sa propre bibliothèque. Il s'agit de montrer que ne n'est pas très compliqué (surtout lorsqu'on connait le langage C++ ou la programmation objet). D'un point de vue pédagogique, le code précédent permet d'expliquer le principe de fonctionnement du capteur. On peut aussi envisager une expérience ou l'élève n'a pas besoin de comprendre ce principe car l'attention est portée sur l'analyse des distances. L'utilisation d'une bibliothèque personnelle permet alors de lui simplifier considérablement la tâche.

b.1 Première étape : création d'une classe

Un certains nombre d'instructions du programme `Radar` (lignes 7 et 10 pour la définition de constantes ; lignes 17,18,19 pour l'initialisation puis lignes 25 à 33 pour le calcul de la distance) permettent de gérer le comportement du périphérique connecté. L'idée est de créer un objet (par exemple de type `Radar`) qui va s'occuper de tout ce qui lui est propre. Dans ce langage (comme en python d'ailleurs), celà se fait à l'aide d'une classe.
Le programme `Radar2` suivant fait exactement la même chose que le programme `Radar`.

Listing 5.7 – Programme Radar2

```
1  /* CONSTANTES */
2
3  #define TRIGGER_PIN          2          // Broche TRIGGER
4  #define ECHO_PIN             3          // Broche ECHO
5  #define MEASURE_TIMEOUT      25000      // 25 ms
6  #define SOUND_SPEED          0.034      // cm/us
7
8  class Radar
9  {
10     public :
11     Radar(byte triggerPin, byte echoPin)
12     {
13         trigger_pin = triggerPin ;
14         echo_pin = echoPin ;
15         pinMode(trigger_pin, OUTPUT);
16         digitalWrite(trigger_pin, LOW);
17         pinMode(echo_pin, INPUT);
18     }
19
20     float distance()// en cm
21     {
22         digitalWrite(trigger_pin, HIGH);
23         delayMicroseconds(10);
24         digitalWrite(trigger_pin, LOW);
25         long measure = pulseIn(echo_pin, HIGH,
               MEASURE_TIMEOUT);
26         return measure * SOUND_SPEED / 2.0  ;
27     }
28     private :
29     byte trigger_pin ;
30     byte echo_pin ;
31 } ;
32
33 Radar radar(TRIGGER_PIN, ECHO_PIN) ;
34
```

```
35 void setup()
36 {
37   Serial.begin(9600);
38 }
39
40 void loop()
41 {
42   float distance_cm = radar.distance() ;
43   Serial.println("Distance (cm):" + String(distance_cm, 1) ) ;
44   delay(500);
45 }
```

Que fait-on pour créer la classe (sans entrer dans les détails, juste pour pouvoir reproduire la démarche par copier/coller) ?

- Ligne 8 on définit une classe que l'on nomme `Radar`. La syntaxe impose le type `class`, une parenthèse ouvrante (ligne 9) et fermante (ligne 31). Cette dernière doit être suivie d'un point virgule.
- On souhaite qu'un certain nombre de fonctions (ou de variables) puissent être accessibles de l'extérieur de la classe, on précise qu'elles doivent être `public` (ligne 10), d'autres n'ont pas vocation à être modifiées de l'extérieur de la classe, on les déclare `private` (ligne 28).
- Chaque classe doit posséder un constructeur dont la syntaxe est le nom de la classe, des parenthèses (même s'il n'y a pas de paramètres), puis le code entre parenthèses. Ici on affecte les différents identificateurs de ports et on initialise les différentes entrées sorties (code qui était dans setup auparavant).
- Une fonction `distance` se charge de calculer la distance. On précise que la fonction retourne une valeur de type `float` (il faudrait préciser `void` s'il n'y avait pas de paramètres de retour). On retrouve dans cette fonction le code présent dans la boucle du programme Radar).

En gros, que doit-on faire ?

- instancier un objet de type `Radar` (par exemple) ;
- lui demander de nous renovoyer la distance à l'obstacle (par exemple en cm.

Et que fait-on pour l'utiliser ?

- ligne 33, on crée une variable *radar* de type /textttRadar en passant les paramètres permettant d'identifier les pins Trigger et Echo ;
- ligne 42, on demande de calculer la distance à l'aide de la fonction `distance`.

On teste, on rectifie éventuellement quelques bugs et... on peut passer à l'étape suivante.

b.2 Deuxième étape : création de la bibliothèque Notre classe `Radar` est tellement puissante qu'on aimerait pouvoir l'utiliser dans d'autres programmes sans avoir à la recopier systématiquement (dans le code de l'exercice sur le radar de recul par exemple !). Nous allons donc l'inclure dans une bibliothèque.

Parmi les constantes, les deux premières ne sont pas intrinsèque au radar, on peut très bien connecter les broches sur d'autres ports ; leur place est donc dans le programme et non dans la bibliothèque.
La vitesse du son, par contre, n'a rien à faire dans le programme puisqu'elle ne dépend pas de la façon dont le capteur est cablé. Il en va de même du délai maximum pour recevoir un écho.
On va donc mettre dans un fichier à part tout ce qui est compris entre les lignes 5 et 31. Conventionnellement on va le nommer `Radar.h`. Ce fichier doit être placé dans un dossier (`Radar` par exemple) lui même inclus dans le dossier `Library` de votre dossier `Arduino`. Il faut fermer et

réouvrir le logiciel afin que la bibliothèque soit prise en compte
.
Le fichier `Radar.h` correspond au listing suivant :

Listing 5.8 – Fichier Radar.h

```
1  #include "Arduino.h"
2
3  /* Constantes pour le timeout : 25ms = ~8m à 340m */
4  #define MEASURE_TIMEOUT      25000
5
6  /* Vitesse du son dans l'air en cm/us*/
7  #define SOUND_SPEED          0.034
8
9  class Radar
10 {
11     public :
12     Radar(byte triggerPin, byte echoPin)
13     {
14         trigger_pin = triggerPin ;
15         echo_pin = echoPin ;
16         pinMode(trigger_pin, OUTPUT);
17         digitalWrite(trigger_pin, LOW);
18         pinMode(echo_pin, INPUT);
19     }
20
21     float distance()// en cm
22     {
23         digitalWrite(trigger_pin, HIGH);
24         delayMicroseconds(10);
25         digitalWrite(trigger_pin, LOW);
26         long measure = pulseIn(echo_pin, HIGH,
               MEASURE_TIMEOUT);
27         return measure * SOUND_SPEED / 2.0  ;
28     }
29     private :
30     byte trigger_pin ;
31     byte echo_pin ;
32 } ;
33
34 #endif
```

La première ligne est nécessaire afin d'accéder, dans ce fichier aux fonctions et cosntantes propres de l'Arduino.

Dans le programme proprement dit, il faudra inclure cette bibliothèque (ligne 1) ; il restera alors un code très lisible.

Listing 5.9 – Programme Radar3

```
1  #include "Radar.h"
2
3  #define TRIGGER_PIN          2          // Broche TRIGGER
4  #define ECHO_PIN             3          // Broche ECHO
5  Radar radar(TRIGGER_PIN, ECHO_PIN) ;
6
```

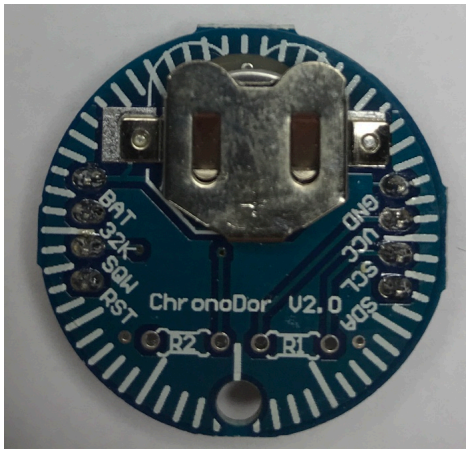
```
7  void setup()
8  {
9    Serial.begin(9600);
10 }
11
12 void loop()
13 {
14   float distance_cm = radar.distance() ;
15   Serial.println("Distance (cm):" + String(distance_cm, 1) ) ;
16   delay(500);
17 }
```

3 On complexifie un peu...

3.1 Une horloge en temps réel

Nous avons déjà évoqué dans le chapitre 2-2 page 18 la gestion du temps avec une carte Arduino. Pour un temps « très » court (disons quelques minutes) il n'y a pas trop de soucis à utiliser l'instruction`millis()` afin de connaître le temps écoulé depuis le branchement de la carte. On peut, parfois, mettre en œuvre des expériences beaucoup plus longue, ou, tout simplement, avoir besoin de connaître l'heure ! Une horloge en temps réel : RTC (Real Time Clock) en anglais est faite pour. De très nombreux systèmes embarqués en possède une, sans parler des ordinateurs.

Nous vous proposons dans ce kit une carte `ChronoDor` basée sur le circuit **DS3231** (ainsi qu'une pile bouton). Cette carte se connecte sur le port I2C (pour une fois, les connecteurs sont dans le même ordre que ceux fournis sur la platine ArdSSK !).



a Régler et afficher la date et l'heure

Bien sûr, nous avons choisi un circuit pour lequel existe une bibliothèque disponible. Nous l'avons incluse dans le dossier `libraries`. Ouvrez le programme **DS3231_simple** (accès via Fichier > Exemples > Arduino-DS3231-master > DS3231_simple). Téléchargez-le et lancez l'exécution. Vous constatez alors dans le moniteur série que la date et l'heure sont affichées... magique ! Que fait-ce programme ?

Listing 5.10 – Programme DS3231_simple

```
1 #include <Wire.h>
2 #include <DS3231.h>
3
4 DS3231 clock;
5 RTCDateTime dt;
```

```
6
7  void setup()
8  {
9    Serial.begin(9600);
10
11   // Initialize DS3231
12   Serial.println("Initialize DS3231");;
13   clock.begin();
14
15   // Set sketch compiling time
16   clock.setDateTime(__DATE__, __TIME__);
17 }
18
19 void loop()
20 {
21   dt = clock.getDateTime();
22
23   // For leading zero look to DS3231_dateformat example
24
25   Serial.print("Raw data: ");
26   Serial.print(dt.year);   Serial.print("-");
27   Serial.print(dt.month);  Serial.print("-");
28   Serial.print(dt.day);    Serial.print(" ");
29   Serial.print(dt.hour);   Serial.print(":");
30   Serial.print(dt.minute); Serial.print(":");
31   Serial.print(dt.second); Serial.println("");
32
33   delay(1000);
34 }
```

- Dans la boucle `loop`, on récupère (ligne 21) dans la variable *dt* une structure de type RTCDateTime qui contient les différentes informations affichées lignes 25 à 31.
- Dans le `setup`, après avoir initialisé la communication sur port I2C, une instruction clé, ligne 16 permet de fixer l'heure de l'horloge grâce à l'instruction `setDateTime`. Celle-ci accepte de nombreux paramètres ; ici, une syntaxe un peu particulière permet de spécifier la date et l'heure sur votre système d'exploitation.

Le programme **DS3231_dateformat** de la bibliothèque montre quelques exemples de formatage possible de la date et de l'heure.

Tant que l'on ne retire pas la pile bouton, vous disposez désormais de la date et l'heure sur cette carte sans avoir à réinitialiser l'horloge.

b Utilisation comme alarme

Toutes les horloges RTC disposent de cette fonctionnalité mais, celle-ci, peut également :

- fonctionner comme alarme ;
- générer un signal carré de 32768 Hz (sur la broche 32K) , 8192 Hz, 4096 Hz, 1024 Hz ou 1 Hz (sur la broche SQW).

Deux exemples sont proposés dans la bibliothèque.

b.1 On scrute l'alarme dans la boucle `loop` Le programme **DS3231_alarm** montre quelques solutions possibles pour poser une alarme. Nous proposons, ci-dessous, un programme simplifié pour gérer une alarme toutes les secondes.

Listing 5.11 – Programme Alarme

```
1  #include <Wire.h>
2  #include <DS3231.h>
3
4  DS3231 clock;
5
6  void setup()
7  {
8    Serial.begin(9600);
9    clock.begin();
10
11   // setAlarm1(Date or Day, Hour, Minute, Second, Mode, Armed
       = true)
12   clock.setAlarm1(0, 0, 0, 0, DS3231_EVERY_SECOND);
13 }
14
15 void loop()
16 {
17   if (clock.isAlarm1())
18   {
19     RTCDateTime dt = clock.getDateTime();
20     Serial.println(clock.dateFormat("d-m-Y H:i:s - l", dt));
21   }
22 }
```

Ce programme fonctionne. Toutes les secondes, la date et l'heure sont affichées.

- l'instruction ligne 12 précise le type d'alarme que l'objet *clock* doit gérer (ici une alarme toutes les secondes) ;
- puis, dans la boucle, on scrute régulièrement *clock* pour savoir si l'alarme est active. Si c'est la cas, on affiche l'heure.

Exercice 3 Quelle heure est-il ? Compléter le programme précédent afin d'afficher la date et l'heure sur l'écran LCD.

b.2 Utilisation d'une interruption Le programme précédent est, certes, fonctionnel mais oblige le programme principal à scruter régulièrement si l'alarme ne s'est pas déclenchée. Un peu comme si votre réveil ne sonnait pas mais affichait « C'est l'heure ! », charge à vous de le regarder à intervalle de temps régulier. Ce serait bien plus pratique si c'était l'horloge qui vous prévenait !

La stratégie à mettre en oeuvre porte le nom d'**interruption**. Celle-ci est d'une grande puissance : à l'instant souhaité, l'horloge va interrompre le programme s'exécutant sur l'Arduino pour exécuter une routine particulière. C'est, d'ailleurs, ce que faisait le timer : à intervalle de temps régulier, il forçait le programme à exécuter une fonction.

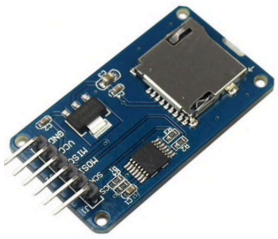
Sur la carte Arduino Uno, seuls les ports digitaux **2** et **3** permettent de gérer les interruptions. Le port utilisé ne doit pas être utilisé par ailleurs. ces ports ont été câblés pour autre chose sur la platine ArdSSK. Il ne faut donc pas l'utiliser pour ce paragraphe ! On rectifiera dans la prochaine version.

3.2 Un lecteur de carte SD pour enregistrer vos données

Le problème de stockage de données ne se pose pas pour un système connecté directement à l'ordinateur ou susceptible de communiquer par Bluetooth par exemple. Il suffit de récupérer via la communication série ces données. Charge alors à l'ordinateur (ou au smartphone) de gérer ces données.
Nous avons vu au paragraphe c.3 page 31 que l'utilisation de l'EEPROM de l'Arduino permettait de sauvegarder quelques informations. Cette solution très simple à mettre en œuvre est intéressant lorsque l'on souhaite que le système, par exemple, redémarre avec des paramètres choisis par l'utilisateur. Il ne faut pas utiliser, par contre, l'EEPROM pour stocker des données : d'une part l'espace est très restreint, d'autre part le nombre d'écriture sur la carte est relativement limité.

Si on dispose d'un système autonome, ou si on souhaite disposer d'un système de sauvegarde de données « extrêmement » rapide ; l'utilisation d'une carte SD peut s'avérer intéressante. Cette utilisation est grandement facilitée par une puissante bibliothèque disponible en standard sur Arduino.

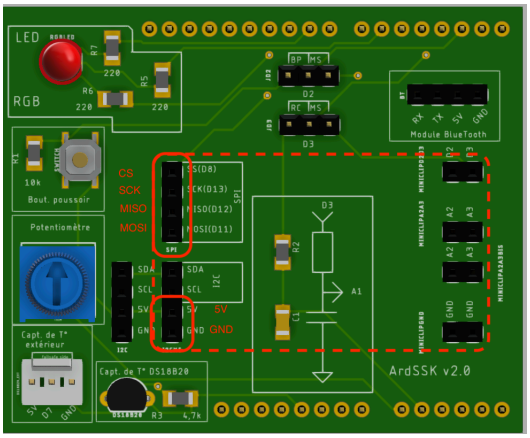
Le périphérique utilisé se connecte à l'arduino à l'aide d'un port **SPI**. Comme le bus I2C que nous avons déjà rencontré, ce bus permet une communication série avec un périphérique extérieur. Le bus SPI est bien plus rapide que le bus I2C mais un peu plus complexe à mettre en œuvre.



Six connexions sont nécessaires :

- alimentation (+ 5V ou 3,3 V) et masse ;
- une ligne d'horloge : `SCK` connectée obligatoirement à la pin `D13` de l'Arduino ;
- deux lignes de transfert `MOSI` et `MISO` respectivement connectées aux pin `D11` et `D12` de l'Arduino ;
- une ligne CS (Chip Select) que l'on connecte sur n'importe quel port numérique de l'Arduino (souvent le port 10, nous l'avons connectée sur le port `D8`) qui permet de sélectionner le périphérique I2C. Contrairement au protocole I2C, chaque périphérique aura son propre port CS permettant de l'activer.

Un bus SPI est proposé sur notre carte.
Ce dernier peut-même être utilisé sur une mini-carte de votre conception qui viendrait se clipser, en partie, justement sur le port SPI.



Connectez les différentes broches du module à l'Arduino.
L'IDE d'Arduino propose parmi ses « Exemples pour toutes cartes » quelques programmes de démonstration. Une description fort bien faite de l'utilisation de la carte SD est proposée par le `Carnet du Maker` (page : https ://www.carnetdumaker.net/articles/lire-et-ecrire-des-donnees-sur-une-carte-sd-avec-une-carte-arduino-genuino/).

Commençons par vérifier que la carte SD fonctionne bien. Pour cela, charger le programme **CardInfo**. Ce programme fait partie de l'IDE d'Arduino : menu `Fichier` puis `Exemple` puis, dans

la catégorie `Exemples pour toutes cartes` sélectionner le sous-menu `SD` et enfin le programme souhaité.

Si tout se passe bien, à l'exécution, vous devez voir apparaitre dans le moniteur série les principales caractéristiques de la carte SD jointe au kit de développement. Il s'agit d'une carte SD de 120 Mo, son formatage est FAT16 et aucun fichier n'est présent sur la carte.
Vous pouvez, bien sûr, tester les autres programmes proposés.

Mais revenons à notre « cahier des charges ». On souhaite :

1. Faire l'acquisition d'une grandeur à l'aide de l'Arduino.
2. Stocker les informations correspondantes sur la carte SD.
3. Pouvoir lire ces informations ultérieurement.

Le premier point ne pose plus de problème pour nous. En anticipant un peu sur la suite, nous pouvons prendre comme programme de référence le programme `ADC_ModeleNoTimer` du chapitre 3. Tout y est... ou presque.

Nous proposons alors le programme **DataLogging_Modele**. Seules les modifications par rapport au programme d'origine sont reportées dans le listing ci-dessous.

Listing 5.12 – Programme DataLogging_Modele

```
...
#include <SPI.h>
#include <SD.h>
...
// variables spécifiques au système étudié
#define chipSelect  8
#define logFile     "data.txt"
File dataFile ;

void setup()
{
  ...
  if (!SD.begin(chipSelect))
  {
    Serial.println("INFO:Problème lecteur carte SD") ;
  }
}

void affiche(String msg)
{
  Serial.println(msg) ;
  BT.println(msg) ;
  if (dataFile)
    dataFile.println(msg) ;
}

void stop()
{
  doJob = false ;
  if (dataFile)
    dataFile.close() ;
}
```

```
33
34 void parse(char ordre, long valeur)
35 {
36   switch (ordre)
37   {
38     case 'A' : // append
39       dataFile = SD.open(logFile, FILE_WRITE) ;
40       break ;
41     case 'W' : // write
42       if (SD.exists(logFile))
43         SD.remove(logFile) ;
44       dataFile = SD.open(logFile, FILE_WRITE) ;
45       break ;
46     case 'R' : // read
47       read() ;
48       break ;
49     default :
50       break ;
51   }
52 }
53
54 void read()
55 {
56   dataFile = SD.open(logFile) ;
57   if (dataFile)
58   {
59     while (dataFile.available())
60       Serial.write(dataFile.read()) ;
61     dataFile.close() ;
62   }
63 }
```

Qu'a-t-on modifier ?

- Lignes 2 et 3 : on importe les bibliothèques SPI et SD.
- Lignes 6 et 7 : on précise le port ChipSelect utilisé ainsi que le nom du fichier.
- Ligne 8 : on crée une variable *dataFile* de type `File` dont le rôle est essentiel par la suite car c'est elle qui nous permettra lecture et écriture sur le fichier texte proprement dit.
- Dans le `setup`, il faut initialiser (ligne 13) la communication avec la carte SD avec le bon ChipSelect. On aurait pu utiliser un objet (comme dans le programme CardInfo par exemple).
- On ajoute quelques ordres à passer à la carte Arduino. Ligne 38, si on passe **A** (comme Append) on ouvre le fichier en écriture (s'il n'existe pas il est créé) et on écrit à la fin du fichier ; les informations déjà présentes ne sont pas perdues. Si on passe l'ordre **W** (pour Write) (ligne 41), si le fichier existe, on l'efface. On le crée à nouveau et on l'ouvre en écriture. Enfin ligne 38, si on envoie l'ordre **R** (pour Read), on appelle la fonction correspondante.
- Cette fonction `read` (lignes 46) se charge d'ouvrir le fichier en lecture, parcourir le fichier et afficher toutes les lignes une à une dans le port série.
- Ces données ont été écrites ligne 24 dans la routine d'affichage traditionnelle.
- Il faut juste penser à fermer le fichier lorsque l'on stoppe l'acquisistion (ligne 31).

Sans forcément connecter de capteur, le programme est fonctionnel. Nous verrons dans le paragraphe suivant un exercice d'application.

On peut imaginer d'autre applications à une carte SD comme, par exemple, écrire dans un fichier texte une suite d'instructions que l'on pourrait exécuter, à la demande, sur la carte.

3.3 Un capteur température-humidité-pression

Le composant de base est le **BME280** ; il s'agit d'une carte permettant l'acquisition de la température, de la pression et de l'humidité ambiante. Il se connecte soit par bus I2C, soit par SPI. D'après le Datasheet du constructeur, ce composant permet de mesurer :

- la température entre 0 et 65 °C (à 0,5 degré près) ;
- la pression entre 300 et 1100 hPa (à 1 hPa près) ;
- le taux d'humidité relative entre 0 et 100 à 3 % près. L'humidité relative (ou degré hygrométrique) correspond au rapport entre la pression partielle de la vapeur d'eau contenue dans l'air et la pression de vapeur saturante à la même température.

La bibliothèque permettant d'exploiter ce capteur connecté sur le port I2C vous a été fournie. Dès lors, il suffit, une fois le module connecté sur le port I2C, de charger le programme de test : `Fichier > Exemples > Adafruit BME280 Librarary > BME280Test`. Lors de l'exécution : température, pression et humidité relative sont affichées dans le moniteur série ainsi qu'une ligne « Approx. Altitude ».

Les principales instructions nécessaires dans ce programme sont reportées dans le listing ci-dessous.

Listing 5.13 – Programme bme280Test

```
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>

#define SEALEVELPRESSURE_HPA (1013.25)

Adafruit_BME280 bme; // I2C

void setup() {
...
  bme.begin() ;
...
}

void printValues() {
...
    Serial.print(bme.readTemperature());
    Serial.print(bme.readPressure() / 100.0F);
    Serial.print(bme.readHumidity());
...
}
```

En météorologie, si on souhaite comparer la pression atmosphérique entre différents lieux, les pressions mesurées doivent être ramenées au niveau de la mer. Un baromètre doit toujours indiquer la pression au niveau de la mer. Un réglage est alors nécessaire soit par une solution mécanique, soit par un calcul.
La formule internationale du nivellement barométrique permet de relier la pression p mesurée, la pression ramenée au niveau de la mer p_0 et l'altitude z (en mètre) :$p = p_0 \times \left(1 - \frac{z}{44330}\right)^{5;255}$.

Une fois connue, à un instant donné, la pression ramenée au niveau de la mer, on peut alors se

servir du capteur comme altimètre : $z = 44330 \times \left(1 - \left(\frac{p}{p_0}\right)^{\frac{1}{5.255}}\right)$. Dans ces conditions, une variation de la pression de 1 hPa, correspond à une variation d'altitude de 10 m.

La bibliothèque fournit deux fonctions :

- `readAltitude(seaLevel)` où *seaLevel* est la pression ramenée au niveau de la mer ;
- `seaLevelForAltitude(altitude, pressionAtmospherique)` qui calcule la pression ramenée au niveau de la mer à partir de l'altitude du lieu et de la pression atmosphérique mesurée.

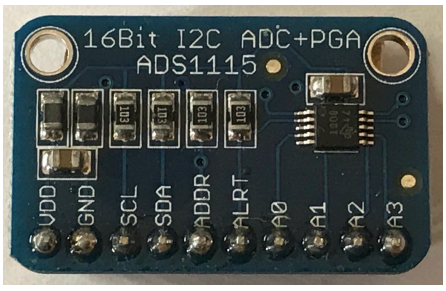
Et en fait... cette fonction donne plutôt de bon résultat. En montant 2 étages, on constate une diminution de la pression et une augmentation de l'altitude de 5 m.

Exercice 4 Une petite station météo
Nous vous proposons un exercice de synthèse : concevoir une station autonome permettant d'enregistrer, sur une journée par exemple température, pression et humidité atmosphérique. Procédons dans l'ordre suivant :

1. Câbler sur une carte Arduino (sans la platine ArdSSK) : le capteur BME280 (sur port I2C), l'horloge RTC (sur port I2C) ainsi que le module carte SD (sur port SPI).
2. S'inspirer des programmes précédents, écrire un programme stockant toutes les secondes sur carte SD les informations heure - température - pression - degré d'humidité. Vous utiliserez la solution développée dans le programme Alarme (page 83) pour la gestion du temps.
3. Si tout fonctionne bien sur 1 seconde, vous pouvez passer à un enregistrement toutes les minutes (utilisez pour cela l'instruction `lock.setAlarm2(0, 0, 0, 0, DS3231_EVERY_MINUTE)` en vous reportant au programme DS3231_alarm pour plus de précisions).
4. Si vous disposez d'une batterie rechargeable de 9 V (ou si vous sacrifiez une pile !), construire une station météo enregistrant les données toutes les 10 minutes (ou autre intervalle de temps). Laisser le programme tourner pendant une journée puis... récupérer et tracer les courbes.

3.4 Un convertisseur analogique-numérique

L'**ADS1115** est un convertisseur analogique-numérique fonctionnant sur bus I2C. Un de ces avantages principaux est de pouvoir mesurer une différence de potentiel entre deux broches... ce qui permet de mesurer des tensions négatives. On peut ainsi aisément traiter l'acquisition du potentiel d'une électrode de mesure par rapport à une électrode de référence.



Les principales caractéristiques sont les suivantes (cf datasheet) :

- conversion possible sur 16 bits ;
- programmation du gain ce qui permet de traiter des signaux entre ± 256 mV et ± 6,144 V ;
- fréquence d'échantillonage variable, en fonction de la résolution souhaitée entre 8 et 860 acquisitions par seconde.

L'utilisation de ce capteur est bien expliquée sur le site d'`Adafruit`. Nous avons, bien sûr, inclus la bibliothèque correspondante dans le dossier `libraries` de l'Arduino.

Trois programmes sont proposés dans l'IDE d'Arduino dans cette bibliothèque. On y accède via le lien : `Fichier > Exemple > ADS1x15 > ...`.

- `singleended` : comme son nom l'indique on peut mesurer de 1 à 4 tensions (positives) par rapport à la masse commune dans ce cas ;
- `differential` : on peut alors mesurer la différence de potentiel entre A0 - A1 (et/ou A3 - A2)
- `comparator` : on fixe une tension de référence et lorsque cette tension est dépassée (sur un des ports spécifiés), la broche ALRT passe à un niveau haut.

Comme sur l'Arduino, les entrées ne sont pas protégées contre d'éventuelles surtension ! L'utilisation de ces programmes nécessite un tout petit peu d'attention car la bibliothèque peut être utilisée pour deux capteurs différents. Le programme `CAN` ci-dessous présente les instructions minimales.

- On importe les bibliothèques nécessaires lignes 1 et 2.
- Ligne 4, on précise que l'on utilise l'ADS1115 et non son petit frère l'ADS1015 !
- Ligne 5, on stocke dans la variable *multiplier* le coefficient multiplicateur permettant la conversion entre la valeur lue (entier sur 16 bits) et la tension effectivement mesurée.
- Ligne 9, on précise le gain. Ici cette instruction est inutile puisqu'il s'agit du gain par défaut `GAIN_TWOTHIRDS`. Les différents gains possibles et coefficients multiplicateurs associés sont donnés dans les commentaires sur les programmes fournis par la bibliothèque.
- Ligne 15 : on effectue la lecture sur un seul port grâce à l'instruction `ads.readADC_SingleEnded(numéro du port)`. On récupère un entier sur 16 bits que l'on convertit en tension ligne 17.
- Ligne 16 : cette fois on fait l'acquisition d'une différence entre la lecture sur le port 0 et la lecture sur le port 1.

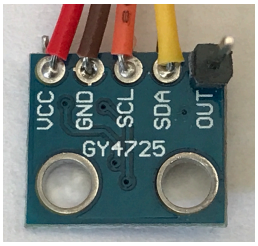
Listing 5.14 – Programme CAN

```
1  #include <Wire.h>
2  #include <Adafruit_ADS1015.h>
3
4  Adafruit_ADS1115 ads;  /* Use this for the 16-bit version */
5  float multiplier = 0.1875 ;
6  void setup(void)
7  {
8    Serial.begin(9600);
9    ads.setGain(GAIN_TWOTHIRDS);  // 2/3x gain +/- 6.144V  1 bit
       = 0.1875mV (default)
10   ads.begin();
11 }
12
13 void loop(void)
14 {
15   int adc2 = ads.readADC_SingleEnded(2);
16   int adc01 = ads.readADC_Differential_0_1();
17   Serial.println("Sur A2 : " + String(adc2) + " soit : " +
       String(multiplier*adc2) + " mV") ;
18   Serial.println("A0 - A1 : " + String(adc01) + " soit : " +
       String(multiplier*adc01) + " mV") ;
19   delay(1000);
20 }
```

Vous pouvez tester, par exemple avec une pile (pas 9 V !) :

- soit en connectant le pôle (-) de la pile à la masse de l'Arduino et le pôle (+) à la broche A2 du module ;
- soit en connectant l'un des pôles de la pile à la broche A0 et l'autre à la broche A1, ce qui permet de vérifier que l'on peut effectivement mesurer une différence de potentiel.

3.5 Un convertisseur numérique-analogique



Ce module permet la conversion d'un signal numérique en analogique (DAC en anglais). Il se connecte sur le port I2C.
Par programmation on va spécifier la valeur d'un entier x sur 12 bits (entre 0 et 4095 donc).
En sortie sur le port `OUT` on aura une tension (par rapport à la masse du système) de $\frac{x \times 5000}{4095} \approx x$ mV.

Il n'existe pas sur l'Arduino Uno de convertisseur analogique numérique. D'autres Arduino en possède. Le module proposé est le **MCP4725**. D'après le datasheet, on peut atteindre une fréquence de 400 kHz.

Vous trouverez quelques informations supplémentaires sur le site d'Adafruit. La bibliothèque correspondante a été installé et vous pouvez accéder aux exemples proposés via le chemin `Fichier > Exemples > Adafruit MCP4725`. Comme il s'agit du capteur MCP4725A1, il faut bien préciser que son adresse est `0x62` lors de l'initialisation.

Le module ne peut délivrer plus de 25 mA en sortie !
Le programme `CNA` propose d'imposer une tension sur sa sortie OUT et de lire cette tension à l'aide du CAN précédent.

Listing 5.15 – Programme CNA

```
1  #include <Wire.h>
2  #include <Adafruit_MCP4725.h>
3  #include <Adafruit_ADS1015.h>
4
5  Adafruit_MCP4725 dac;
6  Adafruit_ADS1115 ads;
7  float multiplier = 0.1875 ;
8
9  void setup(void)
10 {
11   Serial.begin(9600);
12   ads.setGain(GAIN_TWOTHIRDS);  // 2/3x gain +/- 6.144V  1 bit
        = 0.1875mV (default)
13   ads.begin();
14   dac.begin(0x62);
15 }
16
17 void loop(void)
18 {
19   int adc2 ;
20   for (int x = 0 ; x <= 4000 ; x += 100)
21   {
22       dac.setVoltage(x, false);
23       adc2 = ads.readADC_SingleEnded(2);
```

```
24        Serial.println("x : "+ String(x) + " Uimpose : " +
              String(x*5000.0/4095) + " Ulu : " +
              String(adc2*multiplier)) ;
25        delay(1000) ;
26    }
27 }
```

Connectez les 2 modules sur le port I2C ainsi que la sortie OUT sur la broche A2 de l'ADS115. On voit alors sur le moniteur série un relativement bon accord entre les valeurs. La précision des convertisseurs intervient mais, vraisemblablement, plutôt le fait que la tension de référence sur l'Arduino n'est pas « extrêmement » stable.
Pour les deux exercices suivants :

- si on dispose d'un oscillo, il suffit de connecter la masse et la sortie `OUT` du convertisseur analogique numérique pour capter le signal (aucun affichage n'est alors nécessaire) ;
- sinon, prendre comme modèle, par exemple, le programme ADC_ModeleNoTimer du chapitre 3 afin de visualiser le signal de sortie à l'aide du programme TraceQt. On fera l'acquisition sur un des ports analogiques de l'Arduino et on choisira un débit de 115200 bauds.

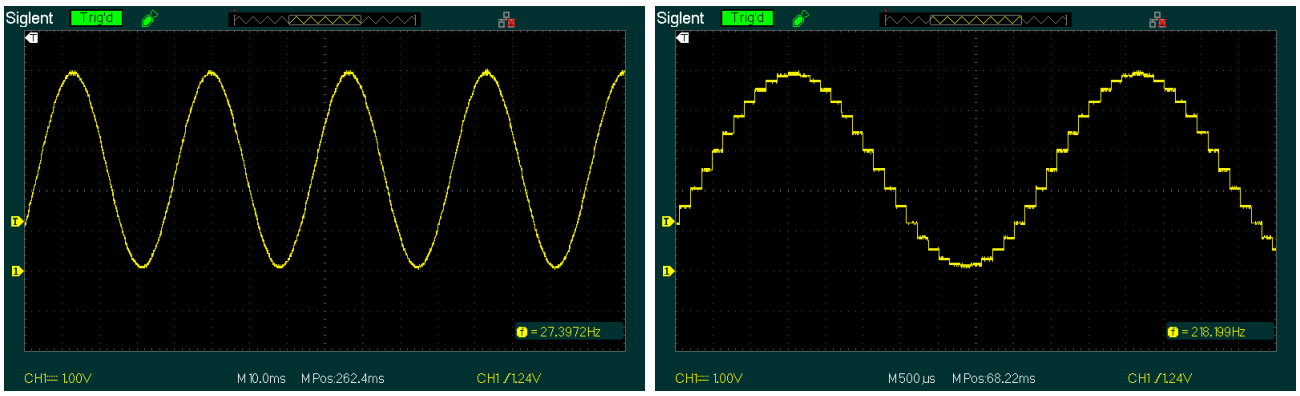
Exercice 5 Signal triangulaire
Sans se soucier de sa fréquence, créer un programme sur l'Arduino qui génère un signal triangulaire.

Exercice 6 Générateur de signal
Modifier le programme précédent afin de :

- générer un signal triangulaire si on envoie l'ordre 'P :0' :
- générer un signal sinusoïdal si on envoie l'ordre 'P :1' ;
- générer un signal carré si on envoie l'ordre 'P :2' ;
- gérer la fréquence si on envoie l'ordre 'F :valeur' ;
- faire une acquisition si on envoie l'ordre 'A :1' et pas d'acquisition si 'A :0'.

Remarque : pour générer un signal triangulaire de fréquence f d'amplitude 1, on peut utiliser la fonction $2 * abs\left(f \times t - int\left(f * t + \frac{1}{2}\right)\right)$. La documentation de l'Arduino précise que l'on ne peut pas appeler la valeur absolue d'une fonction. Le plus simple est de tester le signe de la grandeur dans la grande parenthèse et de prendre soit cette valeur, soit l'opposé (sans appeler la fonction abs).
Une solution élégante pour atteindre une fréquence plus élevée consiste à précalculer la fonction que l'on souhaite tracer. Un exemple est proposé dans la bibliothèque de l'IDE d'Arduino : `Fichier > Exemples > Adafruit MCP4725 > sinewave`. Des tableaux sont stockés dans une zone particulière de la mémoire de l'Arduino lors de la compilation. Ils contiennent, ici la valeur de la fonction sinus discrétisée sur 512, 256, 128, 64 ou 32 valeurs.

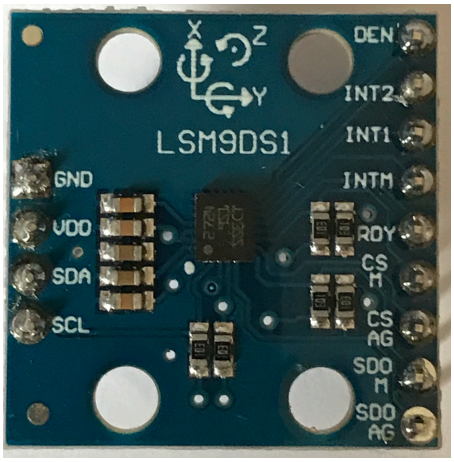
Ici à chaque boucle, on incrémente un compteur afin de récupérer la valeur du sinus et on impose cette valeur au convertisseur.



A gauche, un sinus sur 256 valeurs (fréquence de 27 Hz) et à droite sur 32 valeurs (218 Hz).

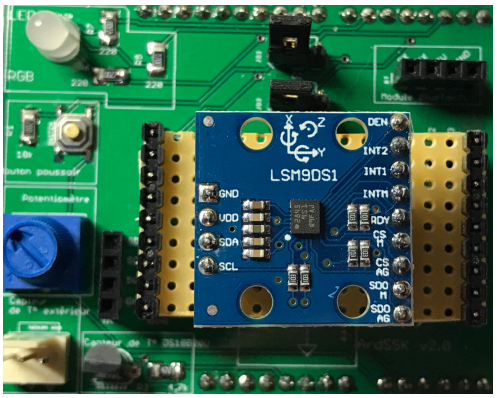
3.6 Last but not least : un accéléromètre !

Le module en question s'appelle : **LSM9DS1** : il s'agit d'un accéléromètre-gyroscope-magnétomètre.



Ce module peut se connecter soit sur le port I2C, soit sur le port SPI si on souhaite un traitement plus rapide du transfert des données du module vers l'Arduino.

C'est un capteur dit à 9 degrés de liberté (9DOF) qui mesure l'accélération sur 3 axes, la rotation sur 3 axes et le champ magnétique sur 3 axes. Le trièdre xyz est représenté sur le circuit.



Le plus dur est, en fait, d'immobiliser le capteur avec, par exemple, l'axe z vertical !
Une solution consiste à fixer le capteur sur une mini-carte qui vient se clipser sur notre kit.

a Présentation du module

Commençons par présenter les instruction minimales pour exploiter ce module, une fois câblé sur le port I2C. Ouvrir le programme `Accelerometre` dans le dossier du chapitre 5. Téléverser l'exécutable sur la carte et ouvrir le moniteur série.

Listing 5.16 – Programme Accelerometre

```
1 #include <Wire.h>
2 #include <Adafruit_LSM9DS1.h>
3 #include <Adafruit_Sensor.h>  // not used in this demo but
    required!
4
5 Adafruit_LSM9DS1 lsm = Adafruit_LSM9DS1();//I2C
```

```
6
7  void setup()
8  {
9    Serial.begin(9600);
10   lsm.begin() ;
11   lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_2G);
12 }
13
14 void loop()
15 {
16   lsm.read();  /* ask it to read in the data */
17   sensors_event_t a, m, g, temp;/* Get a new sensor event */
18   lsm.getEvent(&a, &m, &g, &temp);
19   Serial.print("Accel (m/s^2) X: ");
       Serial.print(a.acceleration.x);
20   Serial.print("\tY: "); Serial.print(a.acceleration.y);
21   Serial.print("\tZ: "); Serial.println(a.acceleration.z);
22   Serial.println();
23   delay(1000);
24 }
```

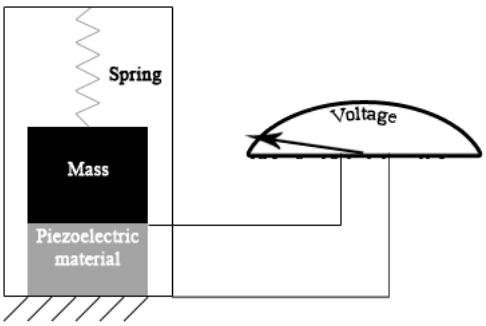
Le code est simple à comprendre maintenant... les résultats obtenus un peu moins.

- On instancie ligne 5 un objet *lsm* permettant de gérer le capteur. Son initialisation a lieu ligne 10 et on précise (valeur par défaut en fait) que l'on mesure une accélération inférieure à 2g (on peut aller jusqu'à 16 g avec ce capteur).
- Dans la boucle, on lit les données (ligne 16), que l'on récupère (ligne 18) entre autre dans la variable *a* (qui correspond à un type struct en C++) et que l'on exploite par la suite via *a.acceleration.x*, *a.acceleration.y* et *a.acceleration.z*.
- En sortie, on affiche le module de l'accélération ainsi que l'accélération sur les trois axes x, y, z spécifié sur le capteur.

On constate sur le moniteur série :

- une certaine fluctuation des valeurs (on pourrait aisément modifier le programme précédent afin d'afficher les valeurs une fois la moyenne prise sur 10 mesures par exemple (cf exercice suivant) ;
- un module de l'accélération proche de 9,6 $m.s^{-2}$... d'une part on s'attendait à 0 puisque notre système est au repos ; d'autre part pourquoi pas 9,8 ?

Une image extraite du site https ://learn.sparkfun.com/tutorials/accelerometer-basics, permet de présenter simplement le principe de la mesure de l'accélération sur un axe.



Le capteur est constitué d'un système masse-ressort. Le déplacement de la masse est mesuré à l'aide d'un capteur piézoélectrique.

Au repos, à cause de l'accélération de la pesanteur, le ressort est allongé ; après calibration, la tension délivrée par le piézo doit correspondre à l'accélération de la pesanteur. En chute libre, notre capteur indiquerait 0.

Calibrer le capteur est une tâche complexe que nous n'aborderons pas ici. Nous nous intéresserons principalement aux variations de ces grandeurs.

En fait, dans de très nombreux exemples, ne pas avoir une accélération nulle au repos n'est pas un inconvénient... Bien au contraire, connaitre la projection de l'accélération permet de déterminer l'orientation du module dans l'espace.

Exercice 7 Accéléromètre à zéro
Écrire un programme qui, dans le setup fait la moyenne sur 50 mesures de l'accélération (toutes les 100 ms) puis affiche l'accélération telle que celle-ci soit nulle quand le module est au repos.
Un exemple fournit avec la bibliothèque Adafruit : `Fichier > Exemple > Adafruit LSM9DS1 > lms9ds1` montre comment récupérer les données du magnétomètre et du gyroscope.

b Un dé magique !

Exercice 8 Dé magique ! Vous donnez à quelqu'un un dé. Vous êtes le magicien, sans regarder (bien sûr) comment le dé a été placé sur la table, vous devinez quelle est la valeur sur la face du dessus.

Vous aurez besoin pour cela de :

- modules LSM9DS1 et Bluetooth ;
- une batterie 9V ainsi que le connecteur afin d'avoir une alimentation autonome de la carte ;
- une petite boite (une des dimensions doit être supérieure à 10 cm, les autres à 8 cm) ;
- un bout de scotch pour fixer le module ;
- un peu de discrétion pour regarder votre smartphone si vous voulez faire carrière dans la magie ;
- un peu de réflexion pour assembler tout ça !

c Oui, c'est comme la WII

c.1 Inclinomètre Reprendre le programme Accelerometre (ou AccelerometreLisse) et s'arranger pour que le capteur soit bien horizontal, l'axe x parallèle au grand côté de l'Arduino par exemple.

Placer le capteur sur un plan incliné et relever l'accélération sur les axes x et z. Victoire... on vérifie bien $\sin(\alpha) = \dfrac{\text{accélération sur x}}{\text{module de l'accélération}}$!

c.2 Un petit bout de la WII Les formules se compliquent singulièrement si on incline l'accéléromètre dans toutes les directions.

Nous vous proposons un exemple de démonstration « clé en main » :

- charger le programme `WWI` sur l'Arduino (dossier Chapitre 5) ;
- ouvrir votre logiciel python préféré. S'il s'agit de Spyder/Pyzo, s'assurer dans les préférences du logiciel, onglet IPython console puis Graphics que l'on a bien Automatic comme backend (cf 2.6 page 106) ;
- ouvrir le programme `AcqAccelero.py` (dans le dossier Chapitre 5) et lancer l'exécution ;

- en bougeant le module LSM9DS1, vous devriez voir mise à jour l'orientation du capteur dans l'espace...
- taper a.stop() dans la console permet d'arrêter l'acquisition.

Reste à concevoir une voiture radio guidée !

d Étude d'un mouvement circulaire uniforme

Il s'agit d'utiliser notre capteur pour étudier un mouvement circulaire uniforme. Expérience « vide grenier » encore... avec un bon vieux tourne disque et un bout de scotch !

On se reportera au paragraphe 3.8 page 112 de l'annexe pour une présentation des résultats obtenus.

Activité 6

Annexes

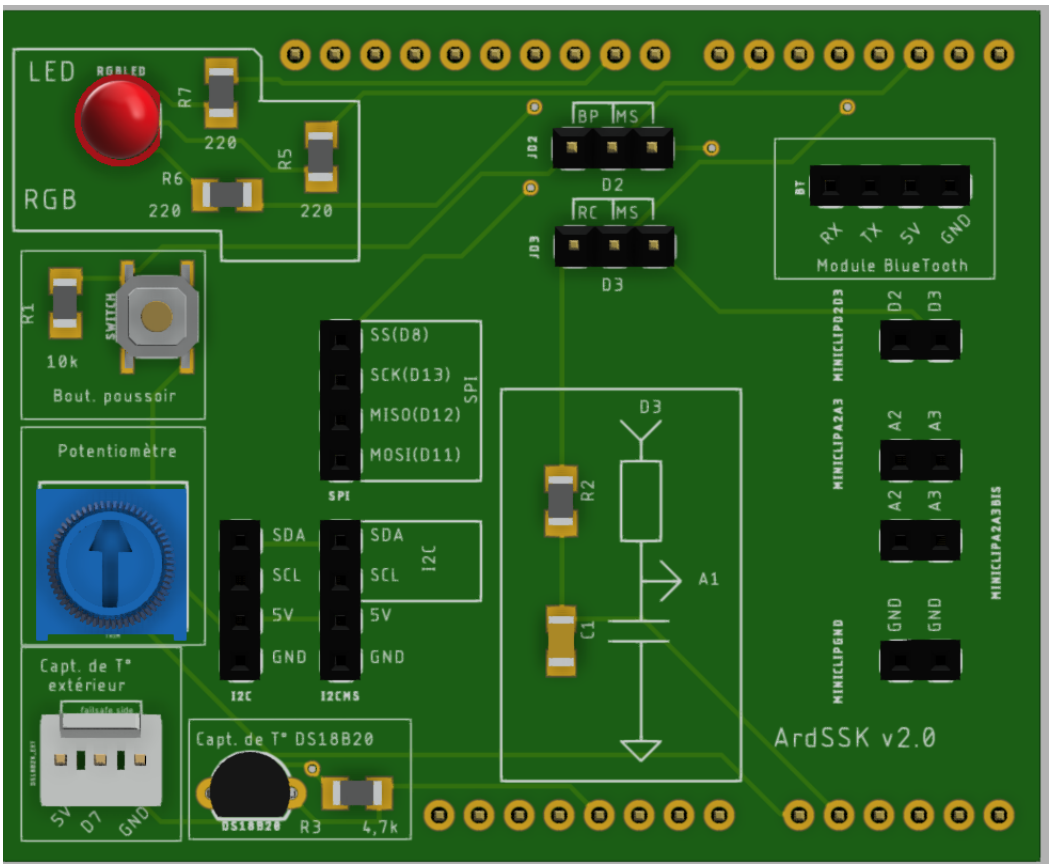
1 Arduino Scientific Starter Kit

Le kit que nous vous proposons contient un certain nombre de composants soudés sur un module (shield dans l'éco-système Arduino) qui s'insère sur la carte Arduino. Vous pouvez ainsi, sans avoir à vous soucier du câblage des composants, apprendre à utiliser l'Arduino ; c'est l'objet des chapitres 2 à 4.

Un certains nombres d'autres capteurs sont proposés ainsi que quelques composants pour vos premières réalisations. Le chapitre 5 y est consacré.

1.1 Le module ArdSSK

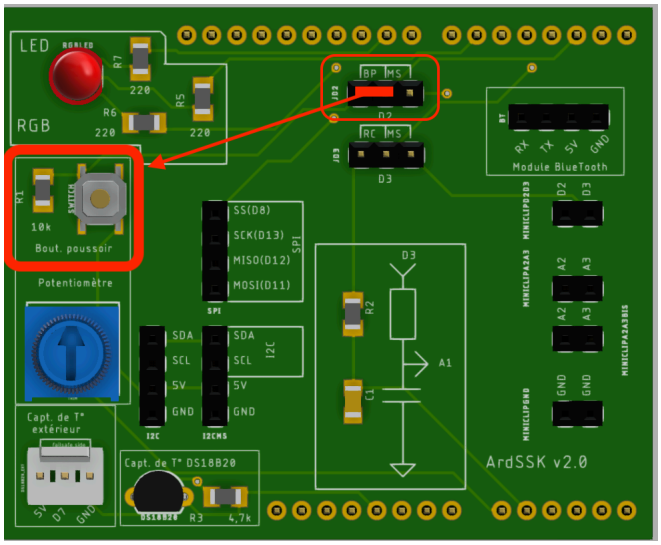
La carte nue est représentée ci-dessous.



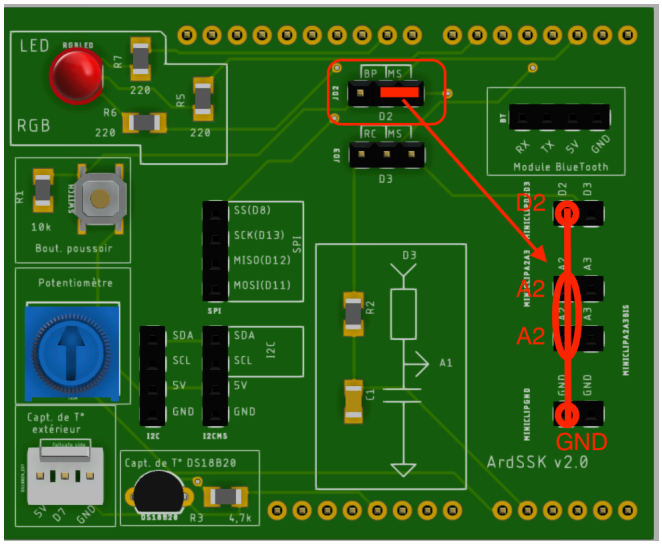
Les ports de la carte utilisés sont les suivants :

- ports analogiques :
  - **A0** mesure de la tension sur la sortie du potentiomètre ;

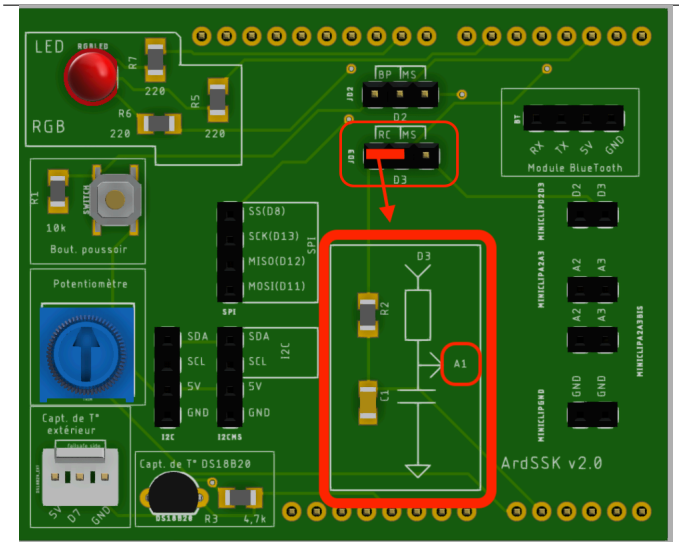
    - **A1** mesure de la tension aux bornes du condensateur dans le circuit RC ;
    - **A2** mesure de la tension au point milieu du diviseur de tension (non câblé) commandé par **D2** ;
    - **A3** mesure de la tension au point milieu du diviseur de tension (non câblé) commandé par **D3**.
- ports numériques :
    - **D0** et **D1**, respectivement RX et TX du port série (comme pour toute carte Arduino) ;
    - **D2** un cavalier permet :
        - soit de tester l'état appuyé ou non du bouton poussoir ;
        - soit de commander le diviseur de tension **D2-A2-GND**.
    - **D3** un cavalier permet :
        - soit de commander la charge du circuit RC ;
        - soit de commander le diviseur de tension **D3-A3-GND**.
    - **D4** et **D5**, respectivement TX et RX du port série utilisé par le protocole `SoftSerial`, par exemple pour la connexion du module Bluetooth (TX de l'arduino relié à RX du module et RX de l'arduino à TX du module) ;
    - **D6**, **D9** et **D10** utilisés respectivement pour les broches R, G et B de la LED RGB ;
    - **D8**, **D11**, **D12**, **D13** disponibles sur le port « minishield », correspondent respectivement aux signaux CS, MOSI, MISO et SCK du port SPI.



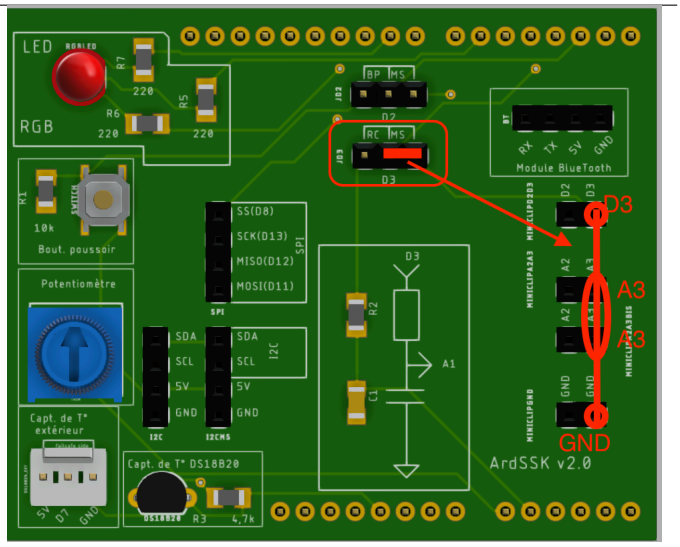
Cavalier D2 vers BP



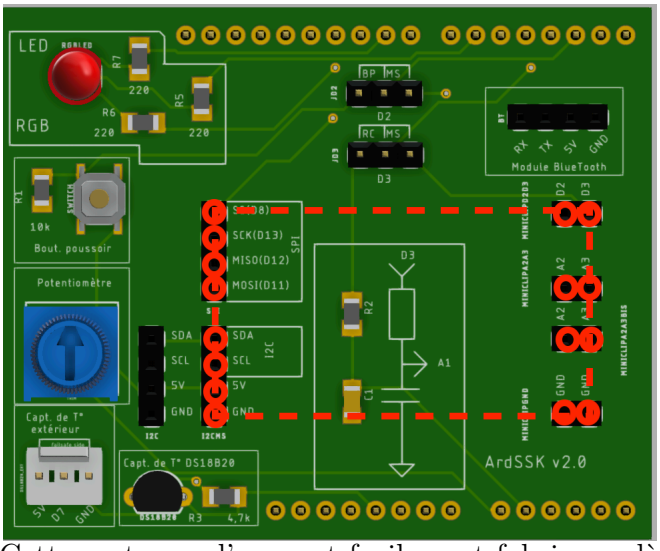
Cavalier D2 vers MS



Cavalier D3 vers RC



Cavalier D3 vers MS



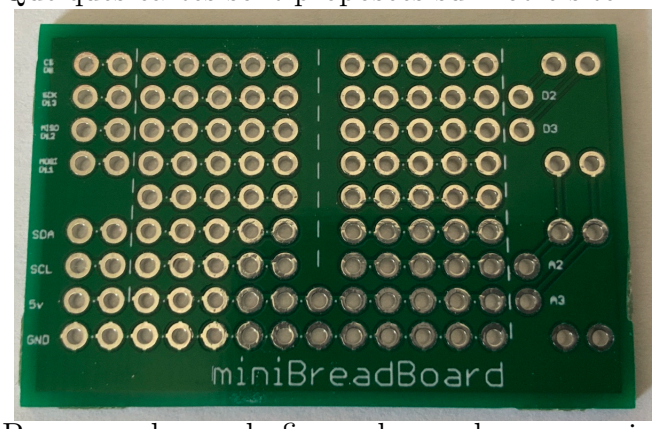
Le module que nous avons conçu permet d'insérer une « mini-carte » supplémentaire grâce aux connecteurs mis en évidence sur la figure ci-contre. Cette carte peut ainsi accéder aux ports GND, 5V, A2, A3, D2, D3 ainsi qu'au port I2C et soit au port SPI, soit aux ports logiques D8, D11, D12, 13.

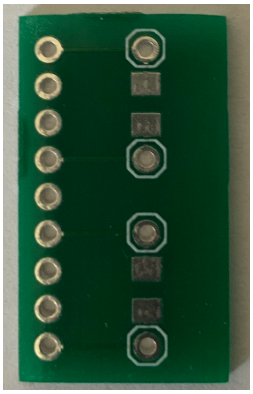
Cette carte que l'on peut facilement fabriquer dès lors qu'on possède un fer à souder et un peu de matériel permet, par exemple :

- réaliser un prototype, station météo par exemple (cf page 136 où la réalisation d'un « mini shield » est décrite) ;
- de fixer un capteur afin d'éviter qu'il ne bouge trop (accéléromètre par exemple)(cf page 92) ;

- proposer en TP une solution stable évitant tant que faire se peut de fausses manœuvres sur le système (cf page 73 ou 72) (ce qui ne peut que contribuer à la pérennité d'un tel investissement).

Quelques cartes sont proposées sur notre site.





Par exemple, sur la figure de gauche une « mini platine d'expérimentation » peut-être câblée et venir s'insérer sur notre carte principale.
Sur la figure de droite un « mini clip » est proposé. Vous pourrez ainsi souder (en traversant ou en cms) sur cette carte le pont diviseur de tension de votre choix et venir le clipser sur les connecteurs D3-A3-A3-GND en bas à droite de la carte ArdSSK.

1.2 Le contenu du kit

Le kit que nous proposons s'articule autour :

- une carte Arduino Uno rev 3 + câble USB ;
- un module ArdSSK (carte représentée ci-dessus) sur lequel est soudé :
  - une LED RGB et ses résistances de protection ;
  - un bouton poussoir et sa résistance pull up ;
  - un filtre RC ;
  - un potentiomètre ;
  - un capteur de température DS18B20 ;
  - deux bus I2C, un bus SPI et deux diviseurs de tension.

a Capteurs et composants

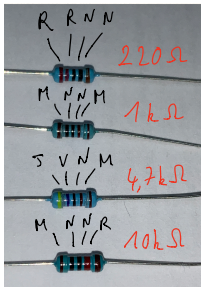
Peuvent ensuite se connecter les capteurs et composants suivants :

- un buzzer (cf 2.1 page 71) ;
- une photorésistance LDR (cf a page 72) et une thermistance (cf b page 73) ;
- un capteur à effet Hall : AH503 (cf 2.3 page 75) ;
- un capteur de température DS18B20 étanche ainsi qu'un second capteur nu ;
- un écran OLED 128x32 ;
- un capteur de distance à ultrason : HC-SR04 (cf 2.4 page 76) ;
- un convertisseur analogique numérique (4 entrées) : ADS1115 (cf 3.4 page 88) ;
- un convertisseur numérique analogique : MCP4725 (cf 3.5 page 90) ;
- un capteur température-humidité-pression : BME280 (cf 3.3 page 87) ;
- un accéléromètre, gyroscope, magnétomètre : LSM9DS1 (cf 3.6 page 92) ;
- une horloge temps réel (pile bouton fournie) (cf 3.1 page 81) ;
- une lecteur de carte SD (carte micro SD fournie) (cf 3.2 page 84) ;
- un module BlueTooth.

b Petit matériel

Afin de pouvoir câbler certains montages, nous avons ajouté :

- une plaque de prototypage ;
- quelques fils de connection ;
- une LED RGB, un potentiomètre ;
- quelques résistances (R, M, N, J, V pour rouge (=2), marron (=1), noir (=0), jaune (=4), violet(=7)) et condensateurs.



1.3 Les programmes

a Programmes Arduino

Vous trouverez dans le dossier `Programmes`, regroupés par chapitre, les quelques quatre vingt programmes présentés dans ce guide.

Quelques programmes python (sans interface graphique) sont également proposés.

b Programmes python avec interface graphique

Ils sont regroupés dans le dossier `Interfaces_Qt`. Il s'agit des programmes :

- LEDRGBQt (script python à utiliser LEDRGBQt.py) ;
- PyToArdQt (script python à utiliser PyToArdQt.py) ;
- TraceQt (script python à utiliser TraceQt.py).

Dans chacun de ces dossiers se trouve le fichier ...Qt.py qu'il faut ouvrir dans l'environnement python et exécuter.

2 Programmation de l'Arduino

2.1 Commandes des Entrées/Sorties

a Affection des Entrées/Sorties

Les affectations des entrées/sorties sont à placer dans la procédure *setup*. Pour affecter une entrée sur une broche :

```
pinMode(8,INPUT) //Affectation de la broche 8 (logique) en tant qu'entrée.
pinMode(A0,INPUT) //Affectation de la broche A0 (analogique) en tant qu'entrée.
pinMode(8,OUTPUT) //Affectation de la broche 8 (logique) en tant que sortie.
pinMode(9,OUTPUT) //Affectation de la broche 9 (sortie PWM car marquée par le symbole
//   sur la carte) en tant que sortie.
```

b Lire les entrées

Lorsqu'une broche "digitale" ou logique est configurée en entrée, il est possible de récupérer la valeur correspondante en entrée de cette broche via : `digitalRead(numero_de_broche)`

Cette fonction renvoie 1 si la tension en entrée de la broche est supérieur à 3V, et 0 si la tension est inférieure.
Il est possible de lire une tension sur une broche analogique, via un convertisseur analogique/numérique, en appelant la fonction :analogRead(numero_de_broche)

La tension lue doit varier entre 0 et 5V. La valeur renvoyée est un entier contenu entre 0 et 1023.

c Imposer une tension en sortie

Sur une broche logique configurée en sortie, on peut imposer soit 0V soit 5V via :
digitalWrite(numero_de_broche,LOW) //impose 0V
digitalWrite(numero_de_broche,HIGH)// impose 5V

Certaines broches spéciales configurées en sortie, et suivies du symbole ~, peuvent délivrer une tension **moyenne** variable grâce à la fonction :

```
analogWrite(numero_de_broche,val)
```

avec *val* une valeur entière comprise entre 0 et 255. Si val=0, la valeur moyenne vaut 0V, si val=255, la valeur moyenne vaut 5V. La sortie est obtenue par modulation de largeur d'impulsion (MLI ou PWM en anglais). Son étude sera réalisée plus tard.

2.2 Quelques instructions propres au langage Arduino

a Délai

L'instruction `delay(ms)` permet d'exécuter une pause de *ms* millisecondes.
L'instruction `delayMicroseconds(mus)` permet d'éxécuter une pause de *mus* microsecondes.

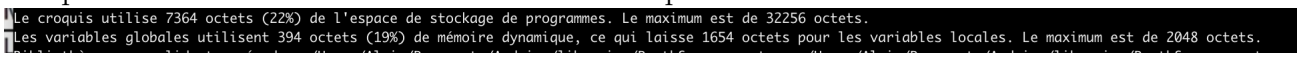
2.3 Quelques éléments de langage C/C++

a Les différents types et les précautions qui vont avec

Contrairement à Python, Arduino requiert qu'on lui renseigne un type à chaque variable créée.
Il faut être vigilant sur le type utilisé en essayant de limiter l'occupation en mémoire. En effet, question mémoire, on se trouve avec un Arduino comme avec un ordinateur d'il y a une quarantaine d'année ! L'Arduino dispose :

- de 32 ko de mémoire flash, celle-ci sert à stocker les programmes à exécuter ; elle perdure après arrêt de l'alimentation ;
- de 2 ko de mémoire volatile pour stocker les données temporaires, en particulier, les variables de votre programme.

L'occupation mémoire apparait en bas de la fenêtre de l'IDE d'Arduino... avec une indication lorsque la taille de la mémoire vive devient critique.



Les types les plus utilisés sont regroupés dans le tableau suivant :

Type de variable	Intervalle	Commentaires
boolean	*True* ou *False*	Occupe 1 octet en mémoire.
char	Caractère ASCII	Occupe 1 octet en mémoire.
String	mots	Occupe 1 octet/caractère.
byte	[0 ;255]	Représentation partielle de $\mathbb{N}^+$ Occupe 1 octet en mémoire.
int	[-32,768 ;32,767]	Représentation partielle de $\mathbb{N}$ Occupe 2 octets en mémoire.
unsigned int	[0 ;65535]	Représentation partielle de $\mathbb{N}^+$ Occupe 2 octets en mémoire.
long	[-2 147 483 648 ;2 147 483 647]	Représentation partielle de $\mathbb{N}$ Occupe 4 octets en mémoire.
unsigned long	[0 ;4 294 967 295]	Représentation partielle de $\mathbb{N}^+$ Occupe 4 octets en mémoire.
float	$[-3,4028235.10^{38};3,4028235.10^{38}]$	Représentation partielle de $\mathbb{R}$ Occupe 4 octets en mémoire.

b Eléments de syntaxe

b.1 Point virgule en fin de ligne

En C/C++ chaque instruction doit se terminer par un " ;" !

b.2 Des parenthèses pour délimiter un bloc de code

En python on utilise des indentations pour identifier les blocs de code (boucle, fonction, condition...), ici le bloc de code sera délimité par une parenthèse ouvrante et une fermante.

b.3 Boucle *for*

```
1 for(int i=0;i<=10; i = i + 1)
2 {
3     //corps de la boucle
4 }
5
6 for(int i=10;i>=0; i = i - 1)
7 {
8     //corps de la boucle
9 }
```

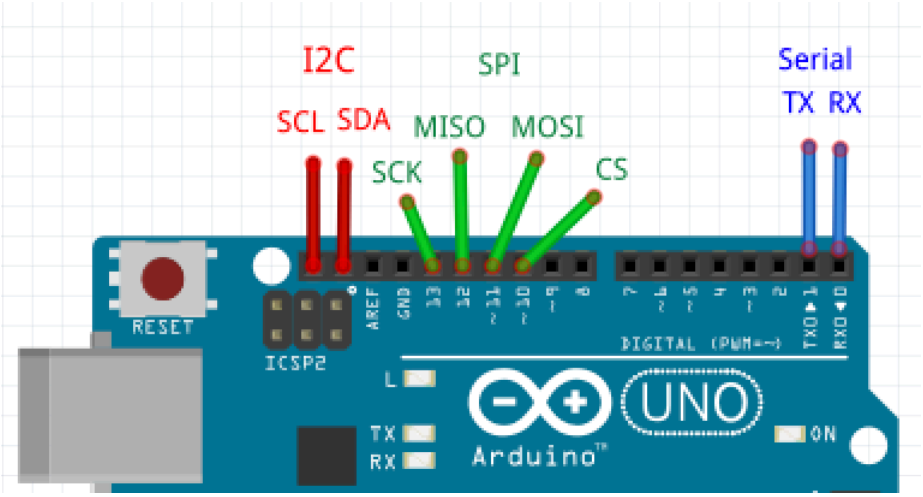
b.4 Boucle *while*

```
1  int i=10;
2  while(i>=0)
3  {
4      //corps de la boucle
5      i--;
6  }
```

b.5 Structure conditionnelle *if*

```
1  int a=0;
2  int b=2;
3  if(a==b)
4  {
5      Serial.println("a=b");
6  }
7  else if(a>b)
8  {
9      Serial.println("a>b");
10 }
11 else
12 {
13     Serial.println("a<b");
14 }
```

2.4 Protocoles de communication



a Serial et SoftSerial

La communication série entre votre ordinateur et la carte Arduino reprend, en fait, les ports 0 (RX) et 1 (TX) de l'Arduino. Il faut éviter d'utiliser ces ports pour autre chose !

La communication série peut aussi être émulée (`SoftSerial`). Deux ports numériques sont nécessaires et peuvent être choisis à votre guise (il suffit de les préciser dans l'initialisation de la liaison série). Sur la carte, nous avons utilisé les ports 4 (TX) et 5 (RX) pour communiquer avec le module Bluetooth.

La communication entre 2 cartes Arduino est également possible (RX de l'un vers TX de l'autre et vice versa), il faudra toutefois relier les masses entre elles.

b Bus I2C

Un certain nombre de capteurs utilisent le bus I2C pour communiquer avec l'arduino. 4 fils sont nécessaires : un premier pour la masse, un second pour l'alimentation 5 V (ou 3,3 V) ainsi qu'un pour l'horloge de synchronisation (SCL) et un dernier pour les données (SDA).

Les ports SCL et SDA sur l'Arduino sont indiqués sur la figure ci-dessus et sont immuables. Par contre, malheureusement, côté capteur, chaque fabricant choisi son propre cablage ; on ne peut donc envisager un connecteur IC2 enfichable compatible avec tous les capteurs.

Plusieurs périphériques IC2 peuvent se brancher en parallèle ainsi. Chacun possède son propre identificateur. Quelques connexions supplémentaires permettent de modifier ces identificateurs afin d'avoir plusieurs capteurs de même nature.

La vitesse de communication du bus I2C peut atteindre 200 kHz (fréquence maximale supportée par l'Arduino).

c Bus SPI

Le bus SPI permet des transferts plus rapides mais le nécessite 6 fils :

- la masse et l'alimentation ;
- l'horloge de synchronisation (port 13) ;
- l'échange de données MISO (port 12) pour les données entrantes et MOSI (port 11) pour les données sortantes ;

Différents périphériques SPI partagent ces 5 ports.
Le sixième sert d'identificateur « ChipSelect » CS : chaque périphérique SPI devra avoir sa propre ligne CS. S'il n'y en a qu'un, le port 10 est souvent utilisé. Sur notre carte, nous avons utilisé le port 8.

d Bus OneWire

Comme son nom l'indique... deux fils sont nécessaires la masse et un autre par lequel transitent toutes les informations. On est libre de choisir le port numérique que l'on veut, il suffit de le préciser.

2.5 Bibliothèque ArdTools

Cette bibliothèque mise à votre disposition permet de concevoir, à partir d'un programme en python (ou de la console), une interface entre la carte Arduino et l'ordinateur.

Dans ce document on importe la bibliothèque sous forme d'alias *ard* de la façon suivante :

Listing 6.1 – Programme FeuxTricolore0

```
1 import ArdTools.ArdTools as ard
```

Trois types d'objets (des classes) vous sont proposés.

a Arduino

Cette classe est utilisée pour envoyer un ordre à la carte Arduino comme on le ferait à partir du moniteur série.

a.1 Pour établir une connexion

Listing 6.2 – Etablir une connexion

```
1 a = ard.Arduino()
2
3 ports = a.ports()
4 print(ports)
5
6 a.connexion(ports[len(ports)-1], 9600)
```

Remarque : l'indice du port ligne 6 est à adapter selon le cas de figure !

a.2 Pour envoyer un ordre La fonction est **envoyer(ordre, feedback=False, timeout=2)** où *ordre* est une chaîne de caractère (charge au programme de l'Arduino de l'interpréter), *feedback* doit être mis à True si on attend une réponse de la carte (par exemple lorsque l'on envoie l'ordre 'I' pour récupérer l'information sur la carte), *timeout* est le temps d'attente (en seconde) de cette réponse.

a.3 Quelques raccourcis

Un raccourci est proposé pour les ordres que les principaux programmes doivent interpréter (cf a page 27) :

synonyme	envoyer(...)	action
go()	envoyer('G')	GO : on exécute en continu ou avec le délai précisé
one()	envoyer('O')	ONE : on exécute une seule fois le travail souhaité
stop()	envoyer('S')	STOP : on arrête
infos()	envoyer('I', True)	INFO : on récupère une information sur le programme
delay(valeur)	envoyer('D :valeur')	DELAI : on fixe le délai (en milliseconde) entre deux actions

b ArduinoAcq

Toutes les fonctions implémentées dans la classe Arduino sont présentes dans **ArduinoAcq**. Cette classe est destinée à l'acquisition de données sans représentation graphique.

- La fonction sauvegarde('nom fichier.txt') permet de sauvegarder les données acquises.
- La fonction go() ou envoyer('G') permet de gérer la réception des données provenant de la carte Arduino.

c ArduinoPlot

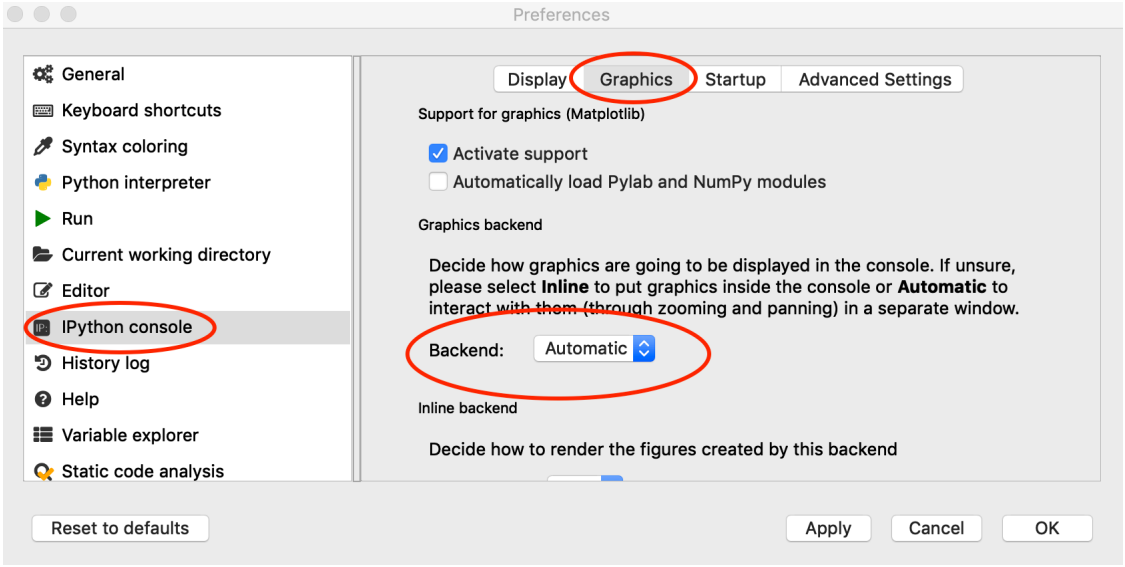
La classe **ArduinoPlot** est, quant à elle, destinée à une acquisition de données avec représentation graphique.

2.6 Quelques soucis avec Spyder !

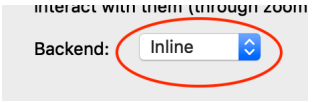
Quelques soucis peuvent se poser avec Spyder :

- sous macOS, lorsque l'on utilise TraceQt ou autre, cliquez pour fermer le fenêtre ne détruit pas l'application et celle-ci reste encore visible dans le dock ;
- si Spyder reste bloqué : il faut cliquer sur le carré rouge en haut à droite de la console et demander d'ouvrir une nouvelle console IPython ;
- si on utilise les programmes TraceQt, PyToArdQt, LEDRGBQt : il faut que la sortie de la console IPython soit Inline ;
- si on utilise les programmes ArduinoPlot, ArduinoAcq : il faut que la sortie de la console IPython soit Automatic !

Ce réglage se fait dans les préférences de Spyder :



Ou, pour une utilisation Inline :

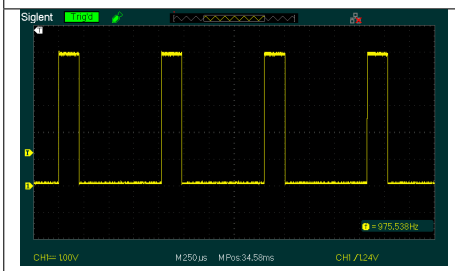
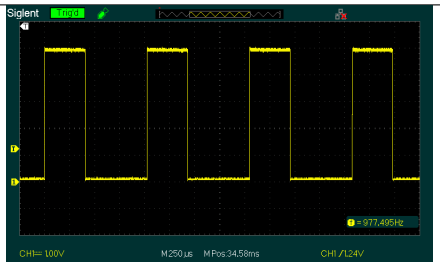
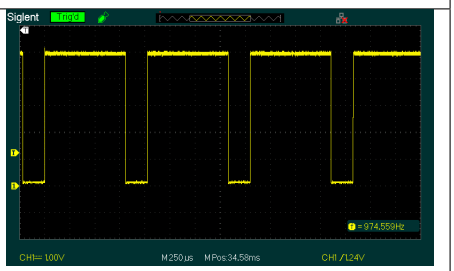


3 Expériences et TP

3.1 Quelques études à l'oscillo

a Sortie PWM

On connecte la masse de l'Arduino et la sortie **6** à un oscillo et on déterminer la fréquence et le rapport cyclique du signal PWM.

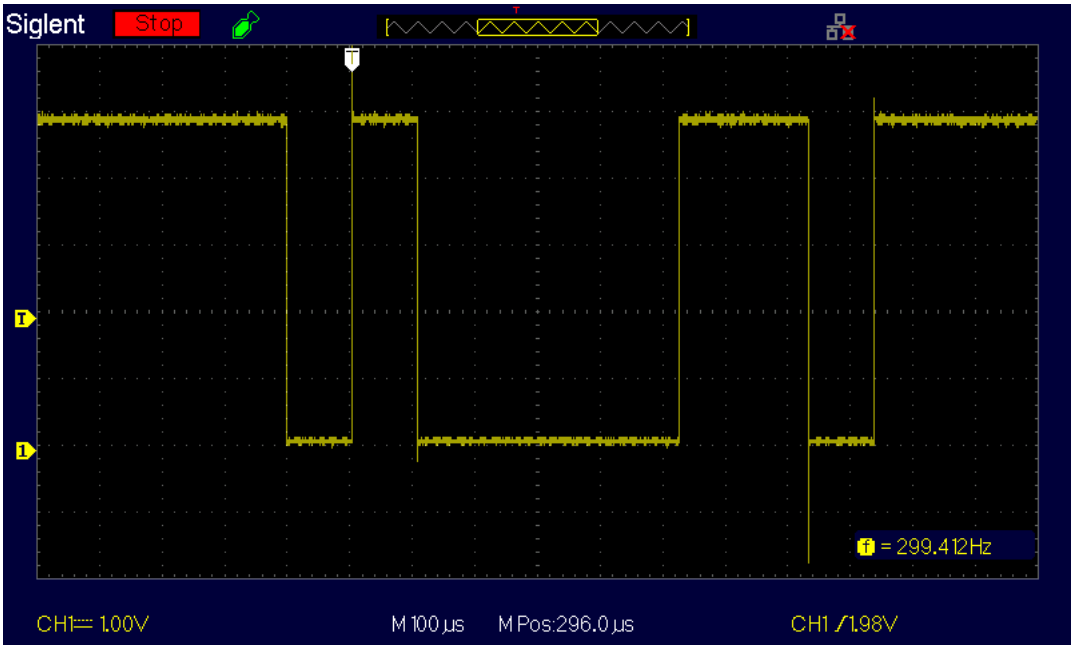
digitalWrite(port6, 50)	digitalWrite(port6, 100)	digitalWrite(port6, 200)
		

Sur le port **6** la fréquence du PWM est voisine de 980 Hz, reste à mesurer

b Étude d'une trame RS232

On communique, ici, à une vitesse de 9600 bauds, il faut donc 104 μs pour envoyer 1 bit.

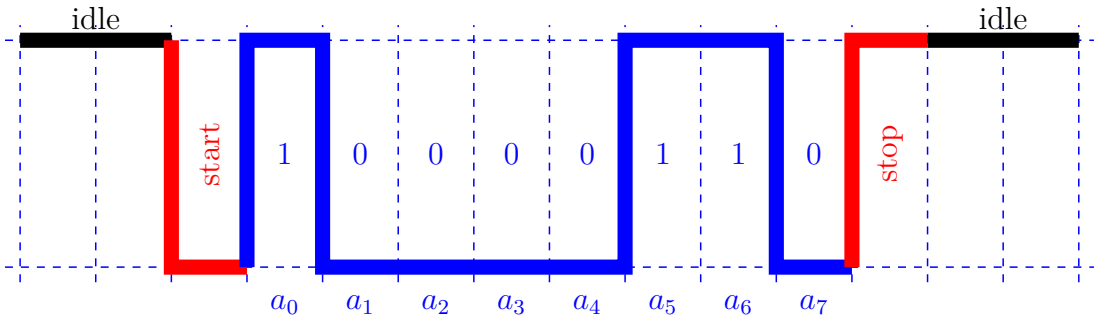
On enregistre, à l'aide d'un oscilloscope numérique, le signal émis sur le port **TX** lors de l'exécution de l'instruction `print('a')`. L'échelle de temps est réglée de telle sorte qu'un carreau corresponde à 100 μs donc environ à 1 bit.



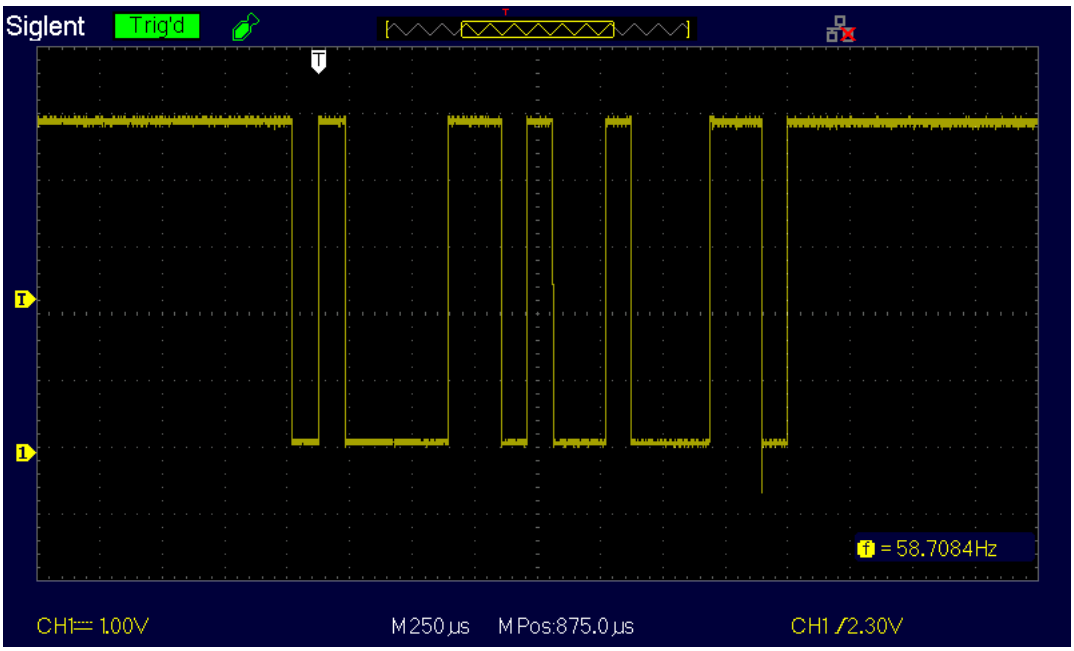
Le transfert sur le port série se fait à l'aide du protocole suivant :

- un niveau + 5 V lorsque aucune donnée n'est échangée (repos ou idle en anglais) ;
- un bit de start à 0 V indique le début d'envoi d'un octet ;
- 8 bits sont alors envoyés en commençant par le bit de poids faible ;
- vient alors un bit de stop à 5 V pour indiquer la fin de l'envoi d'un octet ;
- le niveau reste à + 5 V si aucun caractère n'est envoyé par la suite, sinon on a un nouveau bit de start.

Le signal ci-dessus peut donc être représenté de la façon suivante :



Les 8 bits de l'octet $a_7a_6a_5a_4a_3a_2a_1a_0$ sont transmis dans l'ordre inverse...on reçoit a_0 avant a_1 *etc.*... L'octet reçu correspond donc à `01100001` soit $2^6 + 2^5 + 1 = 64 + 32 + 1 = 97$. Le caractère dont le code ASCII est 97 est bien le caractère **a** !
Vous pouvez vérifier que la trame ci-dessous correspond à **ab**. Attention, on ne sait pas à l'avance combien d'octets vont être envoyés. Chaque octet est précédé d'un bit de start et suivi d'un bit de stop ; il faut donc échanger 10 bits pour communiquer un octet.



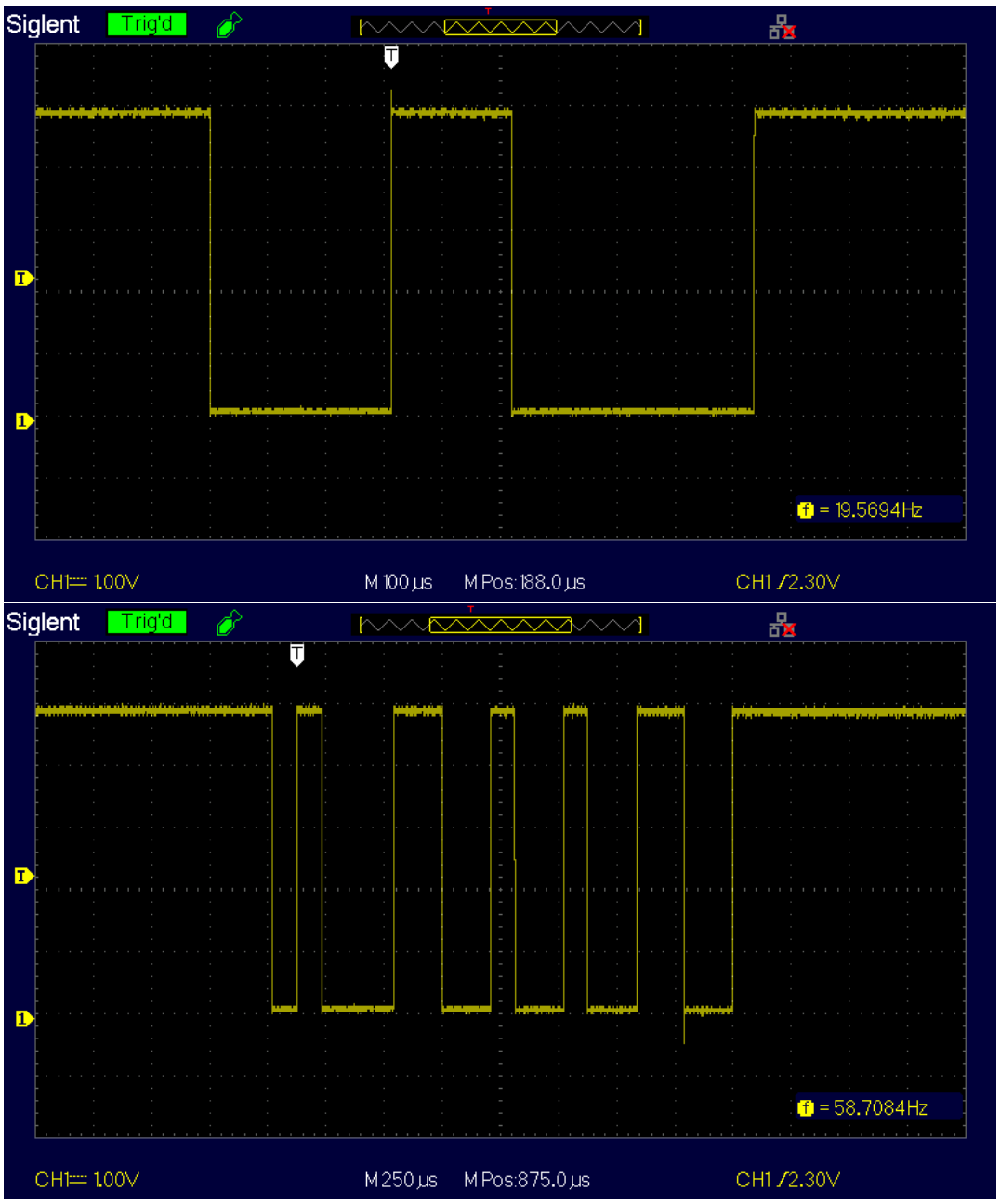
Exercice 1 Différence write et print

Considérons le programme suivant.

Listing 6.3 – Exercice write/print

```
1  void setup() {
2    Serial.begin(9600) ;
3  }
4
5  void loop() {
6    byte a = 12 ;# byte : entier < 255 !
7    Serial.write(a) ;
8    delay(100) ;
9    Serial.print("a") ;
10   delay(100) ;
11 }
```

Les trames reçues correspondent aux images ci-dessous :



Expliquer la différence.

3.2 Circuit RC

On se reportera au paragraphe 5 page 55 pour l'étude de la décharge d'un circuit RC et l'application à la réalisation d'un générateur de tension variable, voire d'un générateur de signal.

3.3 Calorimétrie

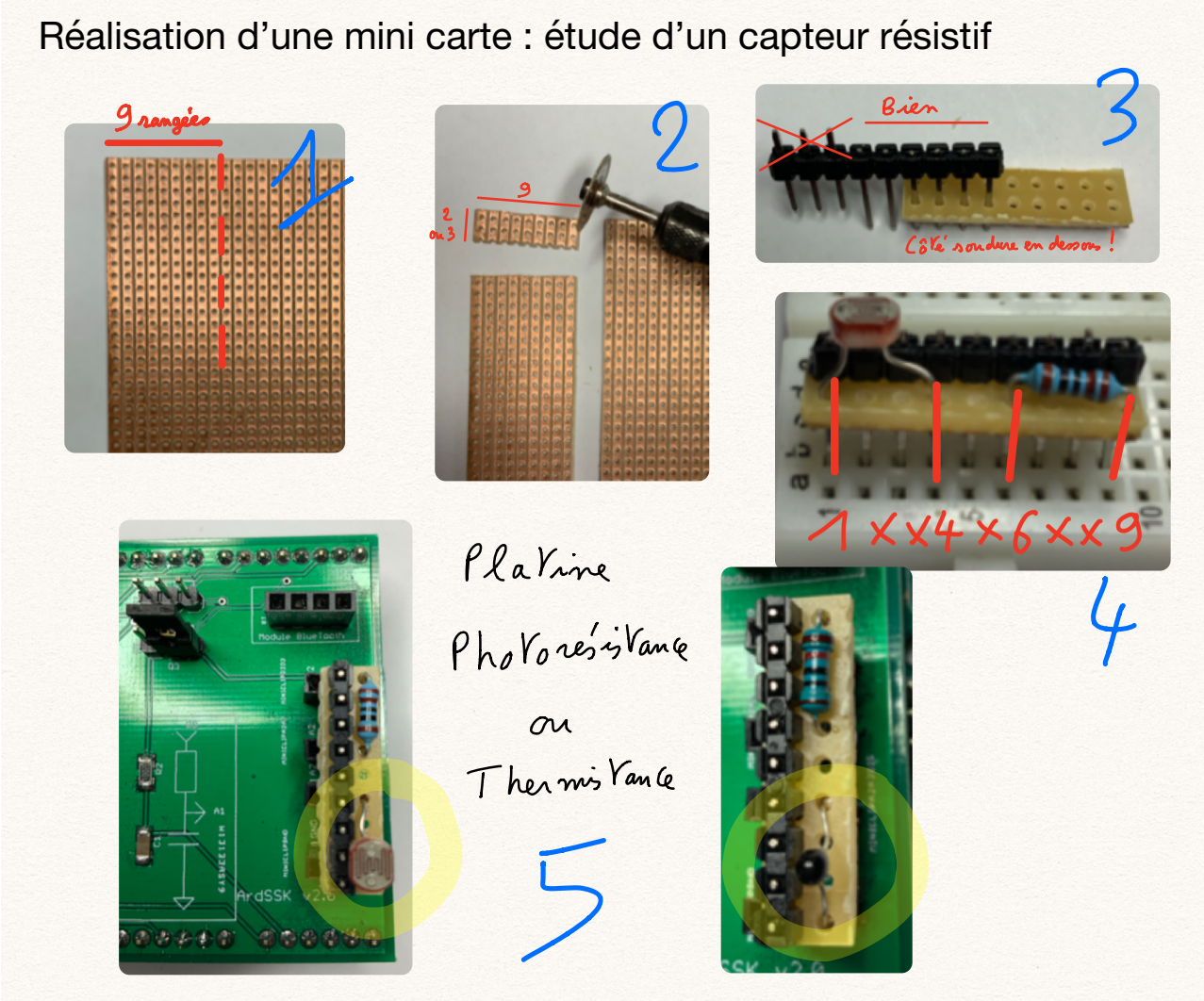
On se reportera au paragraphe 2 page 63 pour l'étude des effets thermiques associés à la dismutation de l'eau oxygénée.

3.4 Étude d'un capteur résistif

On se reportera page 72 pour l'étude d'une photorésistance ou page **??** pour l'étude d'une thermistance.

Notons que pour cette activité expérimentale proposée au programme de première, on peut relativement facilement fabriquer une « mini-carte » contenant uniquement la résistance de protection

et le capteur. Ainsi, la fiabilité du système est renforcée.



Quelques étapes simples permettent de créer une petite carte :

- prendre une plaque d'essai à pastilles ou à bandes et repérer 9 rangées ;
- découper à l'aide d'une petite scie (ou avec un bon cutter) un rectangle de 9 rangées sur 2 (ou 3) rangées (dans le sens de la photo) ;
- préparer un connecteur mâle 9 broches, les faire dépasser au maximum d'un côté (la partie en plastique du connecteur ne doit se trouver côté cuivre !) ;
- reste à souder le connecteur (soudures sur les rangées 1-4 et 6-9) ;
- reste à la clipser sur les connecteurs D3-A3-A3-GND.

3.5 Statique des fluides

La variation de la pression atmosphérique p en fonction de l'altitude h est décrite par l'équation de la statique des fluides : $\frac{\mathrm{d}p}{\mathrm{d}h} = -\frac{p \times M \times g}{R \times T}$ où M est la masse molaire moyenne des gaz de l'atmosphère (29 g.mol^{-1}), g l'accélération de la pesanteur, R la constante des gaz parfaits et T la température absolue.

La séparation des variables donne l'équation : $\int_{p_0}^{p_1} \frac{\mathrm{d}p}{p} = -\frac{M \times g}{R} \int_{h_0}^{h_1} \frac{\mathrm{d}h}{T}$.

Si le terme de gauche s'intègre facilement, il faut connaître la dépendance à l'altitude de la température… ce qui pose problème !

a Combien d'étages ai-je monté ?

Pour une faible variation d'altitude et de pression, on peut considérer que $\Delta p \approx -\frac{p \times M \times g}{R \times T} \times \Delta h$. Soit à 298 K, $\Delta p \approx -10\Delta h$.

Montez quelques étages, le capteur proposé devrait être suffisamment précis pour pouvoir déterminer le nombre d'étages montés.

Ce type d'application se retrouve sur votre smartphone (Barometer sur Iphone par exemple) et permet dans les applications type Santé de déterminer le nombre d'étages montés. Notons que si l'on effectue en dénivelé lors d'une marche, le nombre d'étages n'est pas incrémenté. Un algorithme ad-hoc doit vraisemblablement coupler les données de l'accéléromètre (et du gps ?) et celles du baromètre pour savoir si on monte un escalier ou si on monte une côte.

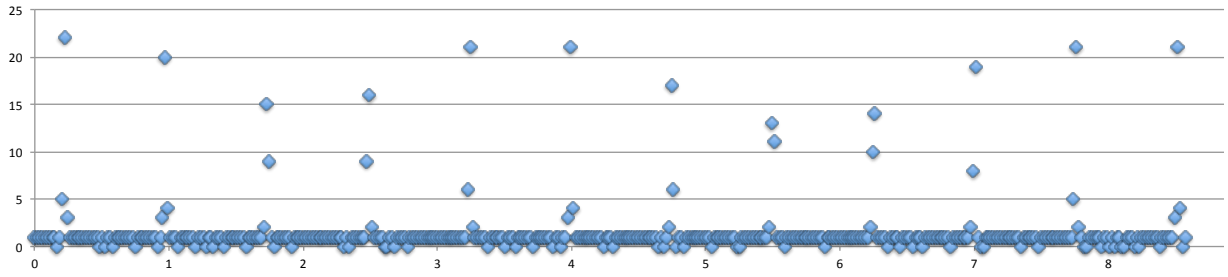
b Formule internationale du nivellement barométrique

Dans ce modèle normalisé, on considère une température de 15 degrés Celcius à l'altitude h_0 correspondant au niveau de la mer et un gradient vertical de température de 0,65 degré pour 100 m. On obtient alors la formule internationale du nivellement barométrique : $p(h) = 1013,25\left(1 - \frac{0,0065 \times h}{288,15}\right)^{5,255}$ (en hPa).

3.6 Compte tour

C'est une expérience « vide grenier »... avec un vieux tourne-disque, un aimant de porte de cuisine et un bout de scotch. A proximité, on fixe le capteur de champ magnétique et on choisit une vitesse de rotation de 18 tours par minutes.

Le tracé de la courbe valeur lue (analogRead() - milieu) en fonction du temps donne la courbe ci-dessous (fichier 78tours.xls joint).

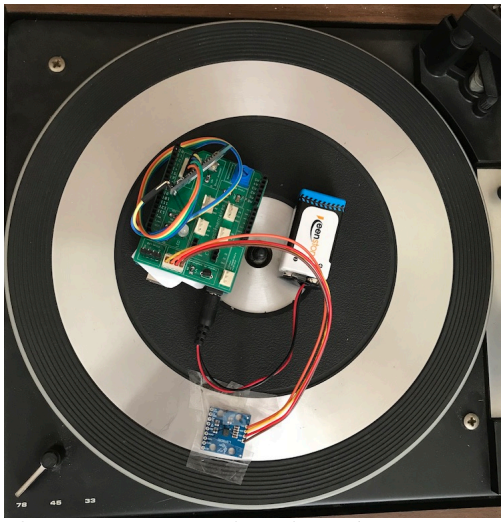


À moyens rudimentaires, approche rudimentaire. On repère les différents maxima « à la main » sur le graphe Excel. En pointant sur le premier maxima on trouve 0,224 s, sur le onzième 7,761 s. Dix tours correspondent donc à 7,537 s. On estime donc une vitesse de rotation de $\frac{60000}{754} \approx 79,5$ tours par minute.

3.7 Un peu de projection

Comme suggéré au paragraphe 5.c.1 page 94, placer l'accéléromètre sur un plan incliné, relever l'accélération sur l'axe vertical et la direction du plan incliné et... proposer quelques projections !

3.8 Étude d'un mouvement circulaire uniforme



La carte est alimentée par une pile afin d'obtenir un système autonome et le capteur est scotché sur un tourne-disque. On transmet alors par Bluetooth les données sur un smartphone. Reste ensuite à récupérer les données et à les traiter.

Le programme `Mouvement` (dossier Annexes) permet de transmettre les données acquises par l'accéléromètre via le module Bluetooth.

Le protocole à suivre est alors le suivant :

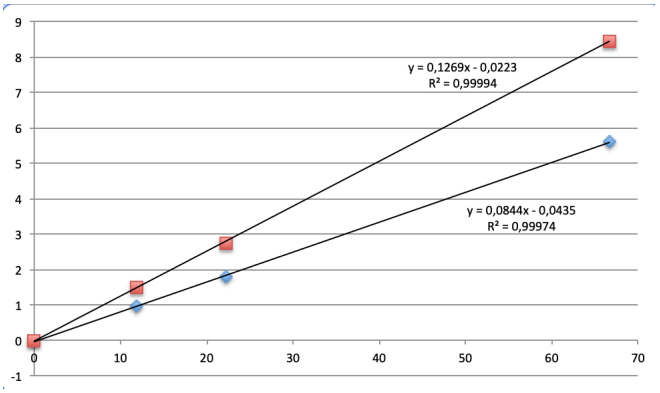
- fixer le capteur à une distance d de l'axe, l'axe z du capteur est vertical et l'axe x le plus possible en direction de l'axe de rotation ;
- mesurer l'accélération selon x au repos (on compense ainsi la non planéité de l'ensemble) ;
- faire tourner à 33, 45 ou 78 tours par minute ;
- les données sont transmises sur le smartphone avec quelques fluctuations !
- faire une moyenne de l'accélération sur x et soustraire l'accélération au repos.

Le tableau suivant regroupe les résultats obtenus lors de nos tests.

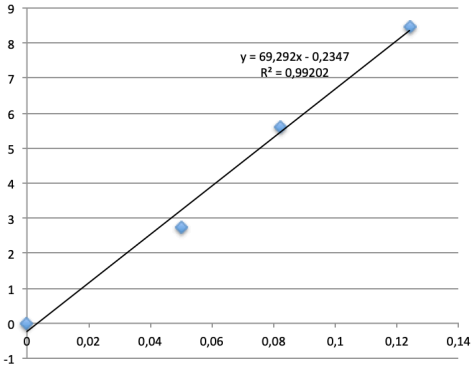
Tours par minute	33	45	78
ω^2 $(\text{rd}.s^{-1})^2$	11,9	22,2	66,7
d ≈ 4,5 cm			2,6
d ≈ 8,2 cm	1,0	1,8	5,6
d ≈ 12,5 cm	1,5	2,8	8,5

En traçant l'accélération radiale en fonction du carré de la vitesse angulaire on obtient, avec un très bon coefficient de corrélation :

- pour une distance d'environ 12,5 cm une droite de pente 0,127 ;
- pour une distance d'environ 8,2 cm une droite de pente 0,084



En traçant l'accélération radiale en fonction de la distance pour une vitesse de rotation de 78 tours, on trouve une droite de pente 69,3 ; ce qui correspond à une vitesse angulaire de 8,3 rd.s^{-1} soit encore 79,5 tours par minute.
Curieux... 79,5 tours par minute était également la vitesse de rotation trouvée à l'aide de notre compte-tour (cf 3.6 page 111).



3.9 Une mini station météo

On se reportera au paragraphe c page 132 pour la présentation de la mise en oeuvre et des mesures obtenus lors du suivi, sur une journée, de température - humidité - pression - luminosité.

4 Solution des exercices

4.1 Chapitre 2

a Feux tricolores

Il s'agit essentiellement de permuter dans la boucle `loop` les différentes couleurs.

- On commence par identifier les différents ports (on aurait pu tout aussi bien écrire const int redLedPin = 9 ; par exemple).
- Dans le `setup` on précise que ce sont des ports utilisés en sortie. Par défaut ils sont à un niveau bas donc la LED est éteinte.
- On utilise les sorties en tout ou rien, donc des sorties digitales, l'instruction à utiliser est `digitalWrite`. Bien sûr on pense spontanément à mettre successivement à un niveau haut la broche correspondant à la couleur souhaitée mais, il faut aussi penser à mettre à un niveau bas l'ancienne ! Lors du premier passage dans la boucle, *blueLEDPin* est à un niveau bas mais lors du second, elle se retrouve à un niveau haut suite à l'exécution de la ligne 21. Si la LED ne s'allume pas en rouge mais en violet, l'erreur est visible !

Une solution possible est la suivante :

Listing 6.4 – Programme FeuxTricolore0

```
1  #define redLEDPin 6
2  #define greenLEDPin 9
3  #define blueLEDPin 10
4
5  void setup() {
6    // put your setup code here, to run once:
7    pinMode(redLEDPin, OUTPUT) ;
8    pinMode(blueLEDPin, OUTPUT) ;
9    pinMode(greenLEDPin, OUTPUT) ;
10 }
11
12 void loop()
13 {
14   digitalWrite(blueLEDPin, LOW) ;
15   digitalWrite(redLEDPin, HIGH) ;
```

```
16    delay(1000) ;
17    digitalWrite(redLEDPin, LOW) ;
18    digitalWrite(greenLEDPin, HIGH) ;
19    delay(500) ;
20    digitalWrite(greenLEDPin, LOW) ;
21    digitalWrite(blueLEDPin, HIGH) ;
22    delay(500) ;
23  }
```

Si on souhaite obtenir une couleur orangée, il faut cette fois utiliser une sortie analogique et utiliser l'instruction `analogWrite`. La modification à apporter au programme précédent est la suivante :

Listing 6.5 – Programme FeuxTricolore1 (extrait)

```
1  void loop()
2  {
3    digitalWrite(greenLEDPin, LOW) ;
4    digitalWrite(redLEDPin, HIGH) ;
5    delay(1000) ;
6    digitalWrite(redLEDPin, LOW) ;
7    digitalWrite(greenLEDPin, HIGH) ;
8    delay(500) ;
9    analogWrite(redLEDPin, 255) ;
10   analogWrite(greenLEDPin, 165) ;
11   delay(500) ;
12 }
```

b SOS

Dans un premier temps, l'approche est simple. Au lieu de changer l'état de la LED, il suffit d'envoyer le message lumineux. Pour éviter d'avoir à recopier de nombreuses fois la même instruction, on crée une fonction `signal(dure)` qui va gérer l'allumage de la lampe pendant *dure* puis son extinction pendant l'intervalle de temps choisi enter 2 signaux. Le code correspondant est donné ci-dessous.

Listing 6.6 – Programme SOS

```
1  #define redLEDPin   6
2  #define buttonPin   2
3  #define lon  600
4  #define court 200
5  #define delai 200
6  boolean lastButton = LOW ;
7
8  void setup()
9  {
10   pinMode(redLEDPin, OUTPUT);
11   pinMode(buttonPin, INPUT) ;
12 }
13
14 void loop()
15 {
16   boolean button = digitalRead(buttonPin) ;
17   if (button != lastButton)
18   {
```

```
19      delay(20) ;
20      button = digitalRead(buttonPin) ;
21      if (button == HIGH)
22      {
23        sos() ;
24      }
25    }
26    lastButton = button ;
27  }
28
29  void sos()
30  {
31    signal(lon) ;
32    signal(lon) ;
33    signal(lon) ;
34    signal(court) ;
35    signal(court) ;
36    signal(court) ;
37    signal(lon) ;
38    signal(lon) ;
39    signal(lon) ;
40  }
41
42  void signal(int dure)
43  {
44    digitalWrite(redLEDPin, HIGH) ;
45    delay(dure) ;
46    digitalWrite(redLEDPin, LOW) ;
47    delay(delai) ;
48  }
```

Pour mettre au point le programme et éviter de changer les valeurs des intervalles de temps dans toutes les lignes 31 à 39, le plus efficace est de définir ces valeurs au début du programme. *long* étant une variable réservée du langage C++, on ne peut pas l'utiliser comme nom de variable ! Par tâtonnement, on arrive ici à un signal long de 600 ms, un court de 200 ms et un intervalle de 200 ms entre deux signaux. D'autres valeurs peuvent, bien sûr, convenir.

Listing 6.7 – Programme SOSContinu (extrait)

```
1   boolean help = false ;
2   #define entre 5000
3   long last = 0 ;
4
5   void loop()
6   {
7     boolean button = digitalRead(buttonPin) ;
8     if (button != lastButton)
9     {
10      delay(20) ;
11      button = digitalRead(buttonPin) ;
12      if (button == HIGH)
13      {
14        help = ! help ;
15      }
16    }
```

```
17    if (millis()-last > entre && help)
18    {
19      sos() ;
20      last = millis() ;
21    }
22    lastButton = button ;
23  }
```

On n'a reporté, ici, que les modifications apportées au programme précédent.

- On introduit la variable *help* de type booléen, mise à jour ligne 14. Cette variable permet de savoir s'il faut émettre ou non un signal.
- Comme la routine `sos` fait appel à `signal` qui utilise un `delay`, ce n'est pas possible d'utiliser un timer (cf page 21). On utilise la stratégie développée page 20 en comptant le temps passé depuis la fin du signal.
- Lorsque l'intervalle de temps dépassent 5000 ms et lorsque *help* est vrai, on émet le signal (ligne 19) et on met à jour la variable *last* afin de compter, à nouveau 5000 ms.

Info sur le programme

On fait tout d'abord un copier/coller du programme LED_EditeurGOSTOP que l'on va modifier.

Il faut définir une chaîne de caractères. Ici via la ligne 2, on aurait pu aussi écrire
const string info = "Programme LED_EditeurInfo : interprète G, S et I".
Reste ensuite à interpréter l'ordre I (ligne 25).

Listing 6.8 – Programme LED_EditeurInfo

```
1  #define redLEDPin 6
2  #define info "Programme LED_EditeurInfo : interprète G, S et I"
3
4  void setup()
5  {
6    Serial.begin(9600) ;
7    pinMode(redLEDPin, OUTPUT);
8  }
9
10 void loop()
11 {
12   if (Serial.available())
13   {
14     String message = Serial.readString() ;
15     message.replace("\r", "") ;
16     message.replace("\n", "") ;
17     if (message == "G")
18     {
19       digitalWrite(redLEDPin, HIGH);
20     }
21     else if (message == "S")
22     {
23       digitalWrite(redLEDPin, LOW);
24     }
25     else if (message == "I")
26     {
27       Serial.println(info) ;
28     }
```

```
29    }
30  }
```

4.2 Chapitre 3

a Éclairage variable

Cet exercice est très simple. On récupère une valeur analogique sur le port A0 (comprise entre 0 et 1023), grâce à la fonction `map` on la convertit en un entier compris entre 0 et 255 afin d'ajuster l'éclairement de la LED (fonction `analogWrite`).

Listing 6.9 – Programme LEDVariable

```
1  #define analogPin A0
2  #define redLEDPin 6
3
4  void setup()
5  {
6    pinMode(redLEDPin, OUTPUT) ;
7  }
8
9  void loop()
10 {
11   int sensorValue = analogRead(analogPin);
12   int redValue = map(sensorValue, 0, 1023, 0, 255) ;
13   analogWrite(redLEDPin, redValue) ;
14 }
```

b Millivoltmètre embarqué

Un objet embarqué doit fonctionner de façon autonome. Les ordres ne peuvent plus provenir de l'ordinateur.

Il faut, ici, combiner les programme ADC_LCD (pour la gestion de la lecture et l'affichage de la tension) et une partie du programme LED_BoutonMarcheArret du chapitre 2 (pour la gestion du bouton poussoir).

On obtient ainsi le programme Voltmetre_Embarque qui peut, par la suite, nous servir de modèle pour la conception d'appareils de mesure autonomes. Nous l'exploiterons dans le chapitre 5.

Listing 6.10 – Programme Voltmetre_Embarque

```
1  #include <Adafruit_GFX.h>
2  #include <Adafruit_SSD1306.h>
3
4  #define buttonPin    2
5  #define analogPin    A0
6  Adafruit_SSD1306 lcd(4);
7  bool lastButton = LOW ;
8  bool doJob = false ;
9
10 void setup()
```

```
11 {
12   lcd.begin(SSD1306_SWITCHCAPVCC, 0x3C);
13   lcd.setTextSize(2);
14   lcd.setTextColor(WHITE);
15   affiche("Appuyez sur le bouton") ;
16   pinMode(buttonPin, INPUT) ;
17 }
18
19 void loop()
20 {
21   boolean button = digitalRead(buttonPin) ;
22   if (button != lastButton)
23   {
24     delay(20) ;
25     button = digitalRead(buttonPin) ;
26     if (button == HIGH)
27     {
28       doJob = ! doJob ;
29       if (!doJob)
30         affiche("Appuyez sur le bouton") ;
31     }
32   }
33   if (doJob)
34     job() ;
35   lastButton = button ;
36 }
37
38 void job()
39 {
40   int sensorValue = analogRead(analogPin);
41   long ddp = map(sensorValue, 0, 1023, 0, 5000) ;
42   affiche("mV = " + String(ddp)) ;
43   delay(100) ;
44 }
45
46 void affiche(String msg)
47 {
48   lcd.clearDisplay() ;
49   lcd.setCursor(0, 0);
50   lcd.println(msg) ;
51   lcd.display();
52 }
```

On retrouve le code des deux programmes utilisés. Le point le plus délicat est la gestion de la boucle. Il faut s'assurer que la fonction `job` soit appelée régulièrement et donc il ne faut pas placer l'appel dans le test du bouton. Le code suivant ne serait pas fonctionnel. En effet, l'appel sera fait une fois ligne 12 lorsqu'on appuiera sur le bouton mais l'affichage ne sera pas mis à jour par la suite si la valeur change (vous pouvez tester !).

Listing 6.11 – Mauvais appel de job()

```
1 void loop()
2 {
3   boolean button = digitalRead(buttonPin) ;
4   if (button != lastButton)
```

```
5   {
6     delay(20) ;
7     button = digitalRead(buttonPin) ;
8     if (button == HIGH)
9     {
10      doJob = ! doJob ;
11      if (doJob)
12        job()
13      else
14        affiche("Appuyez sur le bouton") ;
15    }
16  }
17  lastButton = button ;
18 }
```

Dans un système réel, pour économiser l'énergie, il faudrait prévoir un interrupteur entre la pile et l'Arduino. Ici, on se contentera de débrancher la pile !

c Millivoltmètre connecté

Dans un objet connecté, les échanges d'informations doivent pouvoir se faire avec un smartphone. Le programme LED_ModeleBT nous sert de base de départ puisque l'interface entre l'Arduino et un smartphone est déjà implémenté. La connexion à l'ordinateur est toutefois nécessaire pour la phase programmation (et éventuellement débogage).

Le code proposé du programme Voltmetre_Connecte est, espérons-le, relativement simple à comprendre maintenant.

Listing 6.12 – Programme Voltmetre_Connecte

```
1  #include <MsTimer2.h>
2  #include <SoftwareSerial.h>
3
4  // variables utilisées pour tous les programmes :
5  #define BAUD 9600  // vitesse d'échange pour le port série
6  #define Info "Voltmetre_Connecte"
7  long delai = 1000 ; // Delai en ms entre 2 appels de la
       fonction job
8  #define softTX 4
9  #define softRX 5
10 SoftwareSerial BT(softRX, softTX) ;
11
12 // variables spécifiques au système étudié
13 #define analogPin   A0
14
15 void setup()
16 {
17   Serial.begin(BAUD) ;
18   BT.begin(BAUD) ;
19   MsTimer2::set(delai, job) ;
20 }
21
22 void loop()
23 {
```

```
24    if (Serial.available())
25    {
26      String message = Serial.readString() ;
27      parse(message) ;
28    }
29    if (BT.available())
30    {
31      String message = BT.readString() ;
32      parse(message) ;
33    }
34  }
35
36  void parse(String msg)
37  {
38    msg.replace("\r", "") ;
39    msg.replace("\n", "") ;
40    msg.replace(":", "") ;
41    char ordre = msg[0];
42    msg.replace(String(ordre), "") ;
43    int valeur = 0 ;
44    if (msg.length() > 0)
45      valeur = msg.toInt() ;
46    switch (ordre)
47    {
48      case 'I' :
49        affiche(Info) ;
50        break ;
51      case 'G' :
52        go() ;
53        break ;
54      case 'O' :
55        job() ;
56        break ;
57      case 'S' :
58        stop() ;
59        break ;
60      case 'D' :
61        if (valeur > 0)
62        {
63          delai = valeur ;
64          MsTimer2::set(delai, job) ;
65        }
66        break ;
67      default :
68        parse(ordre, valeur) ;
69        break ;
70    }
71  }
72
73  void affiche(String msg)
74  {
75    Serial.println(msg) ;
76    BT.println(msg) ;
77  }
```

```
78 // traitements spécifiques au système
79
80 void go()
81 {
82     MsTimer2::start() ;  // c'est parti !
83 }
84
85 void stop()
86 {
87   MsTimer2::stop() ; // on stoppe le timer
88 }
89
90 void parse(char ordre, long valeur)
91 {
92   switch (ordre)
93   {
94     default : // rien à faire ici, on le laisse pour le modèle
95       break ;
96   }
97 }
98
99 void job()
100 {
101   int sensorValue = analogRead(analogPin);
102   long ddp = map(sensorValue, 0, 1023, 0, 5000) ;
103   affiche("mV = " + String(ddp)) ;
104 }
```

Ce programme peut nous servir de modèle pour un système d'acquisition connecté. Nous y ferons référence dans le chapitre 5 pour la réalisation (et l'utilisation) d'un accéléromètre BlueTooth.

La limite, ici est que l'on ne peut pas accéder à des lectures en continu puisque le delai du timer ne peut être nul. Le programme VoltmètreContinu_Connecte gère ce cas de figure un petit peu plus complexe (comme le faisait, d'ailleurs le modèle LED_ModeleBT).

d Temps demi décharge

Les seules modifications à apporter au programme ADC_Modele sont reportées ci-dessous.

Listing 6.13 – Programme RC_Tdemi (extrait)

```
1  // variables spécifiques au système étudié
2  #define analogPin    A1
3  #define commandePin 3
4
5  void setup()
6  {
7  ...
8    pinMode(commandePin, OUTPUT) ;
9  ...
10 }
11
12 void job()
13 {
```

```
14    digitalWrite(commandePin, HIGH) ;
15    delay(3000) ;
16    int sensorValue0 = analogRead(analogPin);
17    int sensorValue = sensorValue0 ;
18    digitalWrite(commandePin, LOW) ;
19    start = millis() ;
20    while (sensorValue > sensorValue0/2)
21    {
22      sensorValue = analogRead(analogPin);
23    }
24    long time = millis() ;
25    String msg = "DATA" ;
26    msg = msg + ":TIME:" + String(time-start) ;
27    affiche(msg) ;
28  }
```

La valeur de la sortie analogique lue sur le port **3** est proportionnelle à la tension aux bornes du condensateur. Après avoir chargé le condensateur (lignes 14 et 15), on lit la valeur initiale sur le port **A1**. On annule la tension aux bornes du circuit RC (ligne 18), il ne nous reste plus qu'à mesurer le temps au bout duquel la valeur lue sur la port **A1** est diminuée de moitié (boucle `while` lignes 20 à 23). Reste à afficher le résultat sur le port série.

e Étude en fonction de la tension de charge

Au lieu d'utiliser la broche **3** comme sortie digitale (via un `digitalWrite`), il suffit de l'utiliser comme (pseudo) sortie analogique. L'instruction `analogWrite(valeur)` permet d'obtenir une tension de charge de $\frac{valeur}{255} \times 5000mV$; *valeur* étant un entier compris entre 0 et 255. Il suffit alors de modifier la ligne 14 du programme précédent et d'englober le tout dans une boucle. Par souci de clarté, on peut aussi introduire une nouvelle fonction.

Listing 6.14 – Programme RC_TdemiAuto (extrait)

```
1   void job()
2   {
3     int pwm = 255 ;
4     while (pwm >= 50)
5     {
6       RC(pwm) ;
7       pwm = pwm - 25 ;
8     }
9   }
10
11  void RC(int valeur)
12  {
13    long T = 0 ;
14    int sensorValue0 ;
15    for (int i = 0 ; i < 10 ; i ++ )
16    {
17      analogWrite(commandePin, valeur) ;
18      delay(1000) ;
19      sensorValue0 = analogRead(analogPin);
20      int sensorValue = sensorValue0 ;
21      digitalWrite(commandePin, LOW) ;
22      start = millis() ;
```

```
23      while (sensorValue > sensorValue0 / 2)
24      {
25        sensorValue = analogRead(analogPin);
26      }
27      T = T + (millis() - start) ;
28    }
29    String msg = "DATA" ;
30    msg = msg + ":A1_0:" + String(sensorValue0) ;
31    msg = msg + ":TIME:" + String(T / 10) ;
32    affiche(msg) ;
33  }
```

Quelques remarques :

- Nous avons utilisé une boucle `while` dans la fonction `job`, peut-être plus facile à coder qu'une boucle `for` ici.
- Dans la fonction `RC`, pour une chaque valeur de la tension de charge étudiée, on fait la moyenne de 10 expériences de détermination du temps de demi décharge. La variable T est incrémentée à chaque expérience puis divisée par 10 lors de l'affichage.

Nous obtenons les résultats :

A1_0	1023	922	826	721	625	518	423	323	221
Temps (ms)	51	53	55	56	57	58	58	59	57

Les plus optimistes pourront conclure que le temps mis pour une demi-décharge est indépendant de la tension de charge !

f Générateur tension

L'idée, pour maintenir une tension à peu près constante est de contrôler, en continu, l'entrée analogique. Quand la valeur devient trop faible, on charge le condensateur et quand celle-ci devient trop grande, on décharge.

Les modifications apportées au programme GenerateurTension par rapport à ADC_ModeleNoTimer sont résumées ci-dessous.

Listing 6.15 – Programme GenerateurTension (extrait)

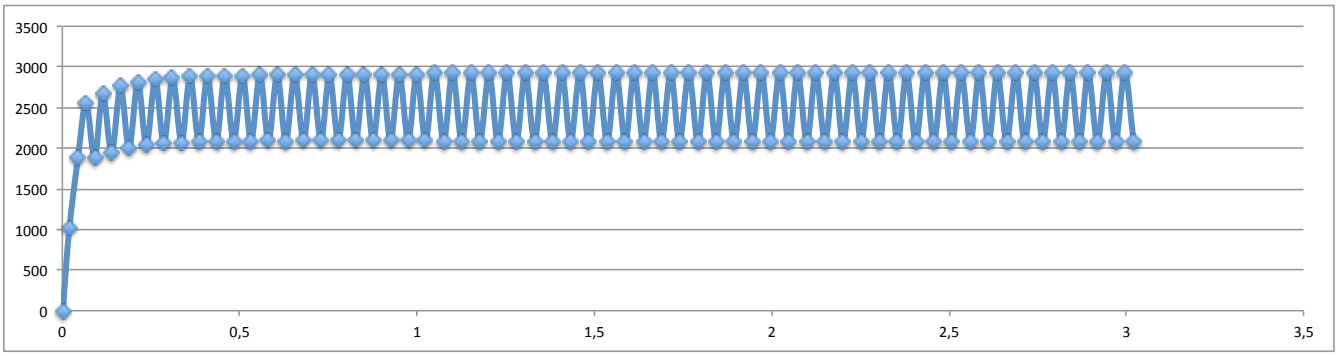
```
1
2   // variables spécifiques au système étudié
3   #define analogPin    A1
4   #define commandePin 3
5   int A1Ref = 1023 ;
6
7   void parse(char ordre, long valeur)
8   {
9     switch (ordre)
10    {
11      case 'T' :
12        A1Ref = map(valeur, 0, 5000, 0, 1023) ;
13        break ;
14      default :
15        break ;
16    }
```

```
17 }
18
19 void job()
20 {
21   int sensorValue = analogRead(analogPin);
22   if (sensorValue < 0.99 * A1Ref)
23     digitalWrite(commandePin, HIGH) ;
24   else if (sensorValue > 1.01 * A1Ref)
25     digitalWrite(commandePin, LOW) ;
26   String msg = "DATA" ;
27   msg = msg + ":TIME:" + String(millis() - start) ;
28   msg = msg + ":A1:" + String(sensorValue) ;
29   affiche(msg) ;
30 }
```

Les modifications apportées sont les suivantes.

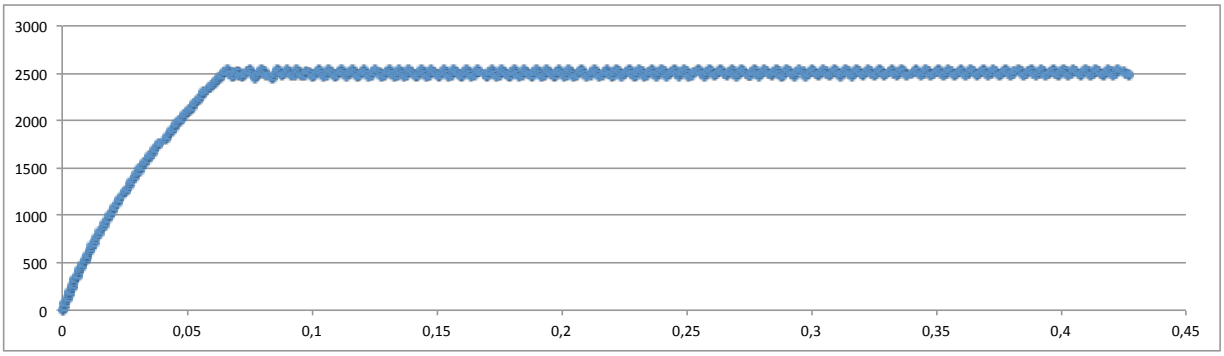
- Ligne 5 on introduit une variable $A1Ref$ stockant la valeur souhaitée de l'entrée analogique sur le port **A1**.
- Lors de la réception de l'ordre 'T :valeur', l'instruction ligne 12 permet de déterminer la valeur de $A1ref$ en fonction de la tension souhaitée *valeur*.
- Dans la fonction `job` (appelée en continu), on fait une lecture analogique de la tension sur le port **A1**.
  - si cette valeur est plus petite que $A1Ref$ moins 1%, on charge le condensateur ;
  - si cette valeur est plus grande que $A1Ref$ plus 1%, on décharge le condensateur ;
  - si la valeur est entre les deux, on conserve l'état précédent.

Dans l'exemple suivant, on souhaitait une tension de sortie stabilisée à 2500 mV on obtient :



Effectivement, on a bien une tension en moyenne voisine de 2500 mV mais les fluctuations sont quand même importantes !// L'analyse de ce résultat est intéressante. On s'imposait comme limite 2500 mV plus ou moins 25 mV ; et on a plutôt des oscillations de 100 mV. Que constate-t-on sur l'intervalle de temps entre 2 valeurs : on a fait une mesure environ toutes les 22 ms, or... notre système a une constante de temps de l'ordre de 50 ms ! Que faire ?

- On peut jouer sur la constante de temps $\tau = RC$. Les composants sont soudés mais, vous disposez d'un pont diviseur de tension ou d'une platine de prototypage et d'un jeux de composants pour faire ce montage avec d'autres valeurs de R et C.
- On peut aussi tenter de diminuer la période d'échantillonnage. En optimisant un peu le code et en augmentant la vitesse de transfert sur le port série on peut obtenir le résultat ci-dessous. La période d'échantillonnage est, ici, ramenée à 1 à 2 ms.



g En guise de révision

La modification de la fonction `job` du programme ADC_ModeleNoTimer est reportée ci-dessous, nous avons également opté pour une vitesse de communication de 115200 bauds.

Listing 6.16 – Programme RC_acq (extrait)

```
1  void job()
2  {
3    digitalWrite(commandePin, HIGH) ;
4    delay(2000) ;
5    int sensorValue0 = analogRead(analogPin);
6    int sensorValue = sensorValue0 ;
7    digitalWrite(commandePin, LOW) ;
8    start = millis() ;
9    while (sensorValue > sensorValue0 / 100)
10   {
11     sensorValue = analogRead(analogPin);
12     String msg = "DATA" ;
13     msg = msg + ":TIME:" + String(millis() - start) ;
14     msg = msg + ":A1:" + String(sensorValue) ;
15     Serial.println(msg) ;
16   }
17 }
```

Le programme python est directement calqué sur le programme acquisitionArduinoAcq.py (page 51).

Listing 6.17 – Programme arduinoAcqRC.py

```
1 import ArdTools.ArdTools as ard
2
3 a = ard.ArduinoAcq()
4 ports = a.ports()
5 a.connexion(ports[len(ports)-1], 115200)
6
7 a.sauvegarde("dataRC.txt")
8 a.envoyer("D:0")
9 a.go()
```

Reste ensuite à exploiter les données, par exemple à l'aide du programme acquisitionRC.py (page 57).

Changer la ligne 3 par `a = ard.ArduinoPlot()` permet d'avoir directement la courbe représentative de la lecture analogique en fonction du temps.

h Générateur triangle

Quelques modifications du programme GenerateurTension sont nécessaires.

- La régulation de la tension par rapport à la référence (variable $A1Ref$) doit se faire le plus souvent possible donc dans la boucle principale.
- La modification de la valeur « cible » doit être, quant à elle, mise à jour dans la fonction `job` appelée à intervalle de temps fixé.

Les modifications apportées au programme GenerateurTension sont reportées dans le listing ci-dessous.

Listing 6.18 – Programme GenerateurTriangle (extrait)

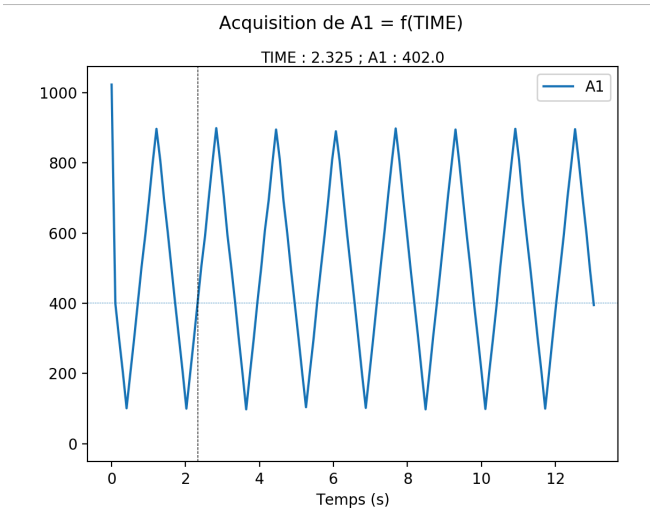
```
#define BAUD 115200  // vitesse d'échange pour le port série
#define Info "Generateur Triangle"

long delai = 100 ;

// variables spécifiques au système étudié
#define analogPin   A1
#define commandePin 3
int A1Ref = 0 ;
int pas = 100 ;
int sensorValue ;
bool up = true ;

void loop()
{
  if (Serial.available())
  {
    String message = Serial.readString() ;
    parse(message) ;
  }
  sensorValue = analogRead(analogPin);
  if (sensorValue < 0.99 * A1Ref)
    digitalWrite(commandePin, HIGH) ;
  else if (sensorValue > 1.01 * A1Ref)
    digitalWrite(commandePin, LOW) ;
  if (doJob && millis() - last> delai)
  {
    job() ;
    last = millis() ;
  }
}

void job()
{
  String msg = "DATA" ;
  msg = msg + ":TIME:" + String(millis() - start) ;
  msg = msg + ":A1:" + String(sensorValue) ;
  affiche(msg) ;

  if (up)
  {
    A1Ref = A1Ref + pas ;
```

```
43      if (A1Ref > 800)
44        up = false ;
45    }
46    else
47    {
48      A1Ref = A1Ref - pas ;
49      if (A1Ref < 200)
50        up = true ;
51    }
52  }
```

Le programme python est très court !

Listing 6.19 – Programme arduinoPlotRC.py

```
1  import ArdTools.ArdTools as ard
2
3  a = ard.ArduinoPlot()
4  ports = a.ports()
5  a.connexion(ports[len(ports)-1], 115200)
6
7  a.go()
```



On peut alors jouer sur la fréquence du signal à l'aide des variables *delai* et *pas*. On pourrait imaginer passer la variable *pas* en paramètre pour avoir un générateur de triangle de fréquence variable, c'est simple à mettre en œuvre maintenant que vous savez tout faire !

Allez, un challenge... concevez un générateur de signal sinusoïdal !

4.3 Chapitre 4

a Capteur de température embarqué

Vous avez compris la logique, on part d'un exemple qui fonctionne : le programme de l'exercice Voltmetre_Embarque (du chapitre 3, correction page 117) et on modifie quelques lignes à l'aide du programme Simple de la bibliothèque Dallas.

Les modifications sont reportées dans le listing ci-dessous. Cette fois encore, il suffit d'inclure les bibliothèques nécessaires, de définir quelques variables et d'instancier quelques objets (lignes 4 et suivantes), d'ajouter une initialisation dans le `setup` et, surtout, de modifier la fonction **job**.

Listing 6.20 – Programme Temperature_Embarque (extrait)

```
1  ...
2  #include <OneWire.h>
3  #include <DallasTemperature.h>
```

```
4
5  #define ONE_WIRE_BUS 7
6  OneWire oneWire(ONE_WIRE_BUS);
7  DallasTemperature sensors(&oneWire);
8
9  void setup()
10 {
11   ...
12   sensors.begin();
13 }
14
15 void job()
16 {
17   sensors.requestTemperatures();
18   float T = sensors.getTempCByIndex(0);
19   affiche("T = " + String(T,1) + " C") ;
20   delay(1000) ;
21 }
```

b Capteur de température connecté

On modifie cette fois le programme de l'exercice Voltmetre_Connecte (du chapitre 3, correction page 119) de la même manière que dans l'exemple précédent.

Listing 6.21 – Programme Temperature_Connecte (extrait)

```
1  #include <OneWire.h>
2  #include <DallasTemperature.h>
3
4  #define ONE_WIRE_BUS 7
5  OneWire oneWire(ONE_WIRE_BUS);
6  DallasTemperature sensors(&oneWire);
7
8  void setup()
9  {
10   ...
11   sensors.begin();
12 }
13 void job()
14 {
15   sensors.requestTemperatures();
16   float T = sensors.getTempCByIndex(0);
17   affiche("T = " + String(T,1)+ " C") ;
18 }
```

c Indicateur à LED

Il suffit juste de tester la valeur de la température ambiante par rapport aux températures de référence choisies. Le code proposé exploite le fait que le deuxième paramètre de la fonction `analogWrite` étant un booléen. LOW étant équivalent à faux et HIGH à vrai, un simple test suffit.

Listing 6.22 – Programme TemperatureLED

```
#include <OneWire.h>
#include <DallasTemperature.h>

#define redLEDPin 6
#define greenLEDPin 9
#define blueLEDPin 10
#define ONE_WIRE_BUS 7
OneWire oneWire(ONE_WIRE_BUS);
DallasTemperature sensors(&oneWire);

float TMin = 22 ;
float TMax = 25 ;

void setup()
{
  pinMode(redLEDPin, OUTPUT) ;
  pinMode(greenLEDPin, OUTPUT) ;
  pinMode(blueLEDPin, OUTPUT) ;
  sensors.begin();
}

void loop()
{
  sensors.requestTemperatures(); // Send the command to get
     temperatures
  float T = sensors.getTempCByIndex(0);
  digitalWrite(redLEDPin, T > TMax) ;
  digitalWrite(blueLEDPin, T < TMin) ;
  digitalWrite(greenLEDPin, T > TMin && T < TMax) ;
  delay(100) ;
}
```

d Rupture de la chaine du froid

Le programme Temperature_Embarque modifié est le suivant.

Listing 6.23 – Programme ChaineDuFroid

```
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#include <OneWire.h>
#include <DallasTemperature.h>

#define ONE_WIRE_BUS 7
Adafruit_SSD1306 lcd(4);
OneWire oneWire(ONE_WIRE_BUS);
DallasTemperature sensors(&oneWire);
long delai = 1000 ;
unsigned long lastTime = 0 ;
long elapsed = 0 ;
bool last = false ;
float T1 = 28 ;
```

```
15
16 void setup()
17 {
18   lcd.begin(SSD1306_SWITCHCAPVCC, 0x3C);
19   lcd.setTextSize(2);
20   lcd.setTextColor(WHITE);
21   sensors.begin();
22 }
23
24 void loop()
25 {
26   if (millis() - lastTime > delai)
27   {
28     lastTime = millis() ;
29     sensors.requestTemperatures();
30     float T = sensors.getTempCByIndex(0);
31     if (last && T > T1)
32       elapsed = elapsed + delai / 1000 ;
33     if (T > T1)
34       last = true ;
35     lcd.clearDisplay() ;
36     lcd.setCursor(0, 0);
37     lcd.println("T = " + String(T, 1)) ;
38     lcd.print("hot = " + String(elapsed) + " s") ;
39     lcd.display();
40   }
41 }
```

Les instructions lignes 29 et 30 étant chronophages, il vaut mieux décompter le temps passé plutôt qu'utiliser la fonction `delay`. L'incrémentation du temps passé au-dessus de la température $T1$ se fait ligne 32 lorsque la température actuelle et la température précédemment mesurée sont supérieures à T1.

e Capteur de température embarqué, 2 températures

Il s'agit de modifier le programme Temperature_Embarque afin de gérer l'acquisition de deux températures. Compte-tenu du code du programme Temperatures (cf 3.2 page 67), seule la fonction `job` est à modifier.

Listing 6.24 – Programme Temperatures_Embarque (extrait)

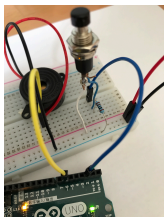
```
1  void job()
2  {
3    sensors.requestTemperatures(); // Send the command to get
       temperatures
4    String msg1 = "T1 : " + String(sensors.getTempCByIndex(0),
       1)  ;
5    String msg2 = "T2 : " + String(sensors.getTempCByIndex(1),
       1)  ;
6    lcd.clearDisplay() ;
7    lcd.setCursor(0, 0);
8    lcd.println(msg1) ;
9    lcd.print(msg2) ;
10   lcd.display();
11   delay(1000) ;
```

```
12 }
```

4.4 Chapitre 5

a SOS

Le montage proposé peut-être le suivant :



- On aurait pu câbler le buzzer sans passer par la platine (directement entre la masse et le port 6).
- On reproduit, sur la platine, le montage du bouton poussoir avec sa résistance de protection comme dans l'entête du chapitre 2.

Listing 6.25 – Programme SOS

```
1  #define buzzPin     6
2  #define contactPin  2
3  #define la          440
4  #define lon         1000
5  #define court       500
6  #define inter       500
7
8  void setup() {
9    pinMode(buzzPin, OUTPUT) ;
10   pinMode(contactPin, INPUT) ;
11 }
12
13 void loop()
14 {
15   if (digitalRead(contactPin) == HIGH)
16   {
17     sos() ;
18   }
19 }
20
21 void sos()
22 {
23   signal(court) ;
24   signal(lon) ;
25   signal(court) ;
26 }
27
28 void signal(int duree)
29 {
30   for (int i = 0 ; i < 3 ; i++)
31   {
32     tone(buzzPin, la, duree) ;
33     delay(duree + inter) ;
34   }
35 }
```

- le fait de définir les durées des impulsions en tête de programme permet d'adapter ces valeurs à ce que l'on souhaite obtenir en ne changeant que les lignes correspondantes ;

- puisque la fonction `tone` ne bloque pas l'exécution, le délai qui suit (ligne 33) doit en tenir compte !

b Horloge

Une simple compilation des fichiers `Alarme` (page 83) et `ADC_LCD` (page 43) permet de créer le programme **Horloge** qui affiche la date et l'heure sur l'écran LCD.

Listing 6.26 – Programme Horloge

```
1  #include <Wire.h>
2  #include <DS3231.h>
3  #include <Adafruit_GFX.h>
4  #include <Adafruit_SSD1306.h>
5
6  DS3231 clock;
7  Adafruit_SSD1306 lcd(4);
8
9  void setup()
10 {
11   clock.begin();
12   clock.setAlarm1(0, 0, 0, 0, DS3231_EVERY_SECOND);
13   lcd.begin(SSD1306_SWITCHCAPVCC, 0x3C);
14   lcd.setTextSize(2);
15   lcd.setTextColor(WHITE);
16   lcd.clearDisplay() ;
17   lcd.display();
18 }
19
20 void loop()
21 {
22   if (clock.isAlarm1())
23   {
24     RTCDateTime dt = clock.getDateTime();
25     Serial.println(clock.dateFormat("d-m-Y H:i:s - l", dt));
26     lcd.clearDisplay();
27     lcd.setCursor(0, 0);
28     lcd.print(clock.dateFormat("d-m-Y", dt)) ;
29     lcd.println(clock.dateFormat("H:i:s", dt)) ;
30     lcd.display() ;
31   }
32 }
```

c Station météo

Il faut câbler :

- le module Bluetooth afin de passer les ordres à la carte ;
- la carte SD sur port SPI (cf chapitre 5.3.2 page 84) ;
- l'horloge RTC sur port I2C (cf chapitre 5.3.1 page 81) ;
- le module de mesure température-humidité-pression BME280 (cf chapitre 5.3.3 page 87) ;
- un diviseur de tension afin de déterminer la différence de potentiel aux bornes de la LDR (cf chapitre 5.a page 72).

Plusieurs solutions s'offrent, bien sûr, à nous afin de construire le programme **Meteo**. Il s'agit, ici d'un bel exercice de d'aggrégation de morceaux de codes existant.

Partir du programme DataLogging_Modele qui gérait déjà la carte SD et le module Bluetooth semble raisonnable ! Les modifications à apporter au programme sont reprises dans le listing suivant.

Listing 6.27 – Programme Meteo(extrait)

```
#include <SoftwareSerial.h>
#include <SPI.h>
#include <SD.h>
#include <Wire.h>
#include <DS3231.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>

// variables spécifiques au système étudié
#define ldrPin  A0
Adafruit_BME280 bme;
#define chipSelect  10
#define logFile     "data.txt"
File dataFile ;
DS3231 clock;
int count = 0 ;
#define maxCount  300

void setup()
{
  Serial.begin(BAUD) ;
  BT.begin(BAUD) ;
  SD.begin(chipSelect) ;
  bme.begin() ;
  clock.begin();
  clock.setAlarm1(0, 0, 0, 0, DS3231_EVERY_SECOND);
}

void loop()
{
...
  if (doJob && clock.isAlarm1())
  {
    count = count + 1 ;
    if (count >= maxCount)
    {
      job() ;
      count = 0 ;
    }
  }
}

void affiche(String msg)
{
  Serial.println(msg) ;
  BT.println(msg) ;
  dataFile = SD.open(logFile, FILE_WRITE) ;
```

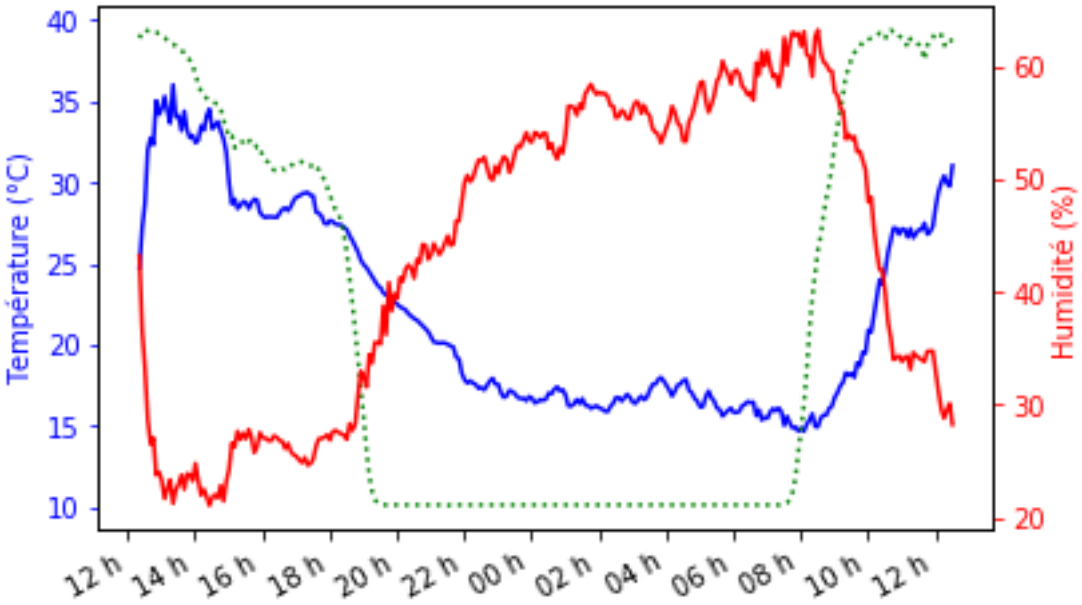
```
48    if (dataFile)
49      dataFile.println(msg) ;
50    dataFile.close() ;
51  }
52
53  void job()
54  {
55    RTCDateTime dt = clock.getDateTime();
56    String msg = "DATA:HMS:" + String(dt.hour) + ":" +
         String(dt.minute) + ":" + String(dt.second) ;
57    msg = msg + ":T:" + String(bme.readTemperature()) + ":P:" +
         String(bme.readPressure()) + ":H:" +
         String(bme.readHumidity()) ;
58    msg = msg + ":L:" + String(analogRead(ldrPin)) ;
59    affiche(msg) ;
60  }
```

- Lignes 1 à 7 : on importe des différentes bibliothèques.
- Lignes 10 à 17 : on initialise les différentes variables et on instancie les objets nécessaires.
- On initialise ces différents objets dans le `setup` lignes 21 à 26. On copie la ligne 12 du code gérant les alarmes de l'horloge du programme `Alarme` (cf chapitre 5.b.1 page 83). Utiliser une alarme toutes les minutes serait préférable. On peut le faire en utilisant l'alarme 2... et la documentation !
- Il faut modifier la boucle `loop`. On récupère l'alarme toutes les secondes (ou toutes les minutes si on a implémenté cette solution). Un compteur permet de n'appeler la fonction `job` une fois sur 300 ; donc toutes les 5 minutes.
- On a modifié légèrement la fonction `affiche`. Comme l'acquisition a lieu toutes les cinq minutes, on a largement le temps d'écrire cette donnée sur carte SD à chaque fois. Ainsi, lorsqu'on débranchera l'alimentation, les données seront sauvegardées !
- La fonction `job`... fait le job ! On récupère :
  - l'heure sur l'horloge (ligne 55) ;
  - la température, pression, humidité sur la carte BME (ligne 57) ;
  - la différence de potentiel aux bornes de la LDR ligne 58.

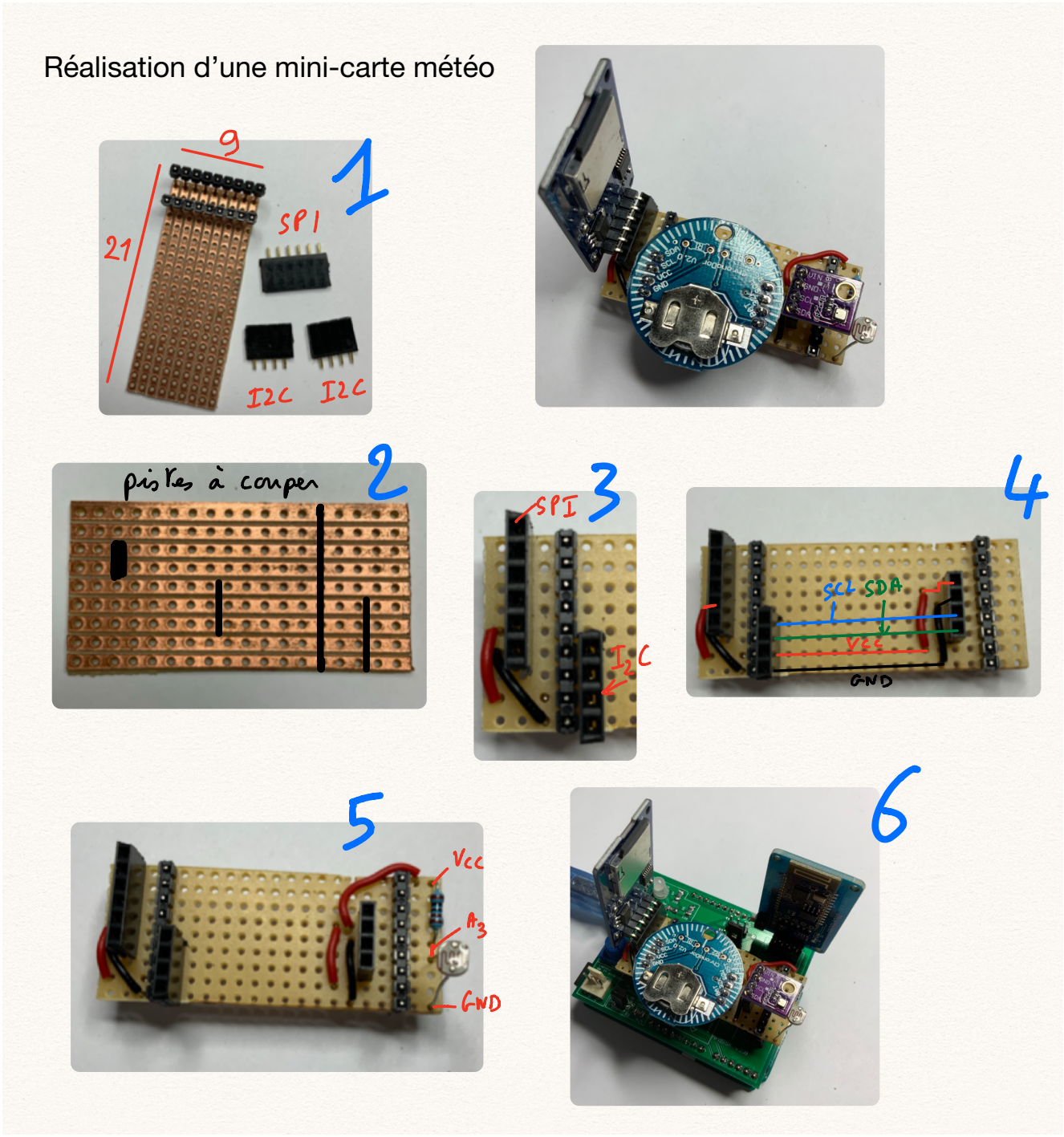
  Reste alors à formater les données. Le formatage utilisé nécessitera un traitement particulier si on souhaite récupérer les données en python.

Le fichier `meteo15octobre.txt` correspond aux données acquises dans la mini station météo ci-contre. Bon... il faisait chaud ce jour là, et notre boite n'était pas vraiment sous abri !

Le code python permettant de tracer des courbes avec des dates en abscisse ne s'invente pas ! Une solution possible est proposé dans le programme `meteo.py`. Aux courbes donnant l'évolution de la température et de l'humidité en fonction du temps à été ajouté une fonction (arbitraire) de l'éclairement (en trait pointillé).



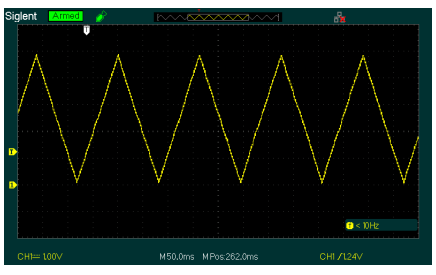
Le document suivant présente quelques étapes de la réalisation d'une mini-carte permettant de stabiliser notre système.



d Signal triangulaire

d.1 A l'oscillo

La modification du programme `CNA` (chapitre 5.3.5 page 90) permet de générer un signal triangulaire. On a créé dans la boucle `loop` du programme TriangleOscillo une boucle infinie incrémentant ou décrémentant un compteur.



Listing 6.28 – Programme TriangleOscillo

```
1 #include <Wire.h>
2 #include <Adafruit_MCP4725.h>
```

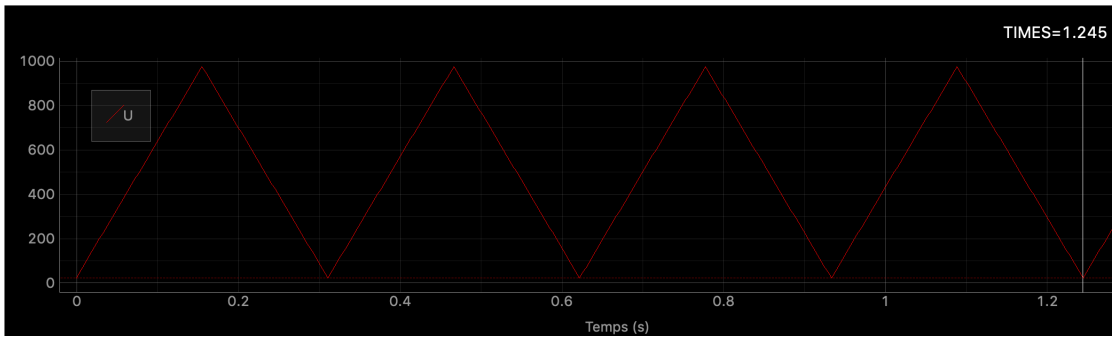
```
3
4  Adafruit_MCP4725 dac;
5
6  void setup(void)
7  {
8    dac.begin(0x62);
9  }
10
11 void loop(void) {
12   long counter = 0 ;
13   int incrementRef = 10 ;
14   int increment = incrementRef ;
15   while (true)
16   {
17     dac.setVoltage(counter, false);
18     if (counter >= 4000 - increment)
19       increment = -incrementRef ;
20     else if (counter <= increment)
21       increment = incrementRef ;
22     counter = counter + increment ;
23   }
24 }
```

d.2 Première étape vers le générateur de signal On modifie donc le programme `ADC_ModeleNoTime` En tenant compte du programme `CNA` on effectue les initialisations nécessaires. Comme dans le programme ci-dessus, on va incrémenter un compteur mais, cette fois, dans la fonction `job`. Les principales instructions, dans le programme `Triangle` ainsi créé sont reportées dans le listing ci-dessous.

Listing 6.29 – Programme Triangle(extrait)

```
1  ...
2  long counter = 0 ;
3  int incrementRef = 100 ;
4  int increment = incrementRef ;
5  ...
6  void job()
7  {
8    dac.setVoltage(counter, false);
9    int adc = analogRead(A3) ;
10   Serial.println("DATA:TIMES:" + String(millis()) + ":U:" +
       String(adc)) ;
11   if (counter >= 4000 - incrementRef)
12     increment = -incrementRef ;
13   else if (counter <= incrementRef)
14     increment = incrementRef ;
15   counter = counter + increment ;
16 }
```

Lors de l'exécution avec le programme TraceQt, choisir un délai d'échantillonage de 10 ms.



La période du signal ainsi généré est d'environ 0,3 secondes. Le compteur allait de 0 à 4000 par pas de 100. 80 points ont donc été générés. On peut ainsi, avec notre système espérer atteindre une période d'échantillonage de 4 ms. Le « maillon faible » de notre système est, à coup sûr, le convertisseur analogique-numérique de l'Arduino (ce serait encore pire avec l'ADS1115 avec 16 bits de résolution) mais notre protocole de transmission de données n'est pas non plus optimal.

e Générateur de signal

L'incrémentation d'un compteur permettait « aisément » de générer un signal triangulaire dans l'exercice précédent. Ici, notre système est plus complexe : on doit à la fois pouvoir gérer plusieurs signaux et la fréquence du signal. Il semble donc judicieux de s'affranchir de ce compteur et de n'utiliser comme seule variable temporelle que... le temps !

- la programmation de la fonction sinus sera alors très simple ;
- pour la fonction signal carré : il suffit, par exemple, de calculer le sinus à la fréquence souhaitée, si le sinus est positif alors la sortie prend sa valeur maximale, sinon elle prend sa valeur minimale.
- la génération du signal triangulaire nécessite quelques calculs... on peut tester la décomposition en série de Fourier décrite dans l'articte « signal triangulaire » sur Wikipedia, ou la formule qui s'y trouve et reportée dans l'énoncé de cet exercice.

Dans le programme `GenerateurSignal`, les principales modifications apportées au programme ADC_ModeleNoTimer sont reportées dans le listing suivant.

Listing 6.30 – Programme GenerateurSignal(extrait)

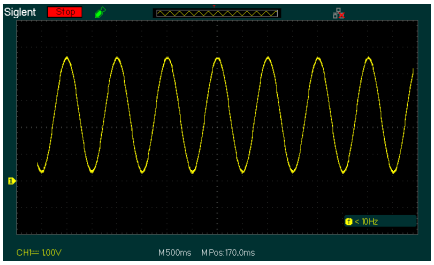
```
1  ...
2  int programme = 0 ; // 0 : triangle - 1 : sinus - 2 : carré
3  int frequence = 10 ;
4  bool acquisition = true ;
5  ...
6  void parse(char ordre, long valeur)
7  {
8    switch (ordre)
9    {
10     case 'P' : programme = valeur ; break ;
11     case 'F' : frequence = valeur ; break ;
12     case 'A' : acquisition = valeur > 0 ; break ;
13   }
14 }
15
16 void job()
17 {
```

```
18    float impose = 0 ;
19    float temp ;
20    unsigned long tms = millis() - start ;
21    float tf = tms * frequence / 1000.0 ;
22    switch (programme)
23    {
24      case 0 : // triangle
25        temp = 4000 * (tf - int(tf + 0.5)) ;
26        if (temp > 0)
27          impose = temp ;
28        else
29          impose = - impose ;
30        break ;
31      case 1 : // sinus
32        impose =  2000 + 2000 * sin(tf * 6.28) ;
33        break ;
34      case 2 : // carré
35        temp = sin(tf * 6.28) ;
36        if (temp > 0)
37          impose = 4000 ;
38        else
39          impose = 0 ;
40        break ;
41    }
42    dac.setVoltage(impose, false);
43    if (acquisition)
44    {
45      int adc = analogRead(analogPin) ;
46      Serial.println("DATA:TIMES:" + String(tf) + ":U:" +
          String(adc)) ;
47    }
48  }
```

Trois nouvelles variables *programme*, *frequence*, *acquisition* sont nécessaires pour gérer l'état du système.

- Elles sont créées lignes 2, 3 et 4 avec leurs valeurs par défaut.
- Elles sont mises à jour dans la fonction `parse` (lignes 10,11,12).
- Elles sont exploitées dans la fonction `job` afin de spécifier, ligne 42, la valeur à imposer au convertisseur.

On obtient, par exemple, pour une fréquence de 1 Hz le signal ci-contre à l'oscillo.



f Accéléromètre zéro au repos !

La conception d'un programme `AccelerometreZero` ne devrait plus vous poser de problèmes maintenant. Une solution possible est la suivante

Listing 6.31 – AccelerometreZero

```
#include <Wire.h>
#include <Adafruit_LSM9DS1.h>
#include <Adafruit_Sensor.h>  // not used in this demo but required!

Adafruit_LSM9DS1 lsm = Adafruit_LSM9DS1();//I2C
float amx, amy, amz ;
float ax, ay, az ;

void setup()
{
  Serial.begin(9600);
  lsm.begin() ;
  lsm.setupAccel(lsm.LSM9DS1_ACCELRANGE_2G);
  calculMoyenne(50) ;
  Serial.println("Initialisation : ") ;
  affiche() ;
  Serial.println("Mesure : ") ;
  amx = ax ;
  amy = ay ;
  amz = az ;
}
void loop()
{
  calculMoyenne(5) ;
  affiche() ;
}

void affiche()
{
  float atot = sqrt((ax-amx) * (ax-amx) + (ay-amy) * (ay-amy) + (az-amz) * (az-amz)) ;
  Serial.print("A : " + String(atot)) ;
  Serial.print("\t AX: " + String(ax-amx));
  Serial.print("\t AY: " + String(ay-amy));
  Serial.print("\t AZ: " + String(az-amz));
  Serial.println();
}

void calculMoyenne(int n)
{
  ax = 0 ;
  ay = 0 ;
  az = 0 ;
  sensors_event_t a, m, g, temp;
  for (int i = 0 ; i < n ; i++)
  {
    lsm.read();
    lsm.getEvent(&a, &m, &g, &temp);
    ax = ax + a.acceleration.x ;
    ay = ay + a.acceleration.y ;
    az = az + a.acceleration.z ;
    delay(100) ;
```

```
52    }
53    ax = ax / n ;
54    ay = ay / n ;
55    az = az / n ;
56  }
```

Dans ces conditions, on récupère en sortie l'accélération réellement ressentie par le capteur. Au repos on obtient, par exemple, les résultats ci-contre.
Notez, ensuite, la grande sensibilité du système.

```
Initialisation :
A : 9.57        AX: 0.62        AY: -0.42       AZ: 9.54
Mesure :
A : 0.07        AX: -0.01       AY: -0.00       AZ: -0.07
A : 0.06        AX: 0.01        AY: 0.00        AZ: -0.06
A : 0.06        AX: 0.01        AY: -0.01       AZ: -0.06
A : 0.06        AX: -0.01       AY: -0.01       AZ: -0.06
```

g Un dé magique

Le plus difficile est de réaliser la boite en calant bien le capteur avec les directions x, y et z parallèles aux côtés de la boite et, une fois celle-ci réalisée... de maîtriser l'Art du détournement d'attention avant d'entamer une carrière de magicien.

Listing 6.32 – De_Magique(extrait)

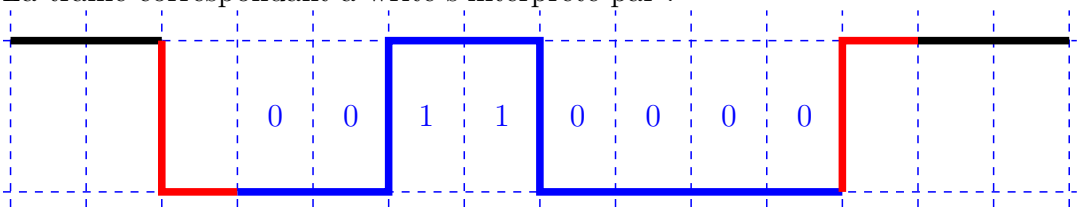
```
1
2   void loop()
3   {
4     lsm.read();  /* ask it to read in the data */
5     sensors_event_t a, m, g, temp;
6     lsm.getEvent(&a, &m, &g, &temp);
7     String face = "?" ;
8     if (a.acceleration.z > 8)
9       face = "1" ;
10    else if (a.acceleration.z < -8)
11      face = "6" ;
12    else if (a.acceleration.x > 8)
13      face = "2" ;
14    else if (a.acceleration.x < -8)
15      face = "5" ;
16    else if (a.acceleration.y > 8)
17      face = "3" ;
18    else if (a.acceleration.y < -8)
19      face = "4" ;
20
21      Serial.println(face) ;
22      BT.println(face) ;
23
24    delay(1000);
25  }
```

Une fois le système bien calé, reste à changer son orientation et noter sur la bonne face. Le principe est simple, on utilise la projection de l'accélération sur les 3 axes en exploitant son signe.

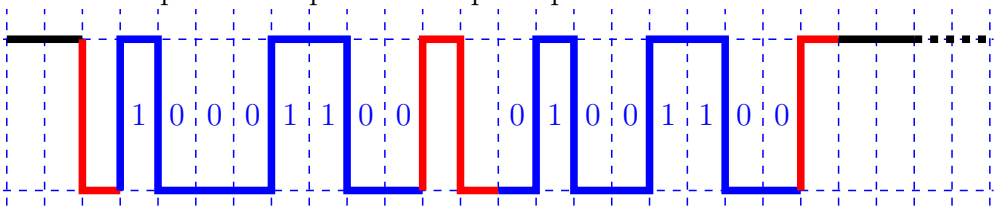
4.5 Chapitre 6

a write vs print

La trame correspondant à write s'interprète par :



L'octet transmit correspond à $2^2 + 2^3 = 4 + 8 = 12$, c'est-à-dire la valeur de l'entier que l'on souhaite transmettre !
Celle correspondant à print s'interprète par :



Le premier octet transmit correspond à $1 + 2^4 + 2^5 = 1 + 16 + 32 = 59$, c'est-à-dire la valeur du code ASCII du caractère **1**.
Le second octet transmit correspond à $2^2 + 2^4 + 2^5 = 2 + 16 + 32 = 60$, c'est-à-dire la valeur du code ASCII du caractère **2**.

On cherchait ici à transmettre la valeur d'un entier (codé sur 1 seul octet) :

- L'instruction **print** a transmis une chaîne de caractères... charge au programme implémenté sur la carte Arduino d'interpréter la réception d'une chaine de caractère.
- L'instruction **write** a transmis directement la valeur de l'octet correspondant... charge au programme implémenté sur la carte Arduino d'interpréter la réception d'un octet.