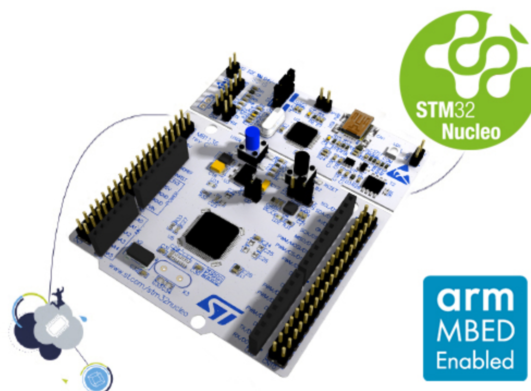
Institut d'Optique Graduate School
Interfaçage Numérique

INTERFAÇAGE NUMÉRIQUE

Travaux Pratiques

Semestre 6

Arduino et carte Nucléo-STM32



Arduino et carte Nucléo-STM32

Ce document est disponible au format électronique sur le site du LEnsE - https ://lense.institutoptique.fr/ dans la rubrique Année / Première Année / Interfaçage Numérique S6 / Arduino.

*Vous trouverez également des ressources concernant les **microcontrôleurs** et les **systèmes embarqués** à l'adresse suivante :*

https://iogs-lense-training.github.io/nucleo-basics/contents/general.html

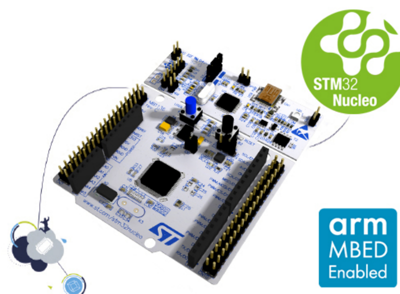
IDE Arduino

Arduino est une plateforme open-source utilisée pour créer des projets électroniques. Elle est composée de deux éléments principaux : **une carte matérielle** (contenant un microcontrôleur) et **un environnement de développement** (IDE Arduino) qui permet de programmer la carte.

Nous nous intéresserons ici qu'à l'environnement de développement qui, après installation d'une extension, permet de programmer d'autres cartes à microcontrôleurs.

Carte Nucleo-STM32

Les cartes Nucleo sont des **plateformes de développement** basées sur les **microcontrôleurs STM32** de *STMicroelectronics*. Elles sont conçues pour faciliter le prototypage et le développement de projets embarqués, similaires aux cartes Arduino, mais elles sont souvent utilisées pour des applications plus complexes et performantes.



Elles sont équipées d'un débogueur ST-LINK intégré, ce qui permet de programmer et de déboguer le microcontrôleur directement sans matériel additionnel.

Le brochage de la carte Nucleo L476RG est fournie en annexe à ce document : Brochage Nucléo L476RG

Installation des cartes STM32

L'interface de développement **Arduino**, ainsi que les bibliothèques associées, est populaire pour les projets *Do It Yourself*, l'éducation et le prototypage rapide en raison de sa simplicité et de son accessibilité. Cependant, elle est initialement prévue pour programmer des cartes de type **Arduino**.

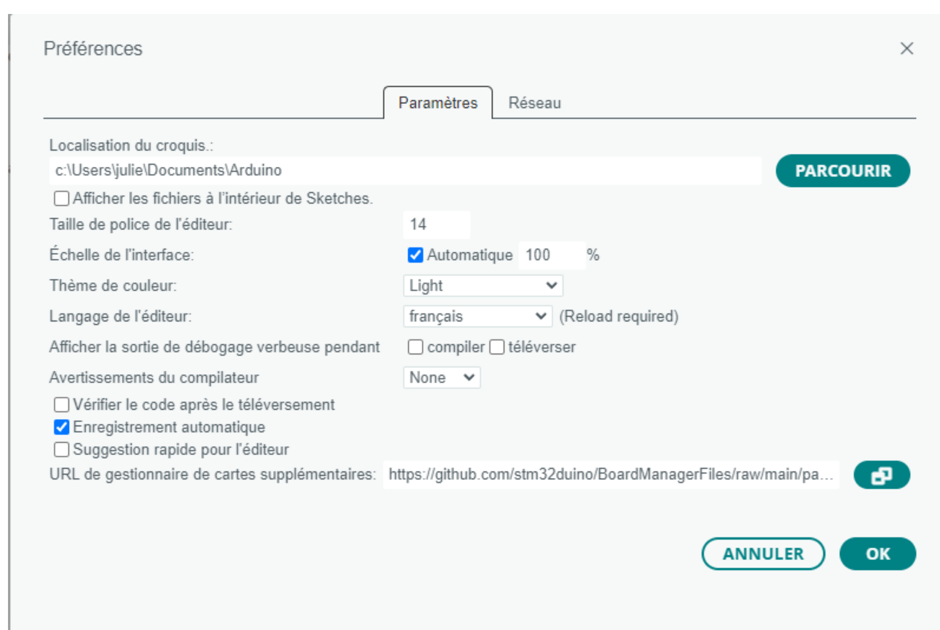
Pour pouvoir bénéficier de l'environnement **Arduino** pour **d'autres cartes de prototypage**, il est indispensable d'installer les extensions associées à ces autres cartes.

Attention ! La version 2 de l'IDE Arduino est fortement conseillée pour bénéficier des dernières évolutions du langage et de l'interface, ainsi que pour garantir une pleine compatibilité avec les cartes Nucléo.

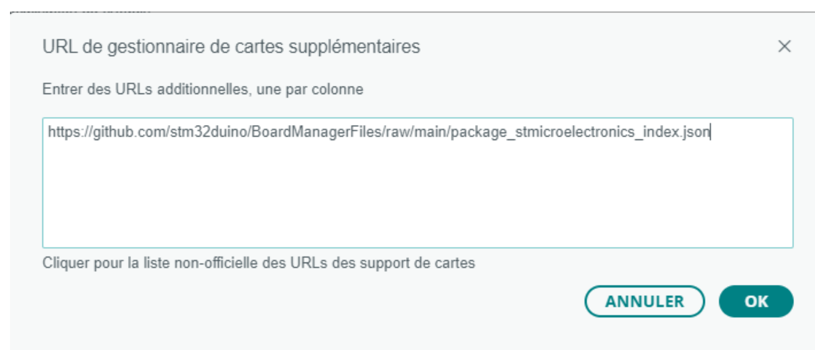
Support des cartes STM32

Avant de pouvoir utiliser l'environnement Arduino pour programmer des cartes intégrant des microcontrôleurs de type STM32, il faut installer le **support pour ces microcontrôleurs**.

Dans le menu **Fichiers** / **Préférences**, sélectionner le volet **Paramètres**.



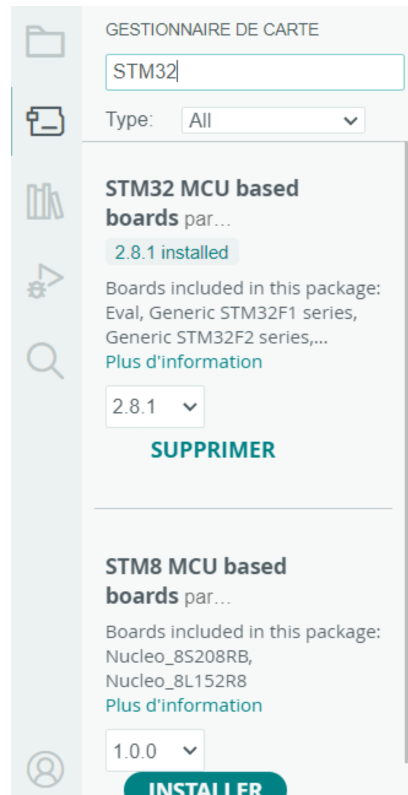
Dans la fenêtre URL de gestionnaire de cartes supplémentaires, ajouter l'adresse suivante :
https://GitHub.com/stm32duino/BoardManagerFiles/raw/main/package_stmicroelectronics_index.json



Extension STM32 MCU based boards

Il faut ensuite télécharger les **bibliothèques** liées aux cartes intégrant des **microcontrôleurs STM32** de *STMicroelectronics*.

Aller dans le menu **Outils** / **Carte** / **Gestionnaire de carte**.



Dans la partie droite de l'interface Arduino, un volet **Gestionnaire de carte** s'ouvre. Dans la zone de recherche, taper STM32.

Dans la liste, installer alors l'extension : **STM32 based boards par STMicroelectronics**.

Premier programme

Afin de vérifier que toute la chaîne de prototypage est opérationnelle, nous allons nous intéresser à un **programme de base** permettant de faire **clignoter une LED** présente par défaut sur la carte Nucléo (c'est également vrai sur les cartes Arduino).

Sélectionner **Fichier** / **Exemples** / **01.Basics** / **Blink** dans la barre de menu.

Le programme ressemble à celui-ci :

```
1  void setup() {
2    // initialize digital pin LED_BUILTIN as an output.
3    pinMode(LED_BUILTIN, OUTPUT);
4  }
5
6  void loop() {
7    digitalWrite(LED_BUILTIN, HIGH);
8     // turn the LED on (HIGH is the voltage level)
9    delay(1000);
10    // wait for a second
11   digitalWrite(LED_BUILTIN, LOW);
12    // turn the LED off by making the voltage LOW
13   delay(1000);
14    // wait for a second
15 }
```

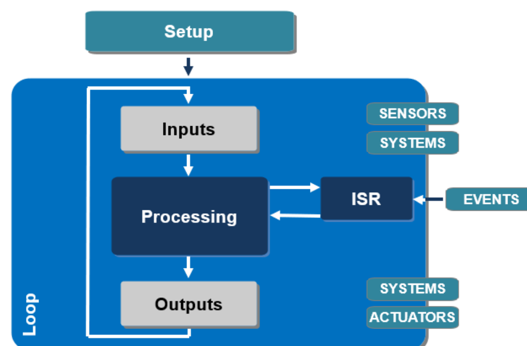
Sur la carte Nucléo la sortie LED_BUILTIN correspond à la LED nommée LD2.

Le langage utilisé par l'environnement Arduino est un langage proche du C++. Le programme ainsi écrit doit nécessairement **être compilé** avant de pouvoir **être téléversé** sur la carte où il sera ensuite **exécuté**.

Structure du code

Un programme Arduino (comme tout autre programme embarqué) est constitué de **deux étapes principales** :

- une **Initialisation** (fonction *setup()* pour Arduino) : exécutée une fois à la mise sous tension de la carte ou lors de l'appui sur le bouton Reset.
- une **boucle infinie** (fonction *loop()* pour Arduino) : exécutée de manière infinie. Cette boucle a pour principale mission, sur un système embarqué, de récupérer les valeurs des entrées, de calculer les valeurs des sorties et de mettre à jour les sorties.



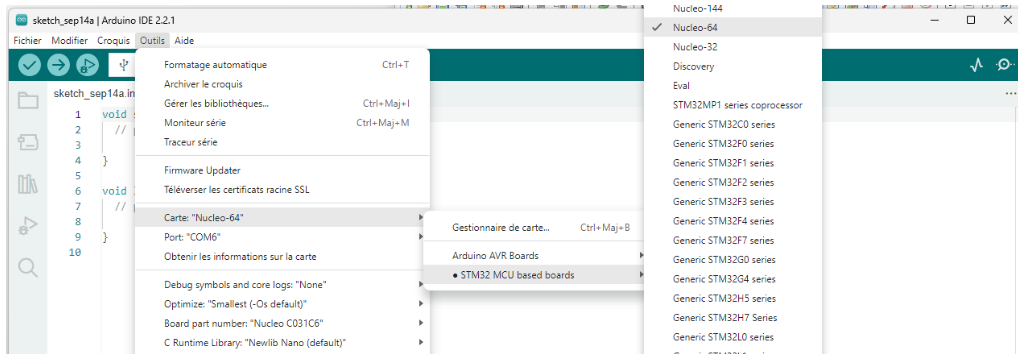
D'autres étapes sont possibles lorsqu'on autorise le fonctionnement par interruption (voir dans la suite de ce document).

Choix d'une carte

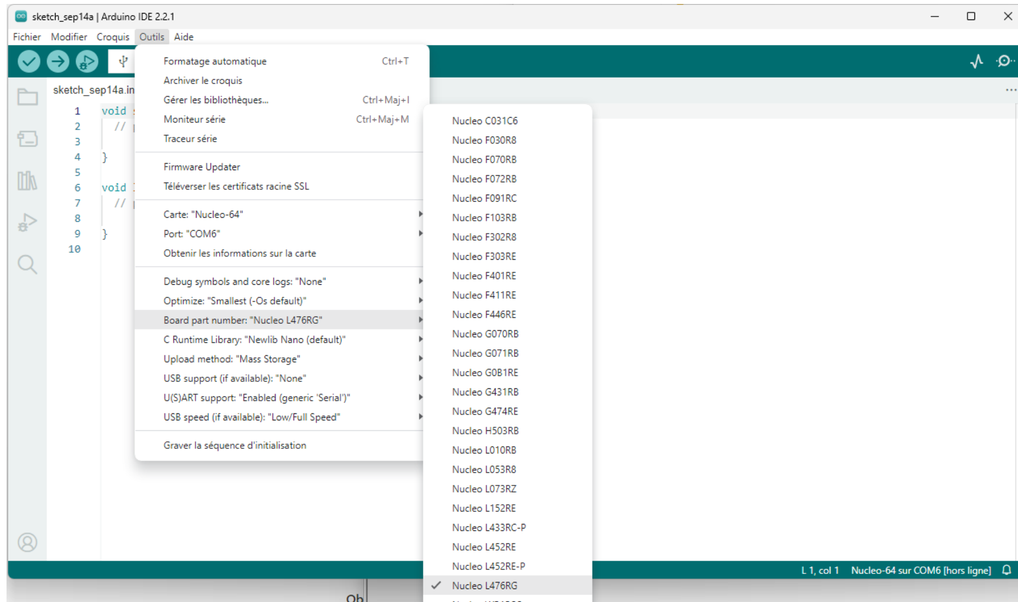
La compilation d'un tel programme se fait pour une cible particulière. Avant de pouvoir compilé, il est donc nécessaire de préciser sur quel microcontrôleur (ou quelle carte de prototypage) ce code sera exécuté.

Pour cela, dans la barre de menu, sélectionner **OUTILS** / **CARTES**.

Dans le cas d'une carte Nucléo de type L476RG, sélectionner ensuite **STM32 MCU BASED BOARDS** / **NUCLEO-64**. Le format pourra changer s'il s'agit d'une autre carte.

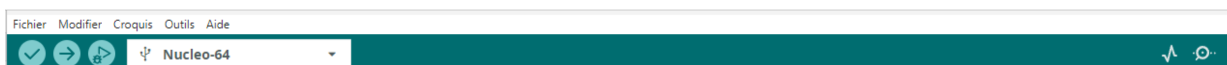


Puis, dans le menu **OUTILS** / **BOARD PART NUMBER**, sélectionner **Nucleo L476RG**.



Compilation

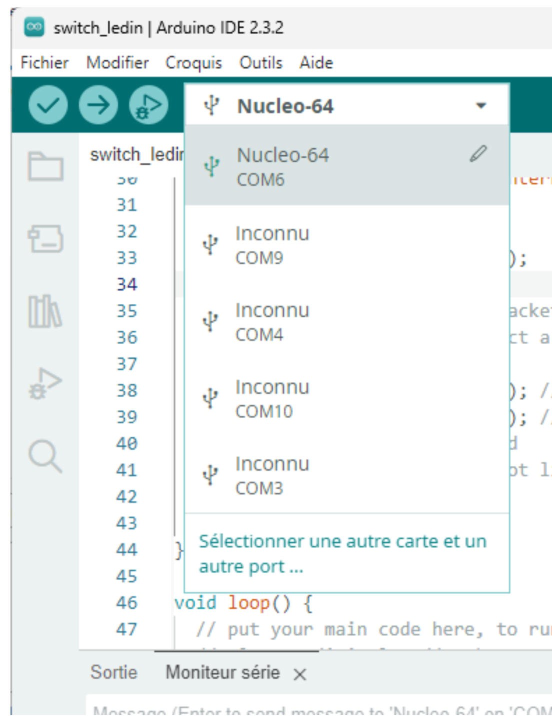
Il est maintenant possible de compiler le programme. Pour cela, cliquer sur la première icône de la barre d'action :



Connexion à une carte Nucléo et téléversement

Il faut ensuite connecter la carte en USB.

Dans la barre des actions possibles sous Arduino, sélectionner le port de communication sur lequel est connectée la carte Nucléo.



Pour téléverser ensuite le code dans la carte, cliquer sur la seconde icône de la barre d'actions (en forme de flèche vers la droite).

Définition de constantes

En langage C++, la définition de constante peut se faire de la façon suivante :

```
1  #define PI 3.14159
```

Cette méthode pourra s'avérer utile pour définir les noms des entrées et des sorties de notre application embarquée. Par exemple :

```
1  #define    SW1       PC6
2  #define    USER_B    PC13
3  #define    LED1      PC7
```

Entrées-Sorties Numériques

Afin de pouvoir interagir avec le monde extérieur, les microcontrôleurs disposent d'un ensemble d'**entrées** et de **sorties**.

Chacune de ces entrées-sorties portent un nom, au format PX_N, où X est le nom du port (A, B...) et N le numéro de la broche.

Toutes les broches peuvent être utilisées en **entrée** ou en **sortie numérique**, c'est à dire un type de signal qui ne peut prendre que **deux états** : haut ou bas (aussi appelés 1 ou 0, ou encore *HIGH* et *LOW* en Arduino).

Certaines broches ont également d'autres fonctionnalités : entrées analogiques, sorties modulées PWM, communication série...

Sorties numériques

Paramétrage

Pour **configurer une broche en sortie**, il faut ajouter l'instruction suivante dans la fonction *setup()* (où *LED1* est le nom d'une broche du composant) :

```
1 pinMode(LED1, OUTPUT);
```

Utilisation

Pour **affecter une valeur à une broche en sortie**, il faut utiliser une des deux instructions suivantes (où *LED1* est le nom d'une broche du composant) selon que l'on veut mettre la sortie à l'état bas (*LOW*) ou à l'état haut (*HIGH*) :

```
1 digitalWrite(LED1, LOW);
2 digitalWrite(LED1, HIGH);
```

Entrées numériques

Paramétrage

Il n'y a aucun paramétrage à effectuer pour configurer une broche en entrée. Elles sont toutes **par défaut** paramétrer dans cette direction, notamment pour protéger les autres périphériques associés à la carte.

Utilisation

Pour **récupérer la valeur d'une broche en entrée**, il faut utiliser l'instruction suivante (où *SW1* est le nom d'une broche du composant) :

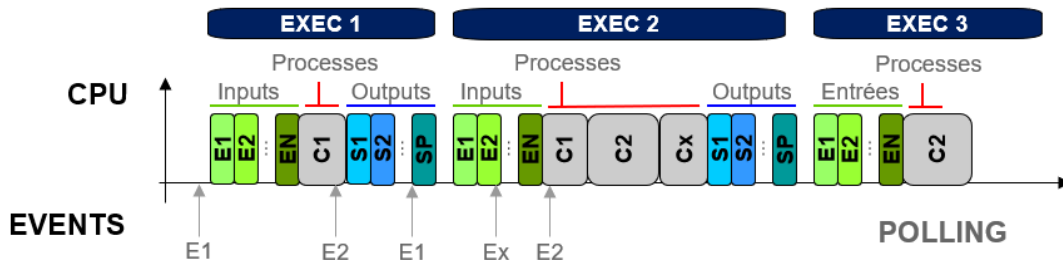
```
1 bool valSw1 = digitalRead(SW1);
```

La variable *valSw1* contient alors l'état de l'entrée *SW1* et peut valoir ***true*** (état logique haut) ou ***false*** (état logique bas).

Scrutation

La **scrutation** (ou *polling* en anglais) est une méthode de **vérification régulière de l'état des périphériques ou des capteurs** dans un système embarqué pour détecter si un événement spécifique s'est produit.

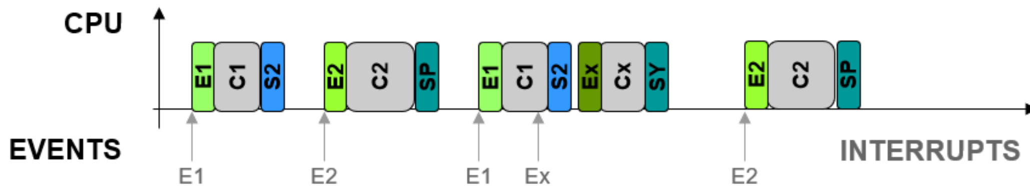
Dans ce contexte, le programme principal exécute une boucle continue (fonction *loop()* sous Arduino) où il interroge périodiquement chaque périphérique pour voir s'il nécessite une action.



La **scrutation** ne permet pas de gérer d'un évènement rapidement et monopolise le microcontrôleur pour vérifier constamment l'état des capteurs ou des périphériques.

Interruptions

Une **interruption** est un signal envoyé au microcontrôleur pour lui demander d'**arrêter temporairement l'exécution de son programme principal** et de **s'occuper d'une tâche prioritaire spécifique**. Lorsque l'interruption est terminée, le microcontrôleur reprend son programme là où il s'était arrêté.



Cette méthode permet au système de **réagir rapidement à des événements externes** - changement de signal sur une broche, compteur qui atteint une certaine valeur - sans avoir besoin de vérifier constamment si l'événement a eu lieu, contrairement à la scrutation.

Pour en savoir plus sur les interruptions, vous pouvez consulter le document à l'adresse suivante :
https://iogs-lense-training.github.io/nucleo-basics/contents/polling_interrupts_rtos.html

Les interruptions sont également souvent utilisées dans les applications où un timing précis est nécessaire (par exemple, un compteur de temps).

Paramétrage

Pour **configurer une broche en entrée permettant une interruption**, il faut ajouter l'instruction suivante dans la fonction *setup()* (où *LED1* est le nom d'une broche du composant) :

```
1 pinMode(SW1, INPUT_PULLUP);
2 attachInterrupt(digitalPinToInterrupt(SW1), swInt, FALLING);
```

Il existe 3 modes possibles pour les interruptions sur des signaux externes (entrées numériques) :
- **RISING** - front montant d'un signal
- **FALLING** - front descendant d'un signal
- **CHANGE** - fronts montant et descendant d'un signal

Routine d’interruption

Une **routine d’interruption**, aussi appelée **ISR** (*Interrupt Service Routine*), est la fonction qui s’exécute automatiquement en réponse à une interruption.

C’est une fonction classique, mais qui ne retourne pas de donnée.

```
1  void swInt(void){
2    bool ledState = digitalRead(LED1);
3    digitalWrite(LED1, !ledState);
4  }
```

Dans cet exemple, à chaque front descendant sur l’entrée *SW1*, la fonction *swInt()* est appelée. Elle lit alors l’état de la sortie *LED1* pour ensuite l’inverser.

Liaison Série et débogage

Lorsqu'on développe une **application sur un ordinateur**, il est facile de rajouter des lignes supplémentaires pour afficher des informations de débogage : soit sur une console, soit dans une partie de l'interface graphique développée.

En **programmation embarquée**, le microcontroleur n'a pas accès à un écran pour pouvoir communiquer facilement ce type d'informations. Il est donc plus difficile de connaître l'état de progression d'exécution du programme ou la valeur d'une variable à un moment donné du programme.

Il existe deux techniques permettant de répondre à cette attente :

- à l'aide d'une sortie externe et d'un oscilloscope (et/ou d'une LED)
- à l'aide d'une liaison série entre la carte et l'ordinateur et d'un logiciel pour communiquer sur ce type de liaison

Nous allons nous intéresser ici à la seconde méthode, la première méthode proposée ne permettant pas de connaître la valeur d'une variable à un instant donné dans le programme.

Liaison Série

Pour connaître la valeur d'une variable à un instant donné dans le programme ou tout simplement afficher un message à l'utilisateur, nous allons nous servir d'une **liaison série de type RS232**, disponible sur les ordinateurs (par l'intermédiaire de la liaison USB pour Arduino) et sur la plupart des microcontroleurs.

Sur une liaison de type RS232, les données sont envoyées un bit à la fois en une seule file, ce qui rend la communication série simple et fiable pour les longues distances. Les données sont envoyées par **paquet de 8 bits** (soit un octet).

La transmission est **asynchrone** et les deux noeuds qui communiquent par ce protocole doivent donc accorder leur vitesse de transmission (*baud rate* en anglais). Sur Arduino, on utilise couramment des vitesses comme 9600 bauds, 57600 bauds ou 115200 bauds.

Paramétrage

Par défaut, la liaison série configurée est celle utilisée par l'ordinateur (liaison émulée par le câble USB). Lors de l'initialisation, il faut préciser la vitesse de transfert (en bauds).

```
1  Serial.begin(9600);
2  while(!Serial);
```

Envoi de données

```
1  Serial.print("LEnsE␣!");
```

```
1  Serial.println(val_pot_m);
```

Réception de données

Moniteur Série Arduino

Entrées analogiques et échantillonnage

Acquisition d'un échantillon

où *POT_IN* est le nom d'une broche du composant correspondant à une entrée analogique :

```
1 int analogValue = analogRead(POT_IN);
```

La fonction *analogRead()* renvoie une valeur entière entre 0 et 1023 (10 bits), correspondant à la quantification de la tension présente sur la broche d'entrée analogique passée en paramètre.

Cette valeur, sur les cartes Nucléo, vaut :

$$N = \frac{V_{in}}{3.3\,\mathrm{V}} * 1023$$

Il est alors possible d'afficher cette tension à l'aide de la liaison série :

```
1 Serial.print("Pot = ");
2 Serial.println(analogValue);
```

Et si l'on veut afficher la valeur associée de la tension :

```
1 Serial.print("Pot = ");
2 Serial.print(analogValue * 3.3 / 1023);
3 Serial.println(" V");
```

Attention ! L'envoi de données sur la liaison série, pour l'affichage des valeurs par exemple, prend du temps ! Ce temps dépend bien entendu de la taille du message à transmettre mais également de la vitesse de transmission utilisée sur la liaison série.

Echantillonnage à intervalle régulier

Paramètrage

```
1 #if defined(TIM1)
2   TIM_TypeDef *Instance = TIM1;
3 #else
4   TIM_TypeDef *Instance = TIM2;
5 #endif
6   HardwareTimer *MyTim = new HardwareTimer(Instance);
7   MyTim->setOverflow(10, HERTZ_FORMAT); // 10 Hz
8   MyTim->attachInterrupt(samplingIsr);
9   MyTim->resume();
```

Dans l'exemple précédent, la fonction *samplingIsr()* est appelée à intervalle régulier, ici à 10 Hz.

Routine d'interruption

```
1 int analogValue = 0;
2 bool samplingOk = false;
```

```
1 void samplingIsr(){
2   analogValue = analogRead(POT_IN);
3   samplingOk = true;
4 }
```

```
1  void loop() {
2    if(samplingOk == true){
3      Serial.print("Pot␣=␣");
4      Serial.println(valPot);
5      samplingOk = false;
6    }
7  }
```

Sorties modulées en largeur d'impulsion (PWM)

Pilotage de la luminosité d'une LED

Servomoteur

Branchement alternatif - STM32

Toutes les sorties PWM des cartes Nucléo ne sont **pas nativement configurées pour fonctionner avec Arduino**. On parle de **broches alternatives** (ou fonctions alternatives sur certaines broches).

Si on souhaite pouvoir les utiliser avec Arduino, il est indispensable de rajouter une ligne de configuration dans la fonction *setup()* :

```
1  LL_GPIO_SetAFPin_0_7(GPIOB,  GPIO_PIN_7,  GPIO_AF1_TIM2);
```

Cette ligne permet de modifier le paramétrage de la broche **PB7** afin qu'elle puisse fonctionner en PWM (ou avec la bibliothèque *Servo.h*.

Liaisons I2C et SPI

Protocole I2C

Principe

Liaison physique

Mise en place sous Arduino

Protocole SPI

Principe

Liaison physique

Mise en place sous Arduino

Carte Nucléo-64 / STM32L476 / Broches d'entrées-sorties

INSTITUT d'OPTIQUE GRADUATE SCHOOL ParisTech
LEnsE Laboratoire d'Enseignement Expérimental