

Dossier technique du projet - partie individuelle

Table des matières

1 - SITUATION DANS LE PROJET.....	3
1.1 - RAPPEL DES TÂCHES PROFESSIONNELLES À RÉALISER	3
1.2 - PRÉSENTATION DE LA PARTIE PERSONNELLE.....	3
1.2.1 - Introduction.....	3
1.2.2 - Progression.....	3
1.2.3 - Synoptique de la réalisation.....	4
1.2.4 - Aperçu de la maquette.....	5
1.2.5 - Les interfaces graphiques.....	6
1.2.6 - Logiciels utilisés.....	7
1.2.7 - Diagramme de séquence.....	8
1.2.8 - Le traitement d'une valeur.....	9
1.2.8.1 - L'interpréteur de commande shell_mega28.....	9
1.2.8.2 - Valeur de retour.....	9
1.2.8.3 - Le parser mathématiques.....	10
1.2.9 - Diagramme de classe.....	11
2 - RÉALISATION DE LA TÂCHE PROFESSIONNELLE : COMMUNICATION AVEC LE PORT SÉRIE.....	11
2.1 - CONCEPTION DÉTAILLÉE.....	11
2.1.1 - Diagramme de classe.....	12
2.1.2 - Communication avec la carte.....	13
2.1.3 - Diagramme de séquence.....	13
2.2 - TESTS UNITAIRES : ACQUISITION D'UNE VALEUR.....	14
2.2.1 - Identification du test unitaire.....	14
2.2.2 - Objectif du test.....	14
2.2.3 - Procédure de test.....	14
2.2.4 - Rapport d'exécution.....	15
2.3 - PROBLÈMES RENCONTRÉS.....	15
3 - RÉALISATION DE LA TACHE PROFESSIONNELLE : CONVERSION D'UNE VALEUR.....	15
3.1 - CONCEPTION DÉTAILLÉE.....	15
3.1.1 - Présentation du parser.....	15
3.1.2 - Diagramme de classe.....	15
3.1.3 - Diagramme de séquence.....	16
3.1.4 - Codage du parser.....	16
3.1.4.1 - Configuration du parser.....	16
3.1.4.2 - Préparation des éléments de conversion.....	17
3.2 - TEST UNITAIRE : CONVERTIR UNE VALEUR.....	17
3.2.1 - Identification du test unitaire.....	17
3.2.2 - Objectif du test.....	17
3.2.3 - Procédure de test.....	18
3.2.4 - Rapport d'exécution.....	18
3.3 - PROBLÈMES RENCONTRÉS.....	18
4 - RÉALISATION DE LA TÂCHE PROFESSIONNELLE : CRÉATION DU SÉMAPHORE.....	19
4.1 - CONCEPTION DÉTAILLÉE.....	19
4.1.1 - Présentation du sémaphore.....	19
4.1.2 - Diagramme de séquence.....	19

4.1.3 - Codage du sémaphore.....	20
4.1.3.1 - Constructeur : Création du sémaphore.....	20
4.1.3.2 - Balisage de la section critique.....	20
4.2 - PROBLÈME RENCONTRÉS.....	21
5 - BILAN DE LA RÉALISATION PERSONNELLE.....	21

1 - Situation dans le projet

1.1 - Rappel des tâches professionnelles à réaliser

Le projet se divise en 3 grandes parties. Dans l'ordre d'utilisation pour l'utilisateur, la première est celle de Sami qui consiste dans la configuration d'une expérience ou d'un chargement d'une expérience déjà effectuée. Ensuite, celle faite par Dylan consiste dans le paramétrage d'une expérience puis l'affichage des valeurs mesurées. Enfin, ma partie est la seule où il n'y a quasiment pas d'IHM. La grosse majorité de mon travail se concentrait sur la lecture des données en provenance des capteurs, de leur traitement puis du renvoi celles-ci vers la partie de Dylan.

Mes fonctions, développées au cours de ce projet, sont :

Fonctions Développées	Description
FP2 : Visualisation des valeurs mesurées	Communication avec la carte Arduino pour avoir la valeur mesurée. Traitement de la valeur puis utilisation d'un parser mathématiques si besoin.
Création et mise en place d'un sémaphore	Codage d'un sémaphore afin de pouvoir gérer plusieurs capteurs en même temps
FS3 : Exporter les valeurs mesurées	En fin d'expérience, l'utilisateur doit avoir la possibilité d'exporter les résultats de tous les capteurs dans un tableau.

1.2 - Présentation de la partie personnelle

1.2.1 - Introduction

Ma partie du projet consiste dans la configuration et l'utilisation de toutes les communications faite entre la carte Arduino MEGA et le PC de mesure. Cela concerne la communication PC vers carte Arduino(demande d'une valeur) mais aussi carte Arduino vers PC de mesure (Valeur mesuré) . Le traitement des données retournées par le capteur et la conversion de ces données à l'aide d'un parser mathématique en fonction du capteur utilisé.

Ensuite, , il faudra pouvoir gérer le cas où plusieurs capteurs seront utilisés en même temps. Pour cela, je vais utiliser un sémaphore qui ne va autoriser qu'un capteur à la fois à utiliser le port série.

Enfin, la dernière chose que je devrais faire sera l'exportation de l'entièreté des données mesurées de façon à ce que ce soit exploitable par un tableau. Cette exportation se fera à l'aide du format CSV qui permet de séparer chaque valeur de chaque cellule par un même caractère, souvent une virgule. C'est de là que vient le nom du format : **Comat Separated Value**.

1.2.2 - Progression

Le projet s'est déroulé sur plusieurs sprints tout du long du projet :

SPRINT 0 :

- Lecture du cahier des charges
- Éclaircissement des points incompris
- Prise en main du matériel
- Installation et prise en main des logiciels utilisés
- Réalisation des diagrammes UML du projet
- Documentation du projet

SPRINT 1 :

- Le codage d'ébauche du projet de la liaison série.
- Prise en main de l'interpréteur de commande de la carte d'Arduino
- Réalisation et correction de diagrammes UML/SysML
- Documentation de la liaison série
- Création de la classe utilisant QSerialPort

SPRINT 2 :

- Recherche du sémaphore
- Codage du parser mathématique
- Documentation sur le parser
- Diagramme UML des nouvelles parties terminées

SPRINT 3 :

- Codage du sémaphore
- Débogage la mesure.
- Préparation des différents oraux

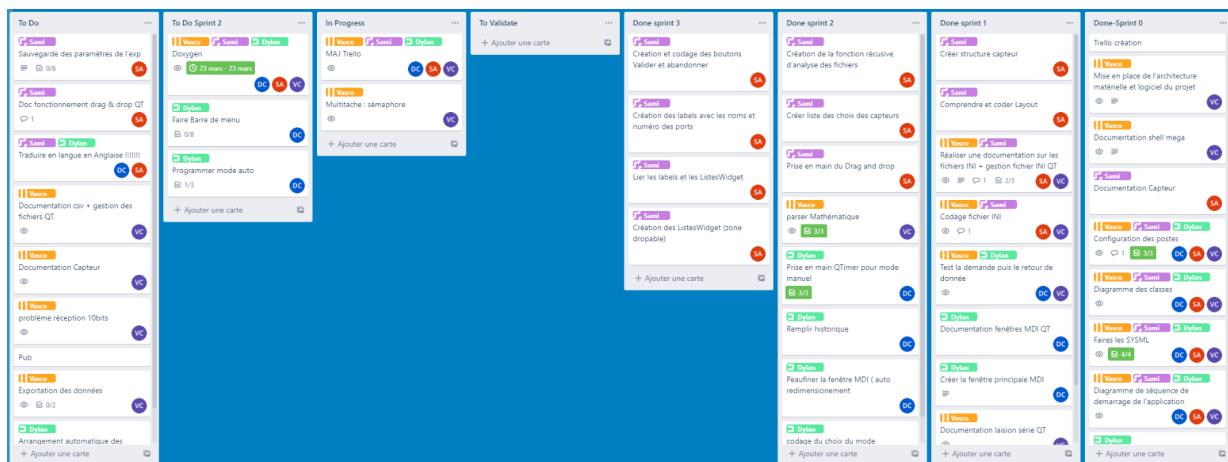


Figure 1: Le Trello du projet

1.2.3 - Synoptique de la réalisation

Notre projet se décompose en plusieurs éléments physiques, le plan de branchement est comme ci-après :

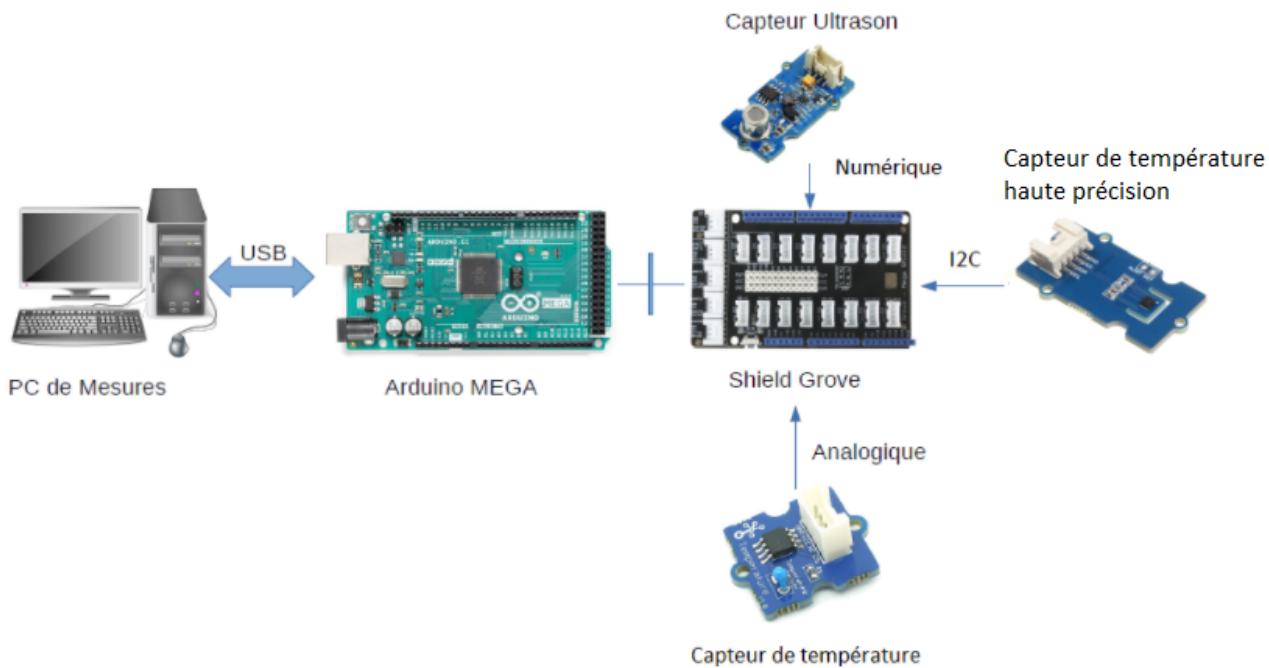


Figure 2: Synoptique général du projet

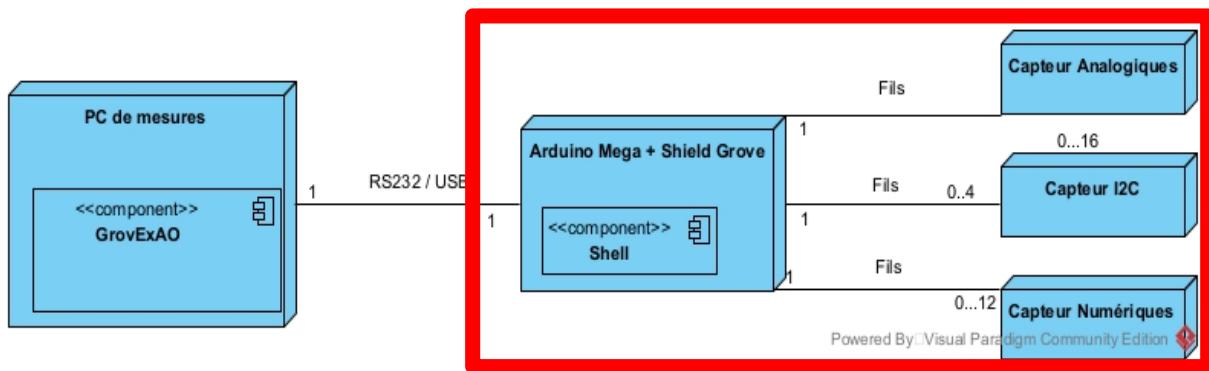


Figure 3: Diagramme de déploiement

Le diagramme de déploiement et le synoptique nous permet de décomposer le projet comme ceci :

- **PC de mesures** : L'appareil sur lequel le logiciel sera fonctionnel. Il est pour l'instant sous Windows, mais par la suite le logiciel sera également fonctionnel sur Linux. Il est connecté par un câble USB à la carte Arduino.
- **Arduino MEGA** : Carte sur laquelle est branchée le shield Grove. Elle est utilisée pour la communication avec les capteurs et a dans sa carte mémoire l'interpréteur de commande pour pouvoir recevoir les données de tous les capteurs branchés grâce à la liaison série/USB.
- **Les capteurs** : Les capteurs mesurent une grandeur physique puis retournent la valeur à la carte Arduino.

1.2.4 - Aperçu de la maquette

Au départ du projet la maquette ci-dessous était déjà faite, nous avons poursuivi la conception du projet sur cette maquette

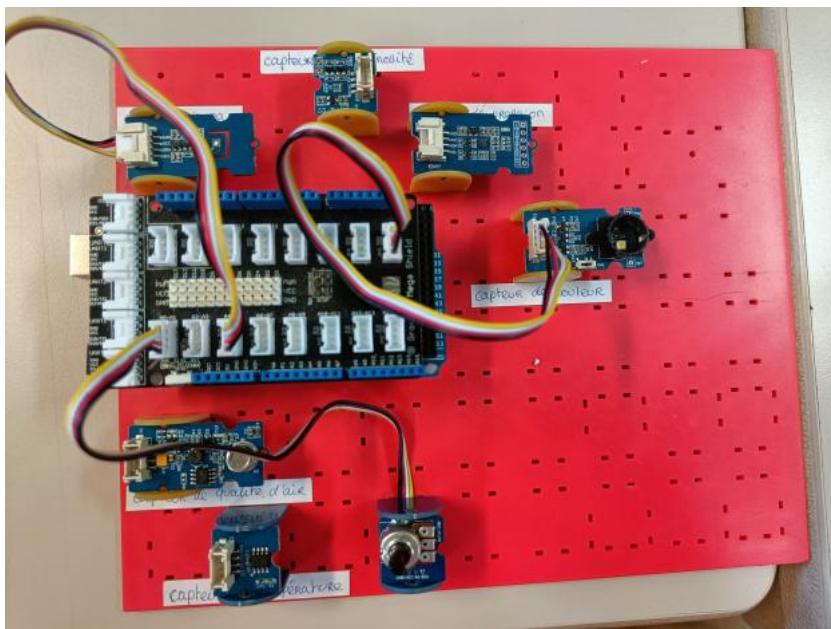


Figure 5: Aperçu globale de la carte

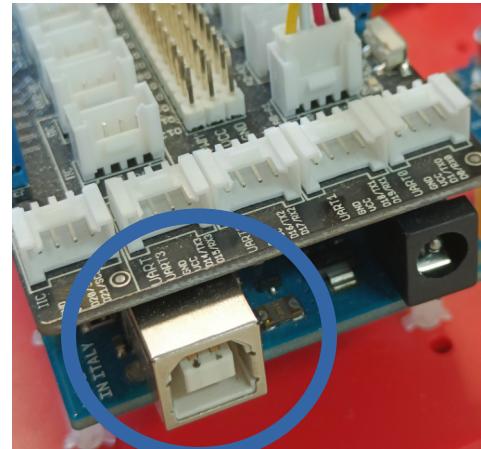


Figure 4: Port série qui permet la connexion avec le PC

1.2.5 - Les interfaces graphiques

L'application fonctionne avec 3 fenêtres différentes :

La toute première fenêtre est celle où le choix est laissé à l'utilisateur de sa prochaine action. Il lui est possible de charger une expérience déjà faites, d'en créer une nouvelle, exporter les valeurs, lancer l'acquisition et enfin de changer la langue du logiciel.

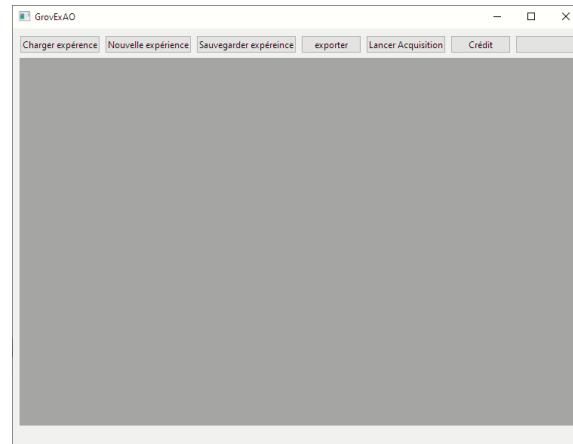


Figure 6: Fenêtre 1 : Accueil

La deuxième fenêtre est ensuite, le choix des capteurs, où l'utilisateur va choisir, en fonction de ses besoins, les capteurs et leurs emplacements.

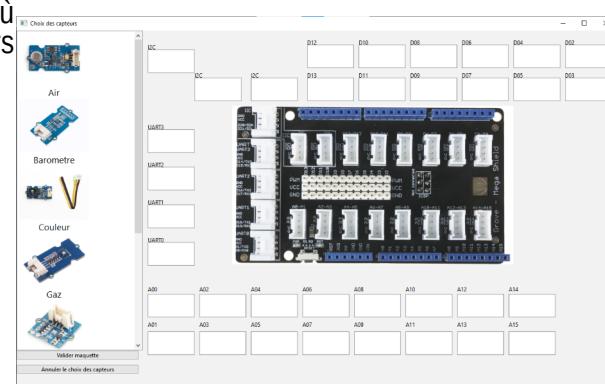


Figure 7: Paramétrage de l'expérience

Le dernier type de fenêtre est la visualisation des valeurs pour chaque capteur choisi pour l'expérience.

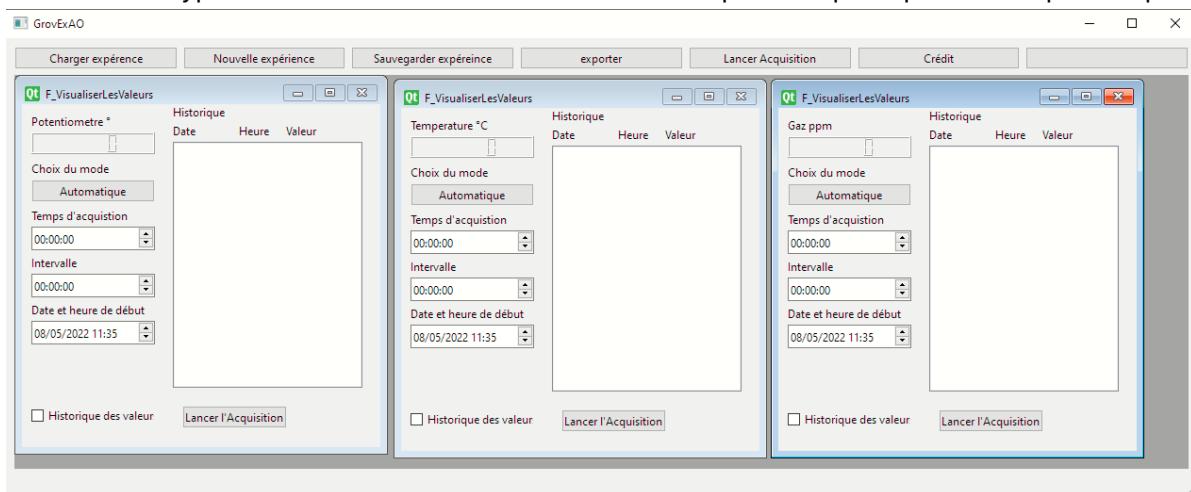


Figure 8: Fenêtre 3 : Visualisation des valeurs

1.2.6 - Logiciels utilisés

Logiciel	Utilisation
	Qt Creator est un IDE multi-plateforme. Il permet en premier lieu de coder facilement du C++ mais également du JavaScript ou encore du Python. Nous utilisons la version Qt 6.2 qui bénéficie du Long Term Support Release, ce qui signifie que Qt assure des mises à jour régulières pendant longtemps. Cette version est la version qui possède le module Qt SerialPort
 LibreOffice The Document Foundation	Suite multi-plateforme pour le traitement de texte.
Doxxygen	Documentation automatique de programme informatique, puis création d'un site internet regroupant toutes les documentations créée.
	Logiciel de création de tous les diagrammes UML/SysML du projet
	Trello est le logiciel permettant de facilement de faire le management du projet. Grâce à cette plateforme, nous pouvons facilement et rapidement savoir ce qu'il reste à faire comme tâche que ce soit pour nous ou pour les autres. Il est également utile pour diviser les différents sprints du projet.
	GitLab est logiciel libre de forge, créé sur la base de git, et propose des outils de maintenance et de suivis de projet. Cet outil nous servira à déposer tout notre projet et à toujours avoir notre projet pour qu'on puisse tous avoir les mêmes fichiers et aussi de pouvoir travailler chez nous.
	Arduino IDE nous a servi au début du projet puis lors de phases de test pour communiquer avec la carte et compiler un programme. C'est grâce à lui que l'interpréteur de commande a pu être installé puis être testé.

1.2.7 - Diagramme de séquence

Ma partie qui se concentre sur l'acquisition d'une valeur, du traitement puis du retour de cette valeur se déroule de cette façon :

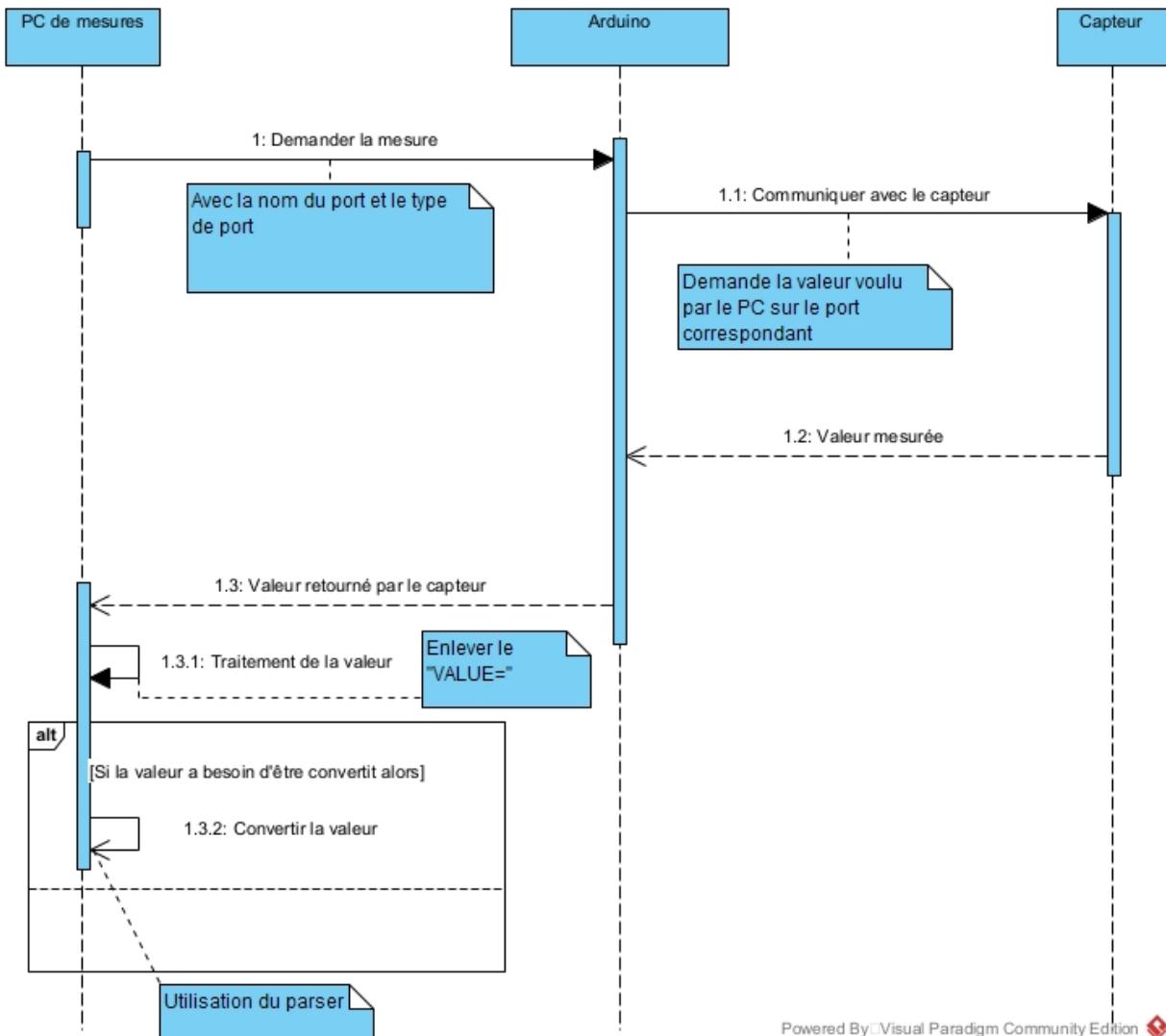


Figure 9: Diagramme de séquence : Acquisition d'une valeur

En premier, le PC de mesure, qui sait à quel capteur il doit demander la valeur, communique avec la carte le port et le type de port. Pour continuer la carte reçoit cette information et demande ensuite, au capteur ciblé la valeur du capteur voulu. Pour finir, la valeur est renvoyée vers le PC de mesure qui va ensuite être traité puis convertis. La suite du programme se trouve à la fenêtre 3 Visualiser les valeurs (voir fig. 8 page 6).

1.2.8 - Le traitement d'une valeur

1.2.8.1 - L'interpréteur de commande shell mega28

Pour pouvoir communiquer avec la carte et ainsi pouvoir recevoir les valeurs de tous les capteurs possibles du shield GROVE, j'ai installé dans la carte Arduino un interpréteur de commande. Cet interpréteur de commande se nomme « **shell_mega28** » conçu par l'entreprise **Technozone51** et programmé par Christophe GROSSE. Cet interpréteur de commande nous servira pour communiquer avec la carte Arduino pour lui indiquer sur quel port on veut avoir la valeur en indiquant le numéro du port et le type de capteur (Numérique ou analogique). Voici comment fonctionne l'interpréteur de commande. En premier, il faudra demander de quel capteur on veut avoir la valeur, tout ceci se passe donc à l'étape « **Demander la mesure** » du diagramme de séquence précédent.

Tout d'abord, il faudra différencier les deux types de valeurs qu'il pourra être retournées :

- Lettre « R » : Retour d'une valeur numérique (La valeur de retour est 1 ou 0)
- Lettre « A-a » : retour d'une valeur analogique (La valeur de retour est comprise entre 0 et 2^n-1)

Pour les valeurs analogiques, il y aura un autre paramètre à prendre en compte qui est le nombre de bit par message. Si nous voulons un message sur 8 bits, il faudra écrire le message d'écriture avec un « **A** » sinon ça sera avec 10 bits et la lettre sera « **a** »

Ensuite, il faudra désigner le numéro de port pour lequel nous voulons savoir la valeur mesurée. Il faudra simplement ajouter après la lettre correspondante la numérotation du port à la suite.

Exemple :

Je veux savoir la valeur du capteur :

- Branché au port analogique 02 sur 10 bits : **a02**
- Branché au port numérique 10 : **R10**
- Branché au port analogique 16 sur 8 bits : **A16**

La librairie QSerialPort n'accepte pas les trames 10 bits, le PC et la carte communiqueront donc exclusivement avec des trames de 8 bits.

1.2.8.2 - Valeur de retour

Après avoir communiqué avec la carte quel capteur le PC veut avoir la mesure, elle retourne cette valeur demandée. À ce moment, le format de valeur renvoyée est du format : **VALUE=<ValeurMesurée>**. L'objectif du premier traitement sera d'enlever les caractères **VALUE=** afin de n'avoir que la valeur mesurée.

```
void Arduino::DonneMesure()
{
    QByteArray ValeurAAfficher;
    QByteArray TableauARetournee;
    QString Formule;

    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    TableauARetournee = Serie.readAll();
    Valeur.append(TableauARetournee);

    //Traitement de chaîne de Valeur
    //Garder uniquement la valeur mesurée
    bool ok (false);
    unsigned int IndiceFinChaine (0);
    double ValeurMesureTraitePreTraitement (0);
    double ValeurMesureTraitePostTraitement (0);
    QString CharactereATrouver ("=");
    QString PetiteChaine;

    IndiceFinChaine = Valeur.lastIndexOf(CharactereATrouver);

    PetiteChaine = Valeur.remove(0, IndiceFinChaine+1);
```

```
PetiteChaine.chop(2);  
  
ValeurMesureTraitePreTraitement = PetiteChaine.toFloat(&ok);  
  
emit (NouvelleDonneeMesuree (this->sPort, ValeurMesureTraitePostTraitement));  
}  
}
```

Le code source ci-dessus représente la lecture du port, souligné en jaune. Une méthode `readAll()` permet de lire la valeur contenu dans la liaison série dans un tableau d'octet. Après avoir récupérer la valeur mesurée, il faut ensuite traiter la valeur pour qu'elle soit affichable dans la fenêtre de visualisation des valeurs. Cette partie est souligné en vert. Comme la forme du retour de la valeur de la carte est tout le temps la même, il est aisément de pouvoir facilement n'avoir que la valeur mesurée. Grâce aux méthodes « `lastIndexOf` » (révèle le dernier emplacement d'un caractère d'une chaîne) et « `remove` » (suppression de caractère jusqu'à un emplacement d'une chaîne) de la librairie `QString`. Nous avons maintenant à disposition uniquement la valeur mesurée informations inutiles.

1.2.8.3 - Le parser mathématiques

Cependant certains capteurs, comme le capteur de température, ont besoin d'avoir leurs valeurs convertis à l'aide d'une formule mathématique afin d'être exploitable par l'utilisateur. Cette formule, si elle existe, est trouvable dans la documentation du capteur. Pour ceci, nous allons utiliser un parser mathématique qui va transformer la chaîne de caractère contenant la formule de conversion en une formule mathématique contenant une variable qui sera la valeur mesurée. Pour ce faire, je vais utiliser un parser nommé **MuParserX** qui fera cette conversion et la calcul.

Cette formule sera contenue dans une structure nommée **Capteur**, cette structure sera présente pour chaque capteur utilisé et contiendra toutes les informations nécessaires à l'expérience :

- Nom
- Unité
- Formule mathématiques
- Type
- Port

Comme nous savons le port sur lequel est branché le capteur, il est facile de savoir s'il y a une formule ou non afin de faire une conversion.

1.2.9 - Diagramme de classe

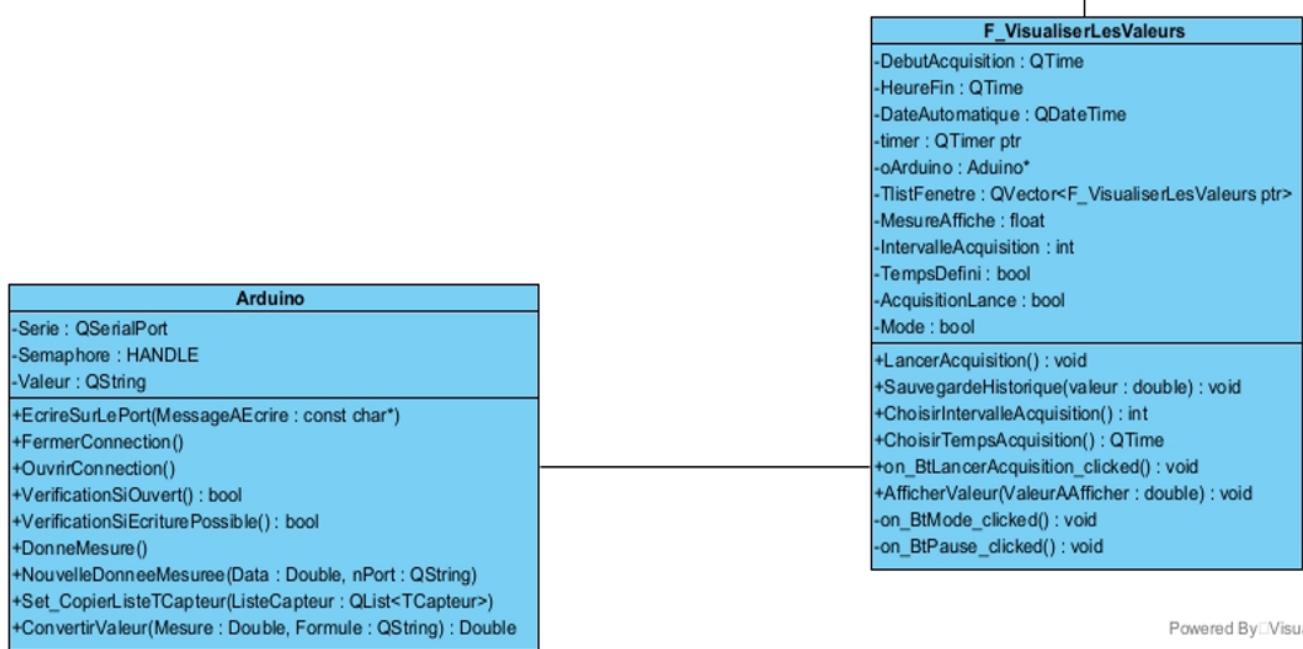


Figure 10: Diagramme de classe de ma partie

Powered By: Visual

Ma partie du projet s'est passé sur une classe :

- La classe **Arduino** : Classe créatrice et gérante de la liaison série. C'est cette classe qui reçoit et traite la valeur puis la retourne vers la classe **F_VisualiserLesValeurs**
- La classe **F_VisualiserLesValeurs** : Classe demandeuse et afficheuse de la valeur mesurée (classe créée par Dylan Etudiant 3)

Après avoir configuré et lancer une acquisition, une fenêtre de visualisation des valeurs pour chaque capteurs utilisés sera ouverte. A chaque moment d'affichage d'une valeur, la fenêtre demandera la valeur voulue. À ce moment, la méthode «`LancerAcquisition` » demandera la valeur en donnant la lettre et le numéro du port. Pour la suite, cela se passe sur la méthode «`DonneMesure` » qui fera donc la réception puis le traitement de la valeur. Pour finir, la méthode renverra cette méthode vers **F_VisualiserLesValeurs** pour qu'elle soit affichée. Cette boucle se fait pour chaque valeur demander par une fenêtre.

2 - Réalisation de la tâche professionnelle : Communication avec le port série

Ma première tâche importante était de configurer et de rendre fonctionnelle l'intégralité de la liaison série avec le module de Qt « Qt Serial Port ». Cette tâche consiste donc à ouvrir et fermer le port, écrire et lire sur le port enfin de vérifier la disponibilité du port avant une écriture ou une lecture.

Le module Qt Serial Port est un module qui permet la gestion des ports séries qu'ils soient physiques ou virtuels, et ce, très facilement. Ce module contient deux classes `QSerialPort` et `QSerialPortInfo`. Celle que je vais utiliser est `QSerialPort`.

2.1 - Conception détaillée

Pour cette partie de projet, j'ai donc créé une classe que nous avons nommée « **Arduino** ». Cette classe sera donc la classe qui va gérer la communication avec la carte Arduino, l'objet qui utilisera cette classe sera « **Serie** »

2.1.1 - Diagramme de classe

Pour cette partie je vais me concentrer sur la classe « Arduino » ;

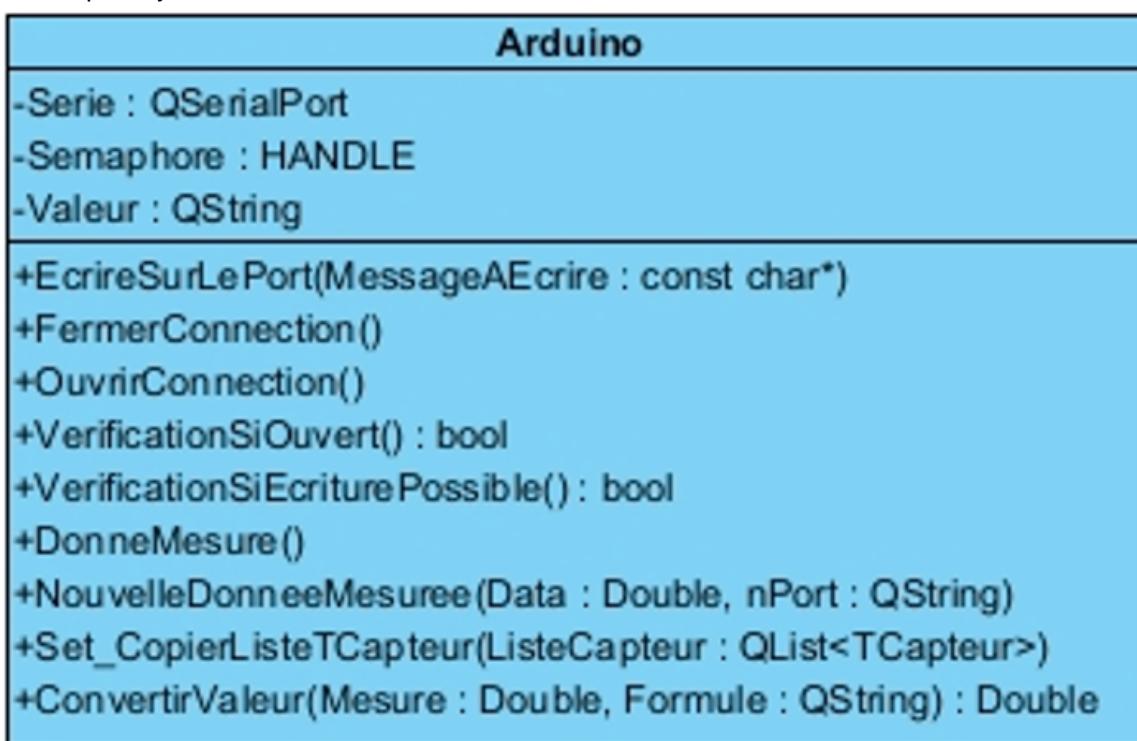


Figure 11: Diagramme de classe : Classe Arduino

Pour commencer, je vais présenter les **attributs** importants pour l'acquisition d'une valeur :

- **Valeur : QString** : Attribut contenant la chaîne de caractères dans laquelle il y a la valeur mesurée avec le « VALUE= » (page 9), il y a ensuite, un traitement de la chaîne pour n'avoir qu'exclusivement la valeur mesurée.
- **Serie:QSerialPort** : Objet représentant la liaison et permettra de configurer la liaison série avec la carte Arduino.

Ensuite, , les **méthodes** importantes sont :

- **Le constructeur par défaut** : Ce constructeur permet la configuration de la communication comme ci-dessous

```
//PARAMETRAGE DE LA LIAISON SERIE
Serie.setPortName      ("COM3");
Serie.setBaudRate      (QSerialPort::Baud19200);
Serie.setDataBits      (QSerialPort::Data8);
Serie.setParity         (QSerialPort::NoParity);
Serie.setStopBits       (QSerialPort::OneStop);
Serie.setFlowControl   (QSerialPort::NoFlowControl);
```

- Le nom du port sert à savoir par quel port USB du PC la connexion filaire, donc la communication, sera faite. Ici, le port COM3 est le port par défaut.
- La vitesse de communication consiste à savoir combien de changement de tension sera faite en 1 seconde. Par défaut, cette valeur est de 9600 bauds, cette valeur est modifiable dans le gestionnaire des périphériques section « Ports (Com et LPT) »
- La parité sert à contrôler si les flux envoyés sont corrects.
- Bit de stop pour savoir quand est-ce que le message est terminé.
- Un contrôle de flux que nous ne faisons pas.

- **OuvrirConnection():void** : Sers à ouvrir la connexion en lecture/écriture et de vérifier si le port est libre.
- **Ecrire(MessageAEcritre : const char *):void** : Sers à écrire sur le port série la chaîne de caractère en paramètre.

2.1.2 - Communication avec la carte

Les deux données traitées se situent dans les méthodes :

- **Ecrire(MessageAEcritre : const char *):void**: Entre la valeur, vers la carte Arduino, du port et du type de port. Le message doit être conforme aux exigences de l'interpréteur de commande.

La suite du code vient du signal de la librairie de QSerialPort. Ce signal représente le moment où de nouvelles données sont disponibles. Dès que ce moment arrive, la méthode **DonneMesure** sera appelée. Cet appel se fait à l'aide d'un connect. Un connect est un connecteur d'un signal et d'un slot. Le principe de ce connecteur est que dès qu'un signal est reçu (ici **ReadyRead**) la méthode contenue dans le slot est exécutée (ici **DonneMesure()**).

```
connect (&Serie, SIGNAL(readyRead()),  
        this, SLOT(DonneMesure()) );
```

- **DonneMesure():void** : Lis la valeur de retour du port série, la traite puis l'envoie vers la fenêtre d'affichage des valeurs.

Cette méthode consiste en premier lieu à lire la valeur mesurée brute avec la fonction **ReadAll()** qui retourne dans un tableau d'octet les données lues. Ensuite, , elle supprime le « VALUE= » du tableau. Ensuite, , il y a deux choix possibles. Soit la valeur est renvoyée sans autre traitement soit elle a besoin d'être convertis à l'aide d'une formule mathématique. Pour cela, je vais utiliser un parser qui fera cette tache. Le fonctionnement détaillé du parser sera expliqué plus bas.

2.1.3 - Diagramme de séquence

Ce qui concerne l'acquisition des valeurs c'est ce qui est encadré en rouge, c'est-à-dire la communication série. Ce qui n'est pas dans la cadré dans la partie 3.

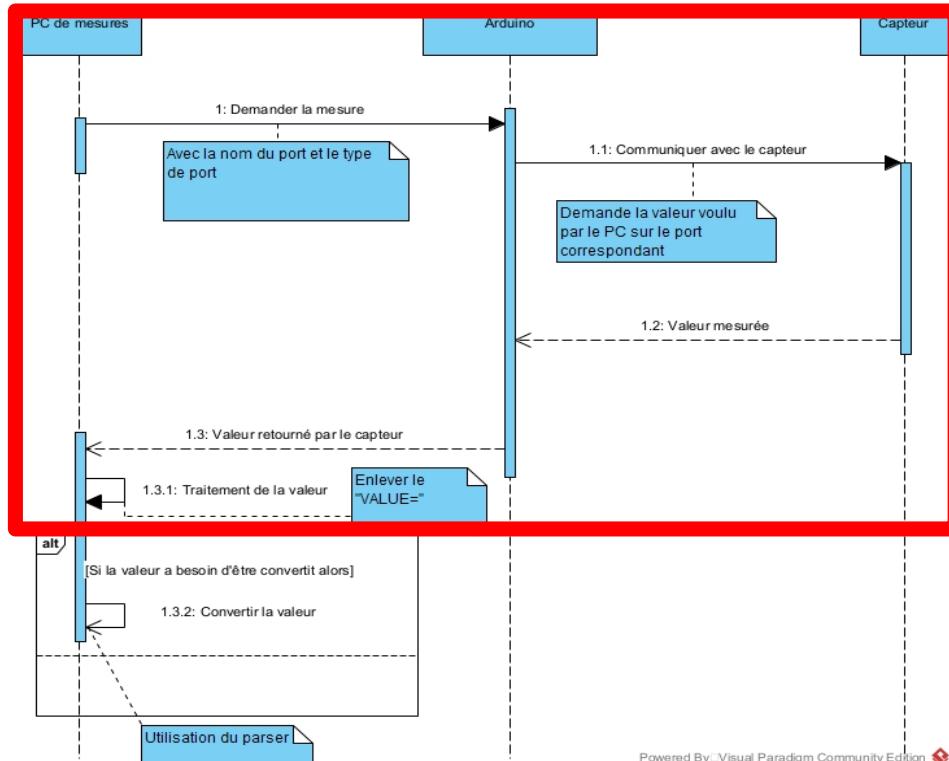


Figure 12: Diagramme de séquence : Liaison série

2.2 - Tests unitaires : Acquisition d'une valeur

2.2.1 - Identification du test unitaire

Acquisition Valeur : U1

2.2.2 - Objectif du test

Le test est de prouver l'efficacité de la classe Arduino en montrant qu'elle arrive bien à retourner la valeur mesurée par un capteur.

Pour faire ce test, je vais utiliser un capteur de température (voir ci-dessous). C'est un capteur analogique qui sera branché sur le port 00 sur les ports analogique de la carte. La commande pour pouvoir avoir la valeur du potentiomètre sur 8 bits est donc : A00.

Ce capteur compatible avec le shield Grove délivre une tension entre 0 et 5 V, en courant continu, en fonction de la température mesurée. Il est fonctionnel pour des valeurs allant de -40 à +125°C avec une précision de 1,5°.

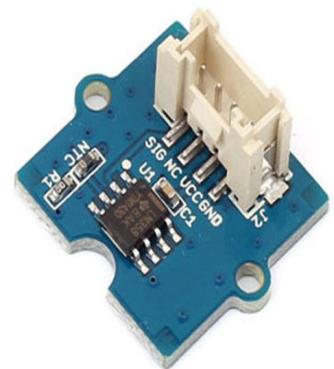


Figure 13: Capteur de température du Shield Grove

2.2.3 - Procédure de test

Id	Méthode testée Description Sommaire	Procédure de test												
		Résultats attendus												
U1.1	Préparer l'expérience	Brancher le potentiomètre sur la prise analogique 0 Capteur correctement branché												
U1.2	Compiler le programme « U1 – Liaison Serie »	Make Le programme compile sans erreurs												
U1.3	Constructeur par défaut : Arduino::Arduino(QObject *parent)	Configure la communication série La liaison est opérationnelle <table border="1"><tr><td>Nom Du port</td><td>COM3</td></tr><tr><td>Vitesse de communication</td><td>19200</td></tr><tr><td>Bits de données</td><td>8</td></tr><tr><td>Contrôle de Parité</td><td>Non</td></tr><tr><td>Bit de stop</td><td>1</td></tr><tr><td>Contrôle de Flux</td><td>Non</td></tr></table>	Nom Du port	COM3	Vitesse de communication	19200	Bits de données	8	Contrôle de Parité	Non	Bit de stop	1	Contrôle de Flux	Non
Nom Du port	COM3													
Vitesse de communication	19200													
Bits de données	8													
Contrôle de Parité	Non													
Bit de stop	1													
Contrôle de Flux	Non													
U1.4	Classe Arduino : DonneMesure():void Lis et traite la valeur retournée par la carte	La méthode reçoit la valeur demandée Trame reçue : « VALUE=119 » Suite au traitement de la trame : « 119 »												
U1.5	Vérification du changement des valeurs	Tourner le potentiomètre La valeur affichée se modifie												

2.2.4 - Rapport d'exécution

Id	OK	KO	Rapport d'exécution
U1.1	*		Capteur branché sur la prise 00
U1.2	*		Pas d'avertissement de compilation
U1.3	*		Message dans qDebug() : Port disponible
U1.4	*		Dans qDebug() : Vision de la trame non traitée puis traitée
U1.5	*		La valeur affichée change

2.3 - Problèmes rencontrés

La première chose sur laquelle j'ai buté est l'utilisation de l'interpréteur de commande, avec en premier l'utilisation de l'IDE d'Arduino « Arduino IDE » qui permet de compiler des programmes dans la carte Arduino. Cette première étape m'a permis de me familiariser avec l'interpréteur de commande, et de comprendre mon premier problème qui est la vitesse de communication avec la carte. Par défaut, la vitesse de communication se fait à 9600 bauds sauf qu'à cette vitesse les deux entités (PC et carte) ne pouvaient pas communiquer. J'ai donc cherché la vitesse correcte qui était finalement 19200 bauds.

L'autre problème était la maîtrise de la librairie QSerialPort enfin de configurer correctement la liaison. Également de recevoir la donnée mesure, néanmoins un problème persiste. Chaque valeur lue est doublée et donc lors de l'affichage, il est affiché à ce moment deux valeurs. Le plus souvent, cette valeur est un 0, il me reste à résoudre ce problème.

3 - Réalisation de la tache professionnelle : Conversion d'une valeur

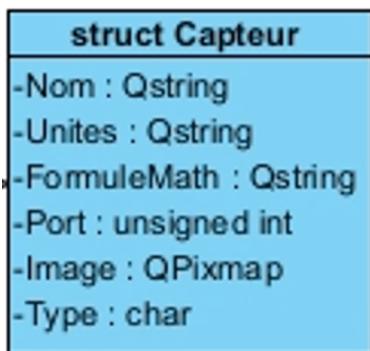
3.1 - Conception détaillée

3.1.1 - Présentation du parser

Si nécessaire un capteur peut être associé à une formule mathématique pour qu'une conversion avant l'affichage puisse être faite. Cette formule se situe dans la structure « Capteur » qui contient la formule sous forme d'une chaîne de caractère, chaîne qui est en l'état impossible à utiliser pour faire un calcul. C'est à ce moment où intervient le parser. Il va transformer la chaîne de caractère en véritable formule et enfin faire le calcul.

Pour le choix du parser j'ai choisi le parser : MuParserX. J'ai opté pour ce parser pour sa puissance, ses nombreuses fonctionnalités ainsi avec son aide, il est possible de traiter de formules mathématiques différentes. Comme la fonction exponentielle, les chiffres imaginaires ou les calculs unaires (avec un opérande)

3.1.2 - Diagramme de classe

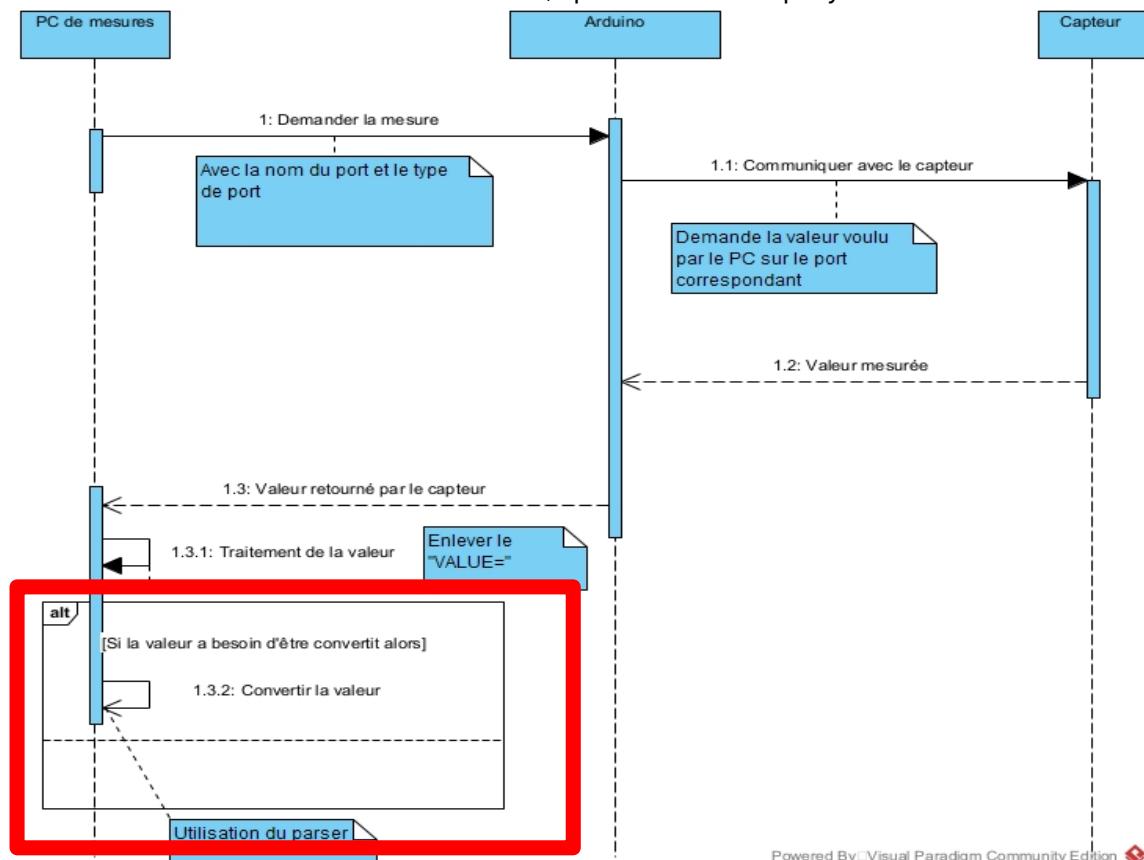


Structure dans laquelle toutes les informations utiles pour chaque capteur utilisé sera contenu. Le champ qui m'intéresse ici est le champ « FormuleMath ». C'est là où, comme expliqué plus tôt, sera contenu la formule mathématique, si elle existe.

Figure 14: Diagramme de classe :
Structure Capteur

3.1.3 - Diagramme de séquence

Cette conversion sera fait dans la classe Arduino, après avoir vérifié qu'il y a un besoin d'un convertissement.



Powered By Visual Paradigm Community Edition

Figure 15: Diagramme de séquence : Parser Mathématiques

3.1.4 - Codage du parser

Comme expliqué précédemment, je vais utiliser un parser mathématique pour convertir une chaîne de caractère en une formule mathématique et de pouvoir y introduire des variables. Ici il n'y aura qu'une variable cela sera la valeur mesurée par le capteur.

Prenons en exemple un capteur analogique qui a pour convertir sa valeur mesure la formule : **cos(a)+50**. Ici la variable sera le « a » et sera remplacé par le parser avec la valeur mesurée. La présentation suivante sera faite dans l'ordre chronologique d'utilisation.

3.1.4.1 - Configuration du parser

La première chose à faire à une installation du paquet correct. Dans ce contexte, un paquet indique quelles fonctions vont être utilisable par le parser. Pour le projet, j'ai choisi le paquet « **pckALL_NON_COMPLEX** ». Ce paquet représente toutes les fonctions non-complexes mis en place par le parser.

```
ParserX PaquetParser (pckALL_NON_COMPLEX);
```

J'ai décidé de prendre ce paquet plutôt qu'un autre parce que nous n'allons pas utiliser de fonctions complexes pour la conversion d'une valeur, j'ai donc pas installé une notion qui n'allait pas être utilisé. Mais il est toujours possible de rajouter ces fonctions.

3.1.4.2 - Préparation des éléments de conversion

Dans la suite des codes sources, il y a deux paramètres qui sont utilisés lors de l'appel de la méthode :

- **Mesure** : Contient la valeur mesurée par le capteur après le traitement de chaîne
- **FormuleMesure** : Contiens la formule du capteur qui était préalablement contenu dans la structure Capteur du capteur associé, dans notre exemple cela sera **cos(a)+50**.

Ensuite, , il faut définir 3 choses pour pouvoir utiliser le parser, la variable de la formule et le formule.

- Définition de la variable de la formule, ici la valeur mesurée. En premier la création de la variable de type **Value** qui contient la valeur mesurée puis une chaîne qui contient le même caractère de la précédente variable.

```
Value    a (Mesure);  
QString ValeurMesure ("a");
```

- Ensuite, je définis cette variable en rapprochant ces deux variables. En résumé, je dit que chaque « **a** » qu'il y a dans la formule doit être remplacé par la valeur mesurée.

```
PaquetParser.DefineVar(ValeurMesure.toStdWString(), Variable(&a));
```

- La définition de la formule mathématique. C'est à ce moment que je transforme la chaîne de caractère en véritable formule mathématique. La fonction **SetExpr** n'accepte que des **string_type** je fais également une conversion vers ce type de données.

```
PaquetParser.SetExpr(StringFormuleMathematique.toStdWString());
```

- Le calcul avec la formule et la variable

```
ValeurConverti = PaquetParser.Eval();
```

À la fin c'est la variable **ValeurConverti** qui a la valeur finale, celle qui sera affichée et visible par l'utilisateur.

3.2 - Test unitaire : Convertir une valeur

3.2.1 - Identification du test unitaire

Conversion d'une valeur : U2

3.2.2 - Objectif du test

La valeur rentrée par un capteur de température à besoin d'une conversion pour que sa valeur soit compréhensible pour l'utilisateur. Cette formule est trouvée dans la documentation du capteur, après simplification, elle donne :

- $1.0 / (\log(255/a) - 1.0) / 4275 + 1/298.15 - 273.15$

Ici, la variable « **a** » représente la valeur mesurée.

Exemple :

- Valeur rentrée par le thermomètre : 112 → Valeur rentrée par la méthode : 20($^{\circ}$ C)

3.2.3 - Procédure de test

Id	Méthode testée Description Sommaire	Procédure de test
		Résultats attendus
U2.1	Connaître la température ambiante	Mesurer la température avec un thermomètre
		Température Ambiante
U2.2	Remplacer la valeur de la variable Mesure par la valeur trouvée lors de U1	Changer la valeur de la variable
		Pas d'erreurs
U2.3	Compiler le programme « U2 – Parser »	Make
		Le programme compile sans erreurs
U2.4	Affichage de la valeur convertie	Une valeur s'affiche sur la terminal
		La valeur correspond au thermomètre
U2.5	Affichage de la valeur convertie	Faire le calcul avec la calculatrice
		La valeur trouvée à la calculatrice est la même que celle trouvé par le parser

3.2.4 - Rapport d'exécution

Id	OK	KO	Rapport d'exécution
U2.1	*		Valeur mesurée par le thermomètre
U2.2	*		Aucunes erreurs n'apparaissent
U2.3	*		Pas d'avertissement de compilation
U2.4	*		Affichage de la valeur convertie
U2.5	*		Valeur convertie égale à la température ambiante

3.3 - Problèmes rencontrés

L'utilisation du parser m'a posé des problèmes sur deux points. D'abord l'installation. Comme le parser utilise des méthodes et des types de variable qui lui sont propre et externe à Qt j'ai dû créer et implémenter la bibliothèque muparserX, qui contient tous les fichiers d'en-tête avec les fichiers CPP associés pour pouvoir utiliser la bibliothèque.

Le deuxième point est simplement l'utilisation du parser et plus précisément la gestion des types et leurs cohabitations avec les types internes et déjà existant de Qt.

4 - Réalisation de la tâche professionnelle : Création du sémaphore

Le problème qui est soulevé par cette tâche est le fait que lorsqu'on utilise une liaison série, la carte ne peut être retournée la valeur d'un capteur qu'un par un. La cause est la liaison série qui ne peut communiquer qu'une trame à la fois. Pour gérer plusieurs lectures de capteurs en même temps, nous allons utiliser un sémaphore.

4.1 - Conception détaillée

4.1.1 - Présentation du sémaphore

Le sémaphore est une structure de données, qui est associée à un compteur d'utilisation, qui est destiné à protéger une ressource qui ne peut être utilisée que par une tâche à la fois. Ici, cette tâche sera la communication liaison série. Le principe d'un sémaphore est la décrémentation et l'incrémentation du compteur à chaque utilisation et arrêt d'utilisation. Chaque utilisation décrémente le compteur et chaque fin d'utilisation récrémente ce compteur. La particularité de ce système est lorsque ce compteur tombe à zéro toutes les autres tâches sont mis en pause jusqu'à ce qu'une d'elle libère une unité du compteur. Le sémaphore apparaît à deux endroits d'un code. Lors de la décrémentation (prendre) puis de l'incrémentation (vente). La snippet qui est entre ces deux étapes est appelé la section critique.

Dans le cas du projet, le sémaphore sera utilisé pour la lecture de la valeur retournée par la carte Arduino, le sémaphore permettra de modérer et limiter l'utilisation de la ressource qui est la liaison série.

4.1.2 - Diagramme de séquence

L'utilisation du sémaphore se déroule de cette façon :

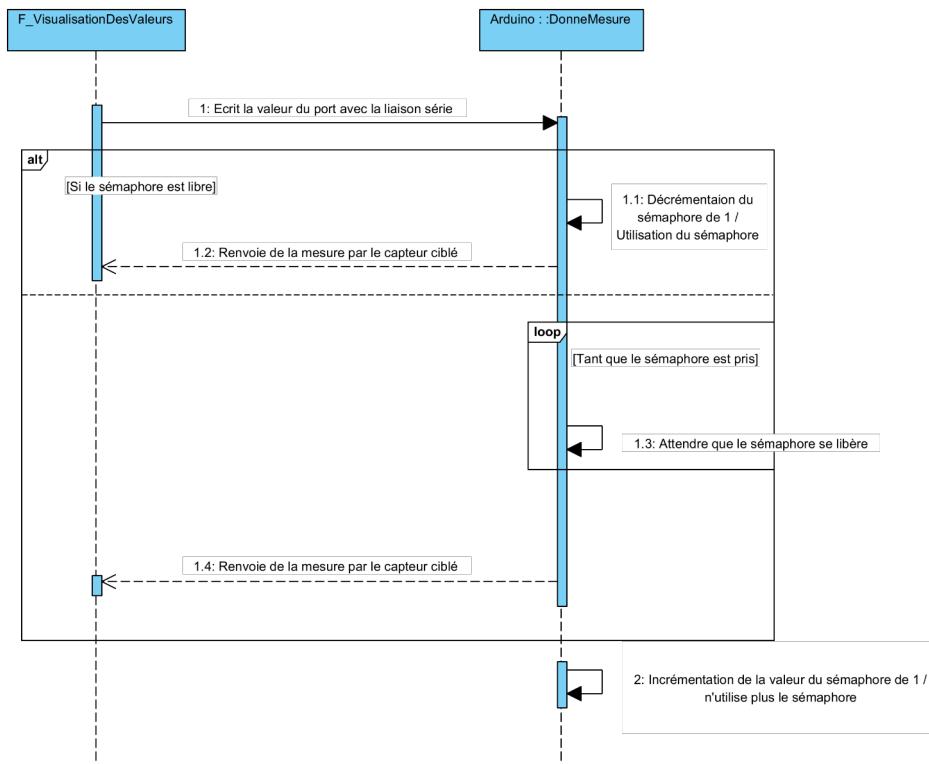


Figure 16: Diagramme de séquence : Utilisation du sémaphore

L'objectif du sémaphore est donc de limiter l'utilisation du port série par le PC. Le sémaphore sera donc utilisé lors de lecture de la valeur retournée par la carte dans la méthode `DonneMesure` de la classe `Arduino`. Si le sémaphore est libre, la lecture se fera sinon la lecture attendra que le sémaphore se libère.

4.1.3 - Codage du sémaphore

Le sémaphore se code de la façon suivante :

- Création du sémaphore
- Initialisation de la prendre : **WaitForSingleObject**
- Initialisation de vente : **ReleaseSemaphore**

Sachant que « prendre » et « vendre » encadreront la section critique.

4.1.3.1 - Constructeur : Crédit du sémaphore

Dans le constructeur de la classe **Arduino** :

```
hSemaphore = CreateSemaphore
            (nullptr,           // Sécurité de l'héritage
             1,                 // Valeur initial du compteur
             1,                 // Valeur maximal du compteur
             (LPCTSTR)"PortSerie"); // Nom du sémaphore
```

4.1.3.2 - Balisage de la section critique

```
void Arduino::DonneMesure()
{
    for (int i = 0 ; i < ListeCapteur.size(); i++)
    {
        if (ListeCapteur[i].Port == Port)
        {
            Formule = ListeCapteur[i].FormuleMath;
        }
    }

    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    WaitForSingleObject(hSemaphore, INFINITE);
    TableauARetournee = Serie.readAll();
    ReleaseSemaphore(hSemaphore, 1,nullptr);

    ValeurAAfficher.append(TableauARetournee);

    Valeur.append(TableauARetournee);

    qDebug() << "Valeur reçue" << Valeur << '\n';

    [...]
    ValeurMesureTraite = oConversion.ConvertirValeur(ValeurMesureTraite, Formule);
    //Renvoie La valeur au connect situé dans F_VisualierLesValeurs
    emit (NouvelleDonneeMesuree (this->sPort, ValeurMesureTraite));
}
```

Ici, les deux utilisations du sémaphore se font sur les lignes soulignées en jaune. La première est le moment où le sémaphore est décrémenté d'un et atteint 0 (Voir nombre maximal 4.1.3.1), ce qui signifie que lorsqu'une nouvelle donnée sera disponible cette donnée attendra que le sémaphore soit à nouveau libre pour pouvoir exécuter la section critique. La deuxième partie est donc le relâchement du sémaphore par la méthode où elle rend le sémaphore qu'elle utilisait pour la prochaine exécution de **DonneMesure**.

4.2 - Problème rencontrés

Le problème que j'ai encore actuellement est l'utilisation du sémaphore et bien savoir s'il est utilisé et d'une façon correcte qui permet bien de protéger la section critique.

5 - Bilan de la réalisation personnelle

Pour rappel mes tâches à faire durant ce projet était donc de recueillir la valeur puis le renvoie de celle-ci via le port série, cette tâche était accompagné de mise en place du parser mathématique. Ensuite, je le suis concentré sur la mise en place du parser et la dernier tâche est l'exportation des valeurs mesurées grâce au format CSV.

Les points validés sont donc :

- **Visualisation des valeurs mesurées**

Cette étape a été le premier et la plus importante de mes tâches. C'est grâce à elle que les fenêtres sont capables de communiquer avec la carte de recevoir les valeurs mesurées. Il reste encore un problème à résoudre, c'est la tête que plusieurs renvoient de valeur sont faite en même temps, ce qui gène à la compréhension des valeurs.

- **Utilisation du parser**

Cette étape obligatoire pour la visualisation des valeurs pour certains capteurs a été la partie la plus difficile de mon projet à cause de mon inexpérience avec le sujet.

- **Mise en place du sémaphore**

Le sémaphore qui sert à protéger la liaison série de plusieurs utilisations en même temps à également été mis en place. Le problème que j'ai encore avec cette partie du projet est que j'ai du mal à prouver qu'il est bien fonctionnel.

Ce qui reste encore à mettre en place :

- Résolution des problèmes cités pour la visualisation des valeurs et le sémaphore
- **Création de l'exportation des valeurs mesurée et d'un IHM.**

À la fin d'une expérience, l'utilisateur doit avoir le choix d'exporter les valeurs mesurées vers un tableau à l'aide du format de fichier CSV.

Fonction Développé	Description	Avancement
FP2 : Visualisation des valeurs mesurées	Communication avec la carte Arduino pour avoir la valeur mesurée. Utilisation d'un parser mathématiques si besoins	Terminé
Création et mise en place d'un sémaphore	Développement d'un sémaphore afin de pouvoir gérer plusieurs capteurs en fonction des paramétrages de l'expérience	En cours, bientôt terminée
FS3 : Exporter les valeurs mesurées	En fin d'expérience, l'utilisateur doit avoir la possibilité d'exporter les résultats de tous les capteurs dans un tableau.	Non commencée