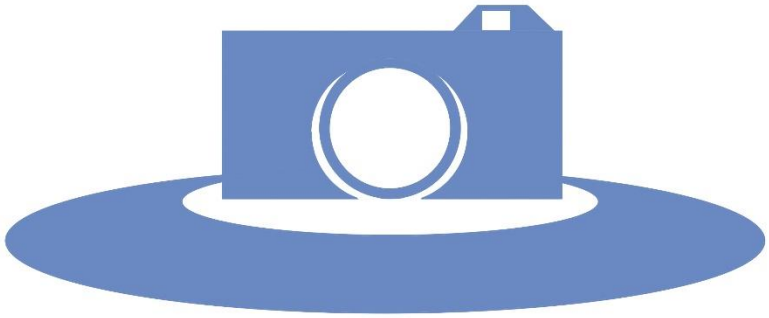


Studio Photo Reconnaissance

Projet BTS SN

Lycée Alphonse Benoit Isle sur la sorgue 2021

Revue finale

Adam VALENZA Gabriel BUFFARD Mathias DEMETRESCO Tony ELECTEUR

Table des matières

Présentation générale du système

Le but du projet est de réaliser un studio de photo **autonome** permettant de détecter des changements potentiels sur des emballages de produits cosmétiques. Ce système nous a été soumis par l'entreprise **CrossDock**.

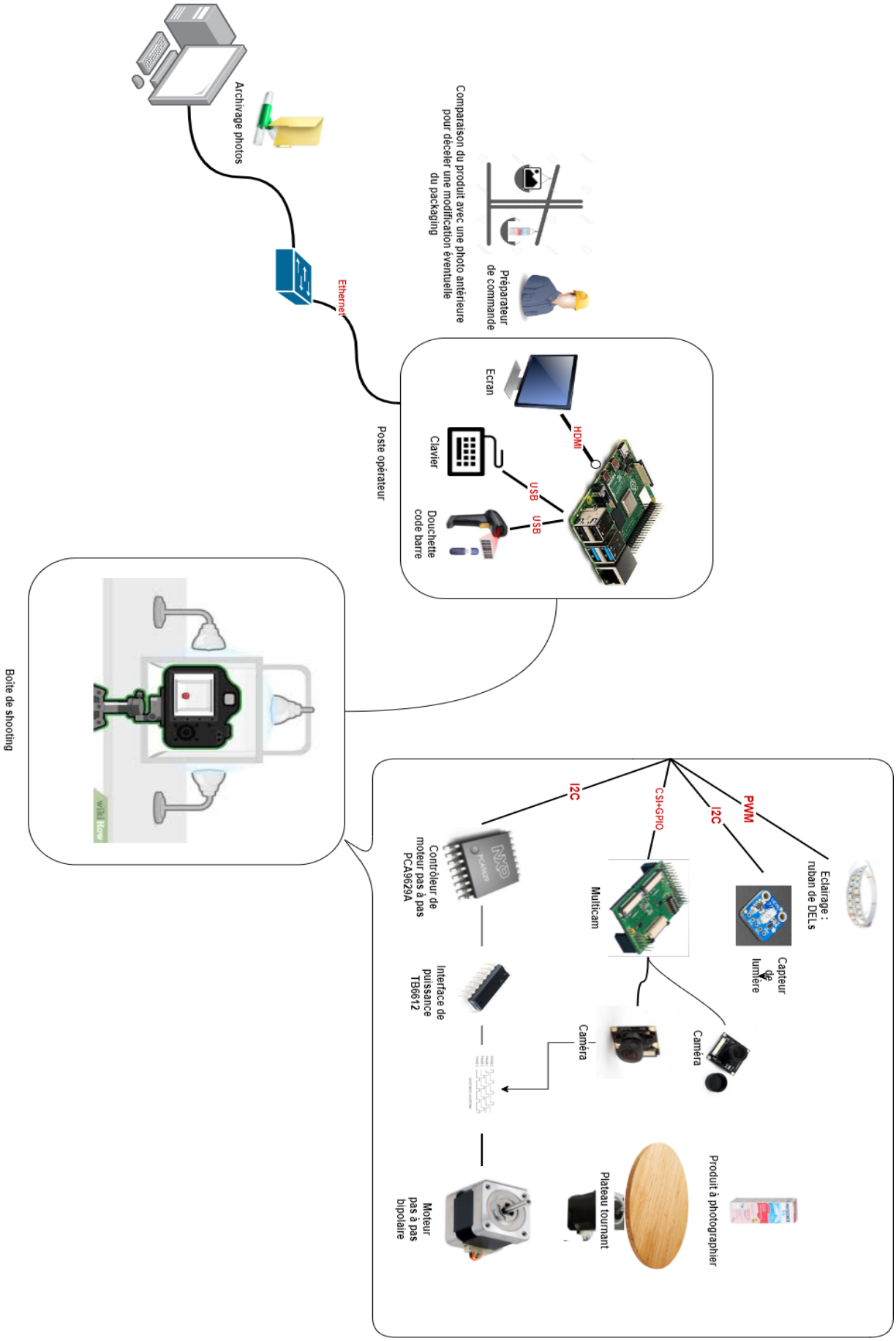
CrossDock est une entreprise de logistique dont l'activité consiste à préparer puis à expédier des commandes constituées de produits cosmétiques et/ou de parapharmacie.

Par expérience, il se sont rendu compte qu'il arrive parfois que pour un même code commande, leurs fournisseurs changent la forme de l'emballage, sa taille, ses couleurs, ces changements pouvant parfois s'accompagner d'une modification de la composition du produit en lui-même.
CrossDock souhaite donc pouvoir en **informer ses clients**.

Le projet consiste donc à fabriquer un studio de photo **piloté** par un **poste opérateur** via une **IHM** permettant de prendre en photos les quatre faces d'un packaging automatiquement pour être capable de constater d'éventuelles différences par rapport à des clichés pris antérieurement.

Le **poste opérateur** est articulé autour d'une **Raspberry pi 3** qui pilote **deux caméras**, un **plateau tournant** via un **moteur pas à pas** tout en assurant la **gestion de l'éclairage** à partir d'un **ruban LED** dont l'intensité lumineuse peut varier en fonction d'un **capteur de luminosité** ambiante. L'ensemble du système tient dans une enceinte qui constitue une boite de shooting.

À noter que le système est destiné à évoluer vers une fonctionnalité permettant la comparaison automatique des différentes images. Cette dernière fonctionnalité n'est pas à notre charge.
Voici un aperçu de la structure du système :


Archivage photos
Ethernet
Comparaison du produit avec une photo antérieure pour déceler une modification éventuelle du packaging
Préparateur de commande
Ecran
HDMI
USB
USB
Clavier
Douchette code barre
Poste opérateur
Boite de shooting
Eclairage : ruban de DELs
PWM
I2C
Capteur de lumière
CSI+GPIO
Multicam
Caméra
Caméra
I2C
Contrôleur de moteur pas à pas PCA9629A
Interface de puissance TB6612
Produit à photographier
Plateau tournant
Moteur pas à pas bipolaire

Fonctionnement du système

L'opérateur doit tout d'abord **scanner** l'article qu'il souhaite comparer à l'aide d'un lecteur de code-barres.

L'article est ensuite positionné sur le plateau tournant.

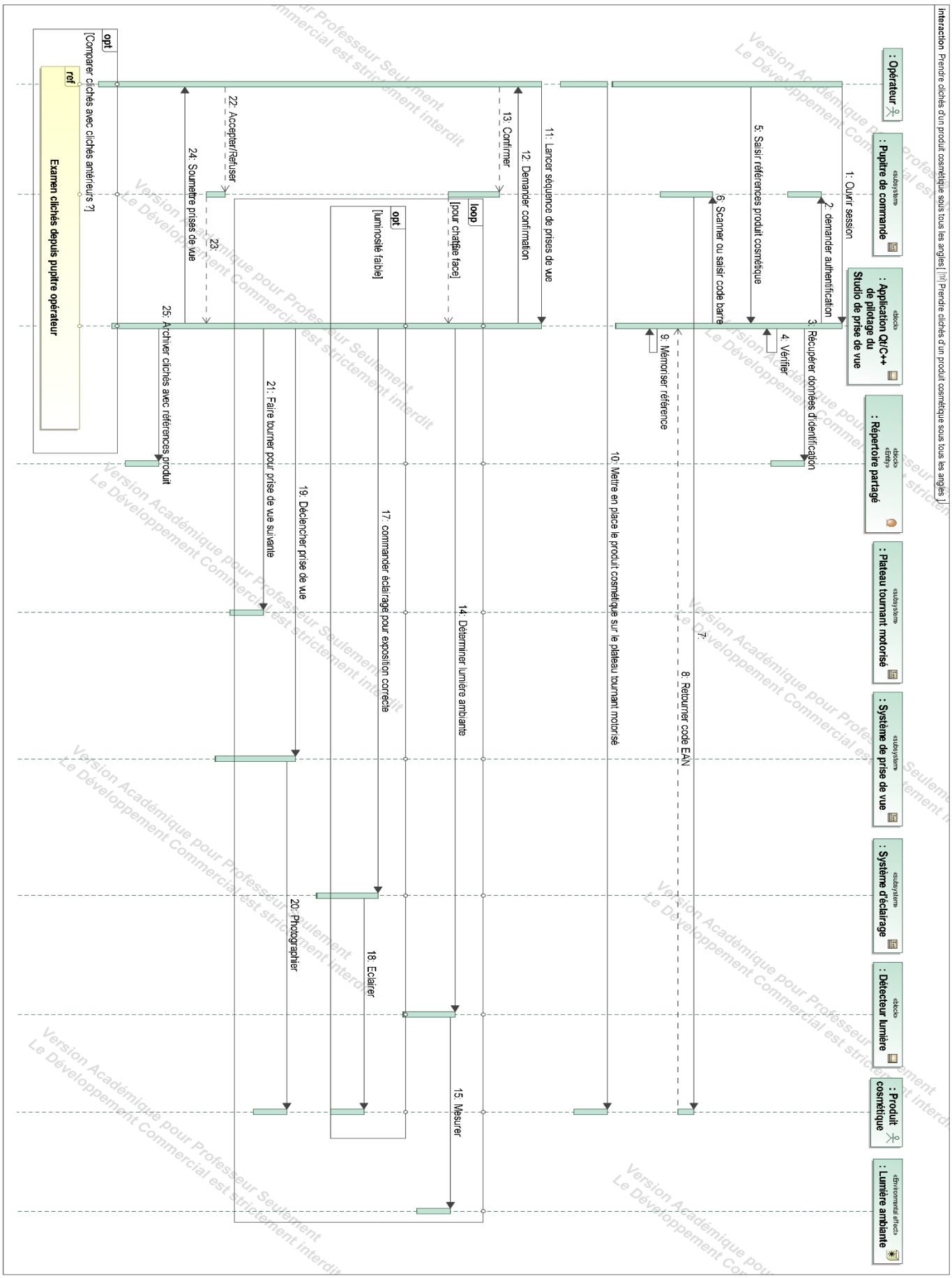
L'opérateur peut ensuite **adapter au besoin**, la luminosité dont le réglage peut également se faire **automatiquement**. Une fois le processus enclenché, une photo de chaque face est réalisée en effectuant une rotation d'un quart de tour du moteur, entre chaque prise de vues.

Suite à cela, les photos pourront être **archivées** sur un post Windows suite à une confirmation de l'opérateur. Si les photos ne sont pas satisfaisantes, le processus peut être renouvelé.

Suite à l'archivage des prises de vues, on peut alors faire une **comparaison** de ces dernières avec les versions précédentes si elles existent, **identifiées par le code-barres** du produit en question.

À chaque différence trouvée, l'opérateur doit **saisir une note** qui sera enregistrée avec les dernières images capturées.

Voici le **diagramme de séquence** pour résumer cela **:**



Cahier des charges

L'entreprise CrossDock n'a pas directement fournie de cahier des charges, ce dernier a été établi au fil de plusieurs discussions entre le lycée et CrossDock.

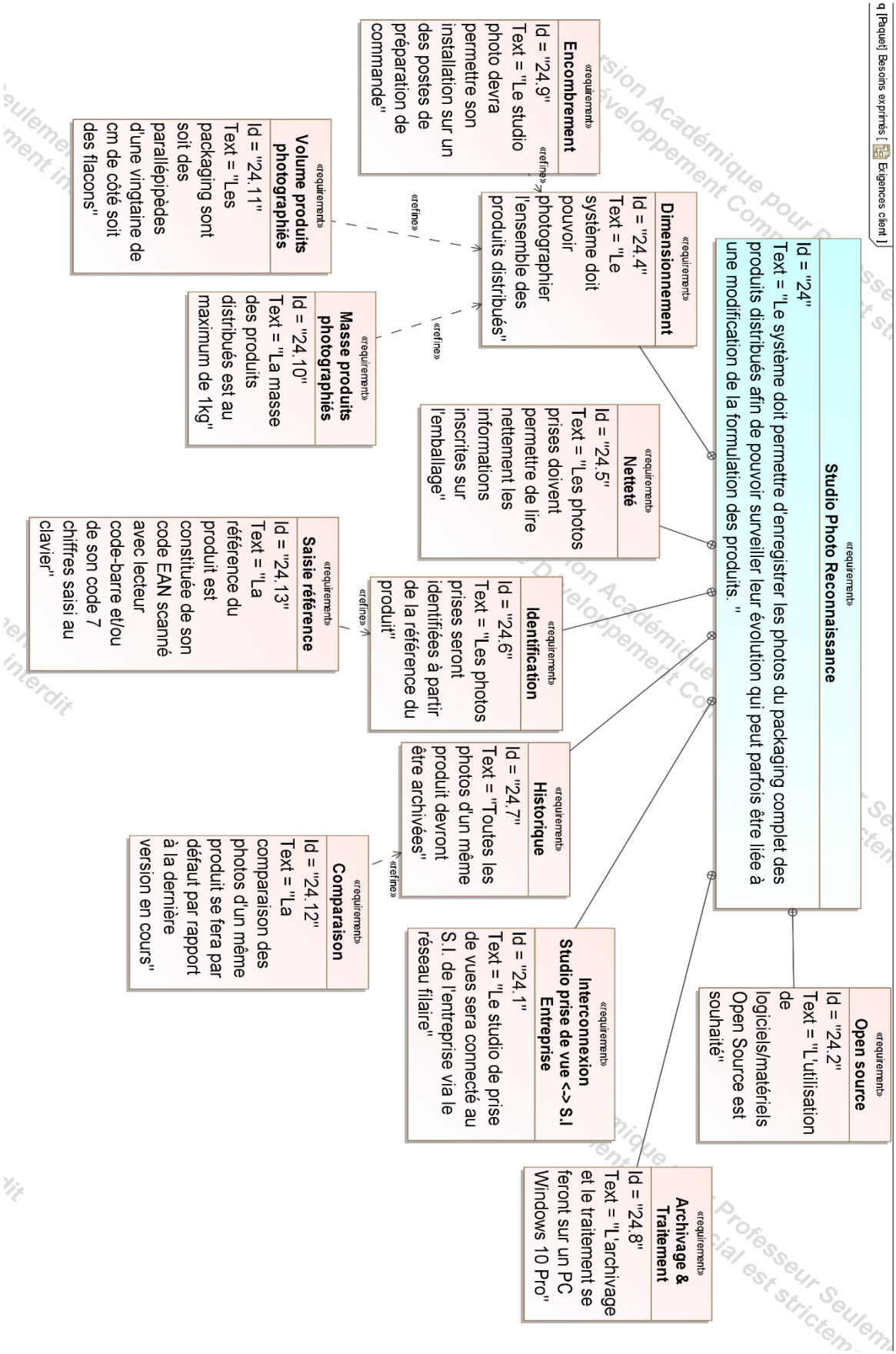
Voici les éléments que nous devons **prendre en considération** :

- Les produits sont dans des **emballages en carton parallélépipédiques** d'une **vingtaine de centimètres** de côté maximum, ou dans des flacons.
- La masse maximale des produits est de **1Kg**
- La netteté des prises de vues doit **permettre la lecture** des informations présentes sur l'emballage.
- Un scanner de code-barres (douchette) doit permettre **l'identification d'un produit**.
- L'encombrement du système doit **tenir sur le poste de travail** des préparateur·rice·s, ou le cas échéant, sur un poste dédié.
- Toutes les photos d'un même produit doivent être **archivées**, la **comparaison** se faisant, par défaut, par rapport à la **dernière version** en cours.
- **Les prises de vues, la commande du plateau tournant et la gestion de l'éclairage se font à partir d'un Raspberry Pi**, l'entreprise ayant une certaine expérience dans ce type de nano-ordinateur.
- **L'archivage** des photos se fait sur un **PC Windows10 Pro**.

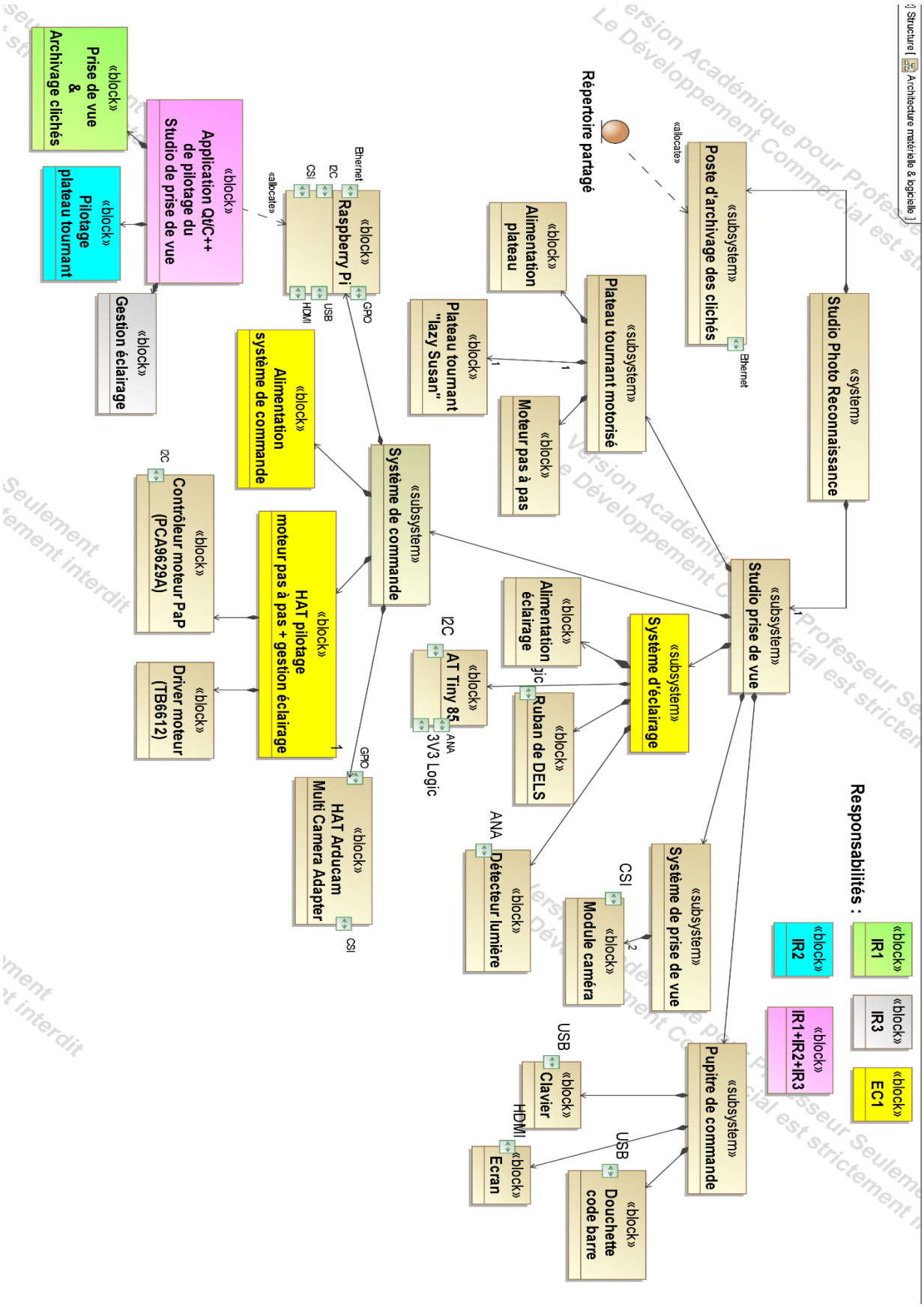
Nous nous sommes également imposé les **contraintes** suivantes :

- Le système doit être **simple** d'accessibilité et facilement rangeable.
- L'IHM doit être la plus **simple et intuitive** possible.
- La prise de vue peut se faire via **deux caméras au choix** (l'une ou l'autre mais pas les deux en même temps) car l'une a une **meilleure qualité d'image mais avec un champ de vision trop réduit** pour les grands objets, et l'autre a une **qualité un peu amoindrie mais pour un meilleur champ de vision** pour les objets encombrants.

Voici le **diagramme des exigences** résumant le cahier des charges ci-dessus :



Pour finir, voici le **diagramme de block** du système **:**



Planification

Répartition des Tâches			
VALENZA Adam Étudiant IR1	BUFFARD Gabriel Étudiant IR2	DEMETRESCO Mathias Étudiant IR3	ÉLECTEUR TONY Étudiant EC1
➢ Concevoir l'IHM ➢ Mettre en œuvre et piloter les caméras ➢ Mettre en œuvre le scanner de code-barres ➢ Concevoir les classes C++ pour la caméra et scanner de code bar. ➢ Concevoir le système d'archivage des prises de vues. ➢ Intégrer le développement au système final	➢ Concevoir l'IHM ➢ Mettre en œuvre et piloter le plateau tournant ➢ Concevoir les classes C++ permettant la communication I2C et les commandes du plateau tournant ➢ Intégrer le développement au système final	➢ Concevoir l'IHM ➢ Concevoir le protocole de communication avec EC1 pour le pilotage de l'éclairage ➢ Mettre en œuvre et piloter l'éclairage ➢ Récupérer les données du capteur de luminosité ➢ Concevoir la classe C++ de l'éclairage. ➢ Intégrer le développement au système final	➢ Mettre en œuvre la carte de commande du plateau tournant ➢ Prévoir des capteurs de fin de course pour le plateau tournant en cas de dysfonctionnement. ➢ Mettre en œuvre le microcontrôleur : « ATtiny 85 » permettant le pilotage des LED ➢ Effectuer la saisie du schéma et le routage de ce Hat. ➢ Produire les fichiers Gerber afin que la fabrication du PCB soit sous-traitée. ➢ Câbler le PCB de la carte et effectuer les essais. ➢ Documenter la mise en service de la carte finale.

Environnement de travail

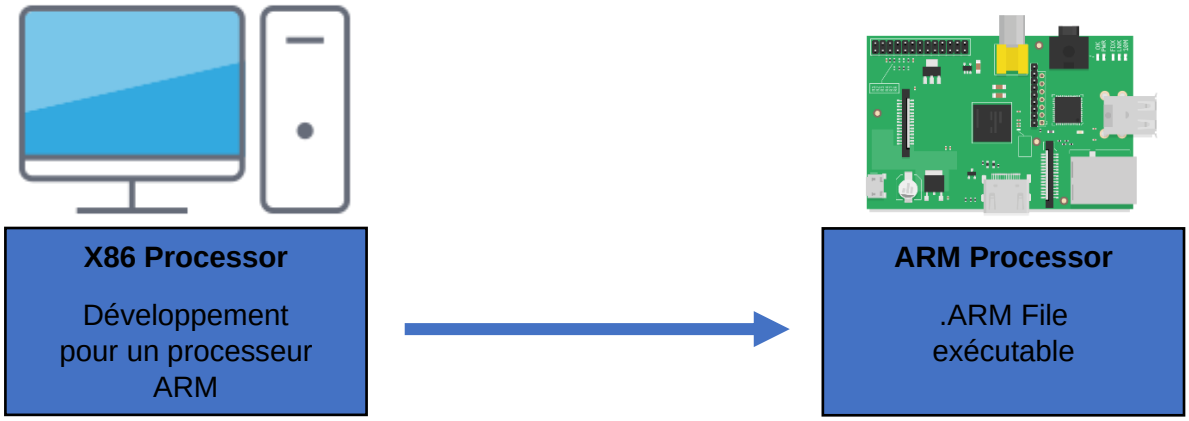
Méthode de développement

Au sein de l'équipe du projet, nous sommes tous équipés d'une **Raspberry pi 3**.

Cette dernière n'est pas très adaptée pour faire du développement de par sa **faible puissance**.

Les étudiants IR ont donc optés pour la méthode de la « **cross compilation** ».

La **cross compilation** (compilation croisée) est la possibilité sur une machine avec un matériel spécifique (architecture) et avec un système d'exploitation donné, de compiler des programmes pour une autre architecture, ou pour un autre système d'exploitation :



Le **principal avantage** est que l'on peut développer depuis un **système ayant assez de ressources** pour pouvoir travailler confortablement, et pouvoir par la suite, envoyer le programme compilé sur le système final.

Pour cela, nous avons installé sur les Raspberry, une version de **Raspberry pi OS** (sans interface graphique) déjà **préconfigurée** pour la **cross-compilation Qt** depuis un **environnement Linux** s'exécutant sur le PC de développement.

Seule la configuration réseau restait à faire :

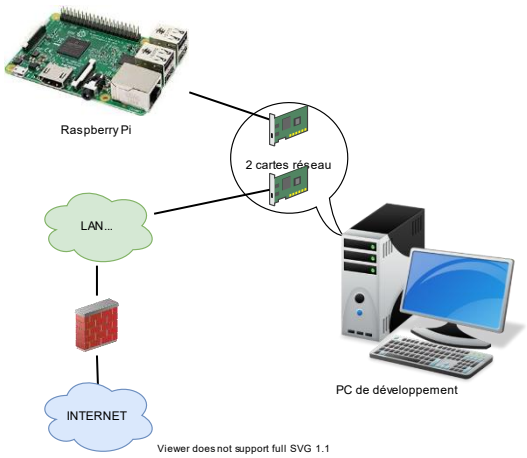
- Attribuer une IP fixe :
  Il existe plusieurs méthodes pour attribuer une IP fixe sous Raspberry pi OS. Mais celle qui est recommandée consiste à modifier le fichier /etc/dhcpcd.conf. Pour cela, il faut décommenter la section suivante et l'adapter à notre convenance :
  #Example static IP configuration :
  #interface eth0
  #static ip_address=192.168.0.10/24
  #static ip6_address=fd51:42f8:caae:d92e::ff/64
  #static routers=192.168.0.1
  #static domain_name_servers=192.168.0.1 8.8.8.8 fd51:42f8:caae:d92e::1

  Exemple de configuration :

```
interface eth0
static ip_address=172.16.18.15/16
static routers=172.168.16.5 #IP PC Carte réseau 2
static domain_name_servers=192.168.1.254 8.8.8.8
```

- Puis redémarrer.

Quant aux PC, on les a équipés d'une **seconde carte réseaux** pour pouvoir connecter nos Raspberry directement dessus :

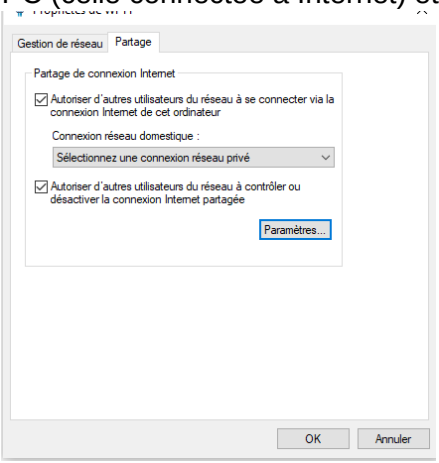


Cette méthode offre **deux avantages principaux** :

- **Pas besoin** d'utiliser **de switch ou de prises réseaux murales** supplémentaires qui peuvent **faire défaut** sur le plateau de développement.

- On **ne dépend pas de la bande passante du LAN** pour accéder à la Raspberry pi, ce qui permet d'être **plus rapide**.

Dans ce cas il faut configurer la carte réseau ajoutée pour qu'elle soit dans le **même réseau que la Raspberry**, et activer **un partage de connexion** entre la carte réseau principale du PC (celle connectée à Internet) et la seconde (celle connectée à la Raspberry).



Activation du partage de connexion depuis les paramètres Windows.

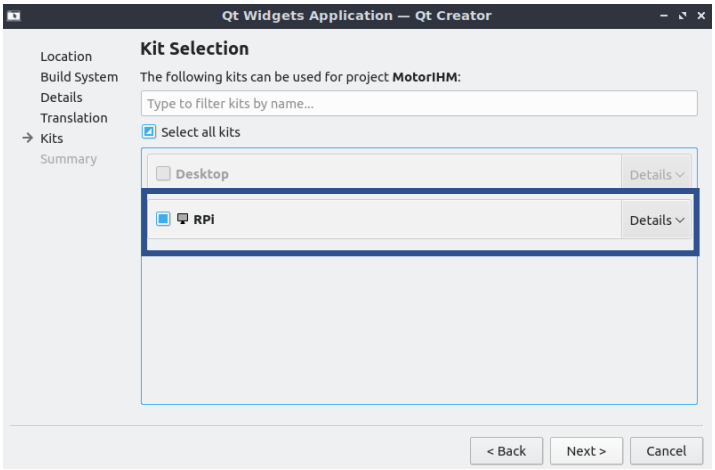
Pour le développement, nous utilisons une **machine virtuelle préconfigurée pour la cross-compilation** avec le **framework Qt**.

La VM est basée sur la distribution « **LUbuntu** » qui est **dérivée d'Ubuntu**, visant à être plus légère, plus économe en ressources matérielles et en énergie.

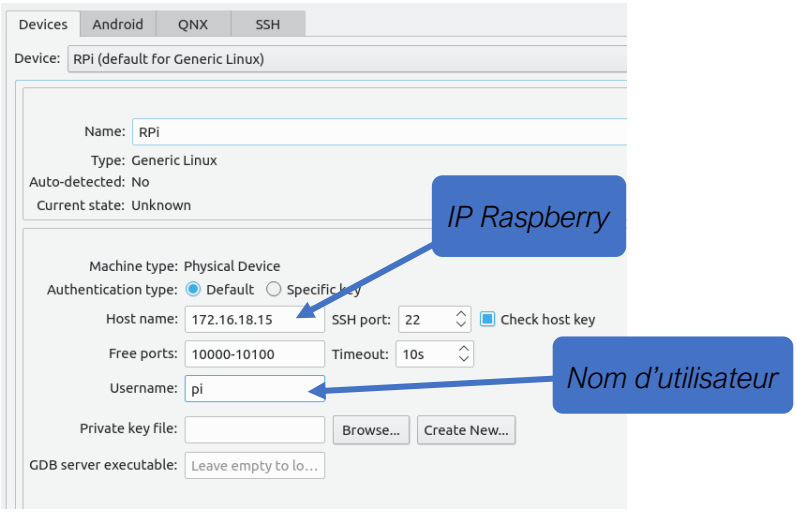
Là encore, seule la configuration réseau restait à adapter.

Nous développons donc, avec le **framework QT** sur **QT Creator**.

Un **kit de développement** pour RPI était déjà installé.



Il ne restait plus qu'à configurer QT pour qu'il puisse **déployer** directement sur la Raspberry lors de la compilation :



Ainsi, une fois le processus enclenché depuis QT, l'application se copie automatiquement sur la Raspberry, et s'exécute. L'IHM apparait, et ce, même si l'OS de la Raspberry n'a pas d'interface graphique.

L'étudiant EC quant à lui, utilise simplement une Raspberry avec **Raspberry PI OS Desktop** (version graphique) étant donné que le développement logiciel ne constitue pas la plus grande partie de son travail.

Gestionnaire de version

Il est important lorsque l'on fait du développement, d'avoir une **trace de son code** et accessoirement des documents annexes.

Pour gérer le **versionning** et pouvoir **collaborer** à plusieurs sur même projet, nous avons choisi d'utiliser **Git**.

Git est un logiciel de gestion de versions décentralisé. Il permet de faire des sauvegardes de documents / code en permettant d'ajouter des descriptions sur les modifications, et d'enregistrer les sauvegardes sur un dépôt distant.

Le dépôt distant est hébergé sur **Framagit** appartenant au réseau d'éducation **Framasoft** :

Git dispose aussi d'un **système de branches** permettant de diviser les différentes étapes de développement pour ne pas affecter ce qui est stable.

Par exemple, pour développer une fonctionnalité ou corriger un bug sur un programme, on peut créer une « **branche parallèle** » qui permettra de garder la version originale intacte tant que la fonctionnalité n'est pas au point.

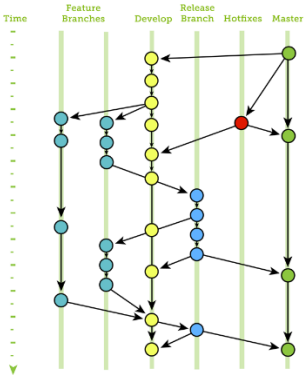


Illustration du système de branches, la branche principale étant la « Master »

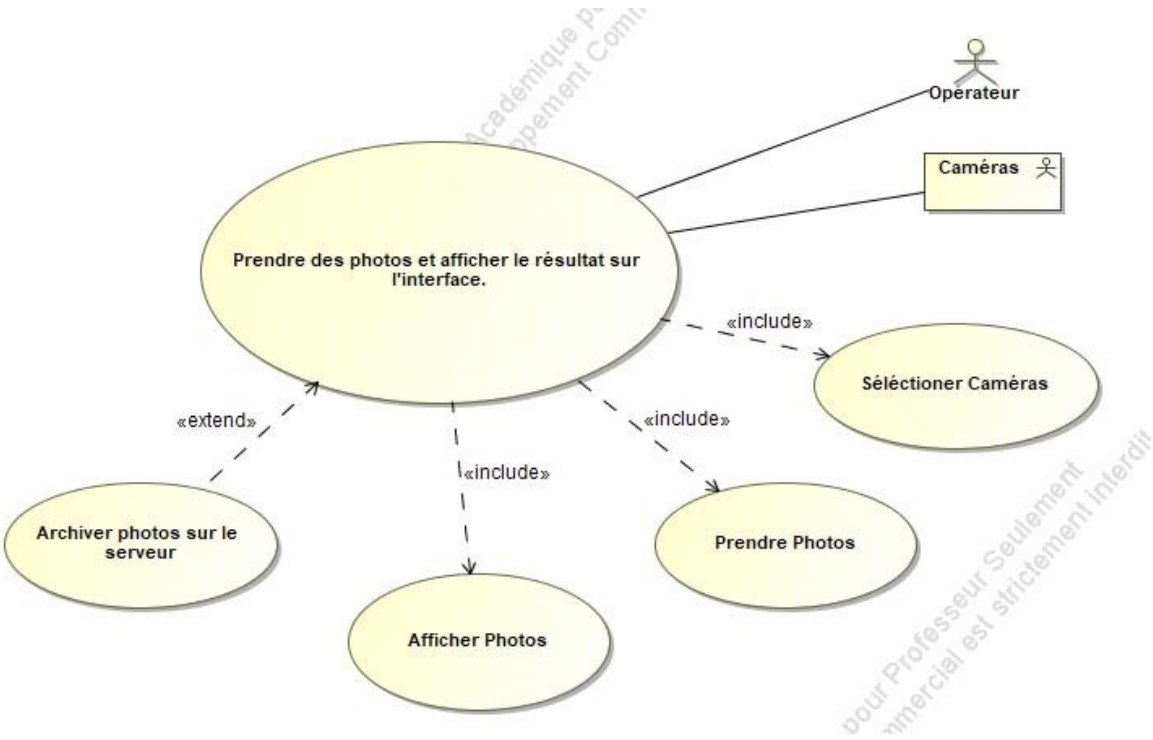
Partie Étudiant IR1 : VALENZA Adam

Objectifs

- Concevoir l'IHM.
- Mettre en œuvre et piloter les caméras.
- Concevoir les classes C++ permettant la communication GPIO et les commandes des caméras.
- Intégrer le développement au système final.

Le **choix** a été fait **d'utiliser deux caméras** : la première permettant de capturer une image d'objet de petite taille et la deuxième, ayant un angle de vue plus élargie, permet de capturer une image d'objet de plus grande taille.

Diagramme des cas d'utilisations



Prototypage : Mise en œuvre rapide

Matériel utilisé :

- Raspberry Pi 3b+ :

Le **Raspberry pi** est un nano ordinateur de la taille d'une carte de crédit que l'on peut brancher à un écran et utiliser comme un ordinateur standard. Sa petite taille, et son prix intéressant fait du **Raspberry pi** un produit idéal pour tester différentes choses, et notamment la création d'un serveur Web chez soi

- Multi Camera Board V2.2 UC-475 :

La carte UC-404 permet de connecter jusqu'à quatre caméras silmultanément sur la Raspberry.

- Caméra 8MP Sony IMX219 avec objectif CS 2718 :

Cette caméra est capable de générer des images statiques de 3280 x 2464 pixels et prend également en charge les vidéos 1080p30, 720p60 et 640x480p90. Il se connecte au Pi par l'interface CSi standard dédiée.

- Camera 5 mégapixel OV5647 sensor :

Cette caméra possède un champ de vision de 160° tandis que la plupart des caméras possèdent un champ de vision de 72°. Elle possède également une résolution de 1920x1080 pixels.

Étude des cameras

Lors de cette partie, nous allons procéder à l'analyse des caméras utilisées lors de la prise de photo. On rappelle que les caméras utilisées sont :

- Une caméra Sony IMX219 8MP

- Une caméra RB-CameraWW 5MP

Ces deux caméras utilisent toutes deux un capteur CMOS, et sont connectées à la raspberry via des câbles « CSI » (Camera Serial Interface).

Capteur CMOS

Nous allons dans un premier temps nous concentrer sur le capteur CMOS. Pour rappel, dans une caméra ou un appareil photo, un capteur est un composant électronique photosensible, c'est-à-dire qu'il réagit à la lumière. Celui-ci convertit les rayons lumineux en un signal électrique analogique. C'est ce signal qui, une fois amplifié, permet d'obtenir une image numérique. Le capteur CMOS quant à lui transforme pour chaque pixel les charges électriques générées par la lumière en tension électrique.

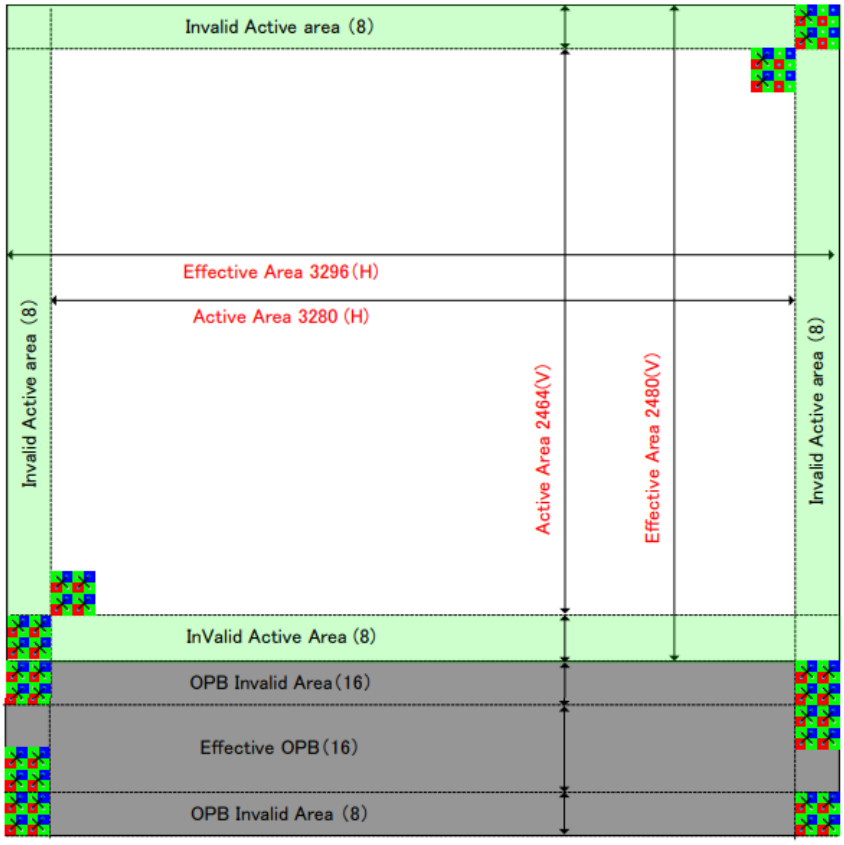
Caméra Sony IMX219 8MP

Nous allons maintenant voir la caméra Sony IMX219 8MP. Comme dit précédemment, cette caméra possède un capteur CMOS, celui-ci étant composé de photodiodes de 1.12 µm x 1.12 µm en silicium.

Elle possède un nombre total de pixels de 3296 pixels en largeur pour 2512 pixels en hauteur soit approximativement 8,28 Million de pixels. Les pixels totaux d'une caméra incluent les pixels de l'image en elle-même, les pixels permettant de définir la couleur de chaque pixel et les pixels OB (pixels permettant de déterminer le niveau de gris sur une image).

Le nombre de pixels effectifs de cette caméra est de 3296 x 2480 pixels soit environs 8,17 Million de pixels. Les pixels effectifs correspondent aux pixels utilisés dans l'image, incluant les pixels permettant de déterminer la couleur de chaque pixel.

Le nombre de pixels actifs est de 3280 x 2464 pixels soit à peut près 8,08 Million de pixels. Les pixels actifs correspondent aux pixels servant à définir l'image.



Matrice de pixel d'une image physique

Pour la capture vidéo, la caméra Sony IMX219 8MP possède plusieurs résolutions :

- Une première d'une résolution d'écran de **1920x1080 pixels** pour **30 images par secondes**.
- Une deuxième d'une résolution de **1280 x 720 pixels** pour **60 images par secondes**.
- Une troisième de **640 x 480 pixels** pour **90 images par secondes**.

Cette caméra possède également un champ de vision de **75°**.

La **carte SD** sur la Raspberry Pi doit pouvoir **mémoriser** les images provenant de la caméra.

<u>Calcul du nombre maximal d'images que peut contenir la carte :</u>

Poids d'une image : nb de pixels par ligne x nb de pixels par colonnes

P = **3296*2512 =** $\mathbf{8,3.10^6}$ pixels

Sachant qu'un pixel est codé sur **24 bits** on multiplie le résultat par 3

Taille img = $\mathbf{8,3.10^6}$ **x 3 = 24,9.** $\mathbf{10^6}$

Afin d'acquérir la taille d'une image en format JPEG, on divise la taille de notre image par 20

Img Jpg = **1245000 = 1,245 Mo**

Sachant que nous possédons une carte SD de **16Go**, le nombre d'image pouvant être stockées serait de : **16Go/1.245Mo** = **12851 images**.

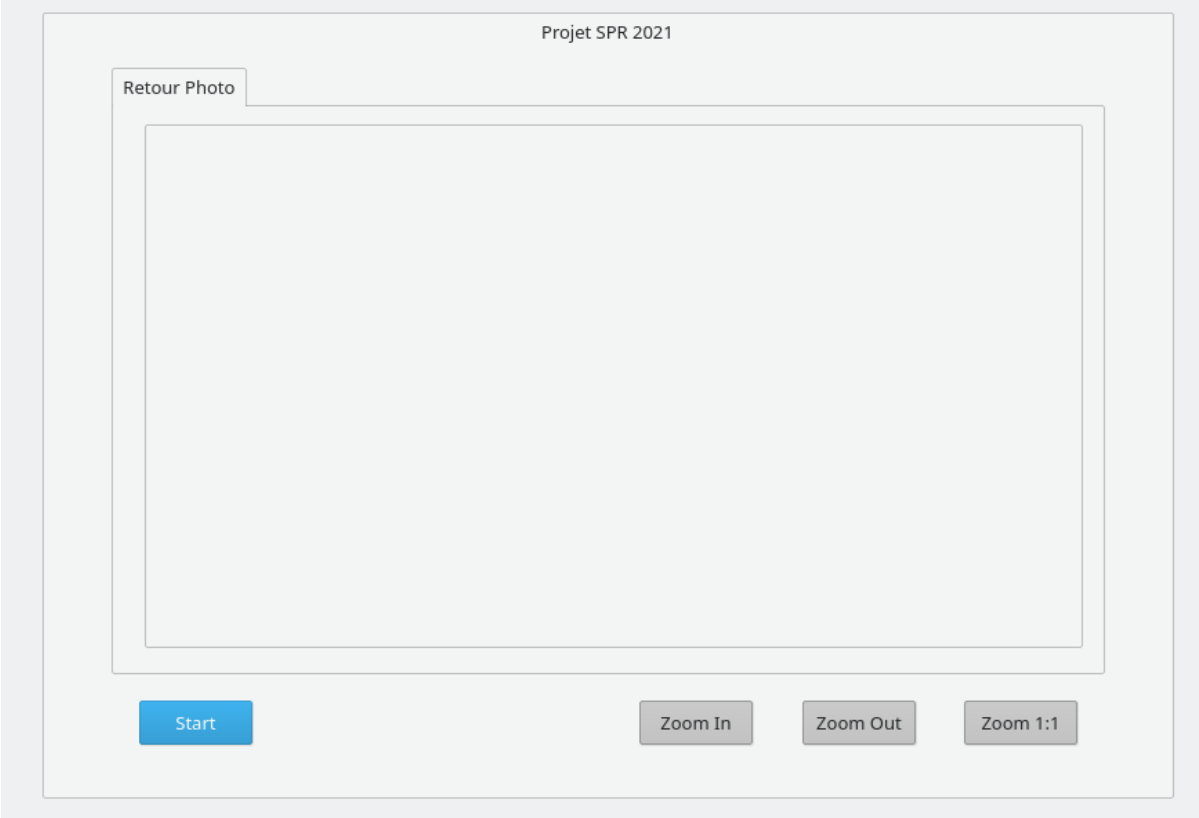
Sans prendre en compte le système d'exploitation et les fichiers déjà présents sur la Raspberry, on pourrait stocker environ **12850 images**.

Prototypage : Conception d'une IHM en C++

L'objectif ici est de pourvoir **prendre une photo depuis une application** Qt et **d'afficher le résultat** sur l'interface de cette même application.

Nous allons donc travailler sur **QT Creator**.

L'interface sur laquelle nous allons travailler est celle-ci :



Réalisation de la classe PHOTO :

Pour cela nous allons coder une classe permettant de lancer le processus de capture d'image, elle nous permet également d'afficher le résultat de la capture et de pouvoir zoomer et dézoomer sur l'image.

Voici son diagramme de classe :

Diagramme de la classe « PHOTO »

Fichier photo.h :

```
#ifndef PHOTO_H
#define PHOTO_H

#include <QMainWindow>
#include <QDialog>
#include <QScrollArea>
#include <QProcess>

QT_BEGIN_NAMESPACE
namespace Ui { class PHOTO; }
QT_END_NAMESPACE

class PHOTO : public QMainWindow
{
    Q_OBJECT

public:
    PHOTO(QWidget *parent = nullptr);
    ~PHOTO();

private slots:
    void on_pbStartProcess_clicked();
    void on_pbBack_clicked();
    void on_pbNext_clicked();
    void on_pbReset_clicked();
    void onRaspistillProcessFinished(int exitCode, QProcess::ExitStatus exitStatus);

private:
    Ui::PHOTO *ui;
```

```
    bool setImage(QString imgPath);
    void scaleImage(double factor);
    QImage m_image;
    QScrollArea *m_scrollArea;
    double m_scaleFactor;
    QProcess * m_raspistillProcess;

    const QString IMG_PATH = "/var/tmp/img.jpg";
};
#endif // PHOTO_H
```

Le contrôle de la caméra est basé sur un **QProcess** permettant **d'exécuter des programmes**, cela peut être des **commandes Linux**.

Tout d'abord, voici l'implémentation du constructeur :

```
PHOTO::PHOTO(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::PHOTO)
{
    ui->setupUi(this);

    m_scrollArea = new QScrollArea(ui->groupBox_2);
    m_scrollArea->setBackgroundRole(QPalette::Dark);
    m_scrollArea->setWidget(ui->label);
    m_scrollArea->setVisible(false);

    m_raspistillProcess = new QProcess(); //Instanciation de la classe QProcess
}
```

On commence par initialiser l'interface graphique, puis on crée une **instance de la classe QProcess()**.

Cette instance est ensuite utilisée pour prendre des photos grâce à la raspberry via le slot **on_pbStartProcess_clicked()** dont voici l'implémentation :

```
void PHOTO::on_pbStartProcess_clicked()
{
    QString cmd = "raspistill";
    QStringList args;
    args.append("-n");
    args.append("-t");
    args.append("500");
    args.append("-md");
    args.append("6");
    args.append("-o");
    args.append("/var/tmp/img.jpg");

    m_raspistillProcess->start(cmd, args); //Execute la commande raspistill


    connect(m_raspistillProcess
            , QOverload<int, QProcess::ExitStatus>::of(&QProcess::finished)
            , this
            , &PHOTO::onRaspistillProcessFinished
            );

}
```

Ici, on utilise la méthode `m_raspistillProcess->start(cmd, args);` pour lancer la prise de vue. Elle prend en argument le **programme à exécuter** qui est la **commande raspistill** dans

notre cas. Cette dernière permet de piloter une caméra depuis la raspberry. Le deuxième argument qu'il faut fournir sont les paramètres que prend la commande (fournie en premier argument). Voici la liste des arguments que l'on fournit :

- **-n** : permet de **désactiver la prévisualisation** avant la prise de vue.
- **-t** : permet de **définir un delai** avant de prendre une photo. Par défaut, il y'a un délai de 3 seconds entre l'exécution de la **raspistill** et la prise de vue.
- **-md** : permet de **définir le mode de prise de vue**. Les modes correspondent à des **résolutions d'image prédéfinis**.
- **-o** : permet de définir le chemin et le nom sous lesquels vont s'enregistrer les images prises.

Ensuite, pour afficher l'image, on utilise la méthode **setImage()** :

```
bool PHOTO::setImage(QString imgPath)
{
    QImageReader reader(imgPath);

    reader.setAutoTransform(true);

    const QImage img = reader.read();
    if (img.isNull()) {
        QMessageBox::information(this, QGuiApplication::applicationDisplayName(),
                                 tr("impossible de charger %1: %2")
                                 .arg(QDir::toNativeSeparators(imgPath), reader.errorString
()));
        return false;
    }

    m_image = img;

    m_scaleFactor = 1.0;

    m_scrollArea->setVisible(true);

    ui->label->setPixmap(QPixmap::fromImage(m_image));

    return true;
}
```

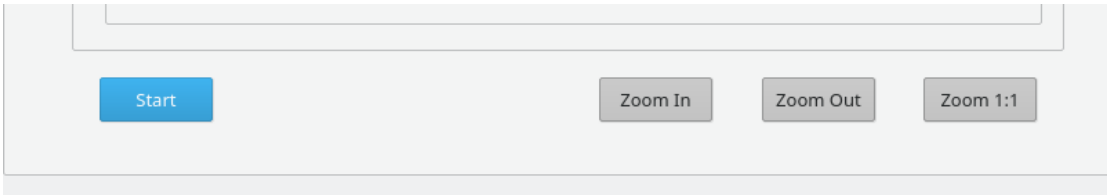
Cette méthode va donc charger l'image qui a été prise par la fonction précédente et l'afficher dans un **QLabel**.

Voici l'implémentation des autres méthodes :

```
void PHOTO::scaleImage(double factor)
{
    if(factor != 1) {
        m_scaleFactor *= factor;
        ui->label->resize(m_scaleFactor * ui->label->pixmap()->size());
    } else {
        ui->label->adjustSize();
        m_scaleFactor = 1;
    }

    QScrollBar * scrollBarH = m_scrollArea->horizontalScrollBar();
    QScrollBar * scrollBarV = m_scrollArea->verticalScrollBar();
```

```
    scrollBarH->setValue(int(factor * scrollBarH->value()
                            + ((factor - 1) * scrollBarH->pageStep()/2)));
    scrollBarV->setValue(int(factor * scrollBarV->value()
                            + ((factor - 1) * scrollBarV->pageStep()/2)));

    ui->pbBack->setEnabled(m_scaleFactor < 3.0);
    ui->pbNext->setEnabled(m_scaleFactor > 0.333);
}

void PHOTO::on_pbBack_clicked()
{
    scaleImage(1.25);
}

void PHOTO::on_pbNext_clicked()
{
    scaleImage(0.8);
}

void PHOTO::on_pbReset_clicked()
{
    scaleImage(1);
}

void PHOTO::onRaspistillProcessFinished(int exitCode, QProcess::ExitStatus exitStatus)
{
    setImage(IMG_PATH);
}
```

La dernière étape est donc d'appeler ces méthodes lorsque l'on appuis sur les différents boutons.



Ces boutons sont connectés a des slots qui ont directement été générés depuis Qt-Designer :

Ces **méthodes** (**slots**) sont donc **automatiquement appelées** lors de l'appuis sur les boutons correspondant à ces derniers.

Voici leurs implémentations :

```
void PHOTO::on_pbBack_clicked()
{
    scaleImage(1.25);
}

void PHOTO::on_pbNext_clicked()
{
    scaleImage(0.8);
}

void PHOTO::on_pbReset_clicked()
{
    scaleImage(1);
}
```

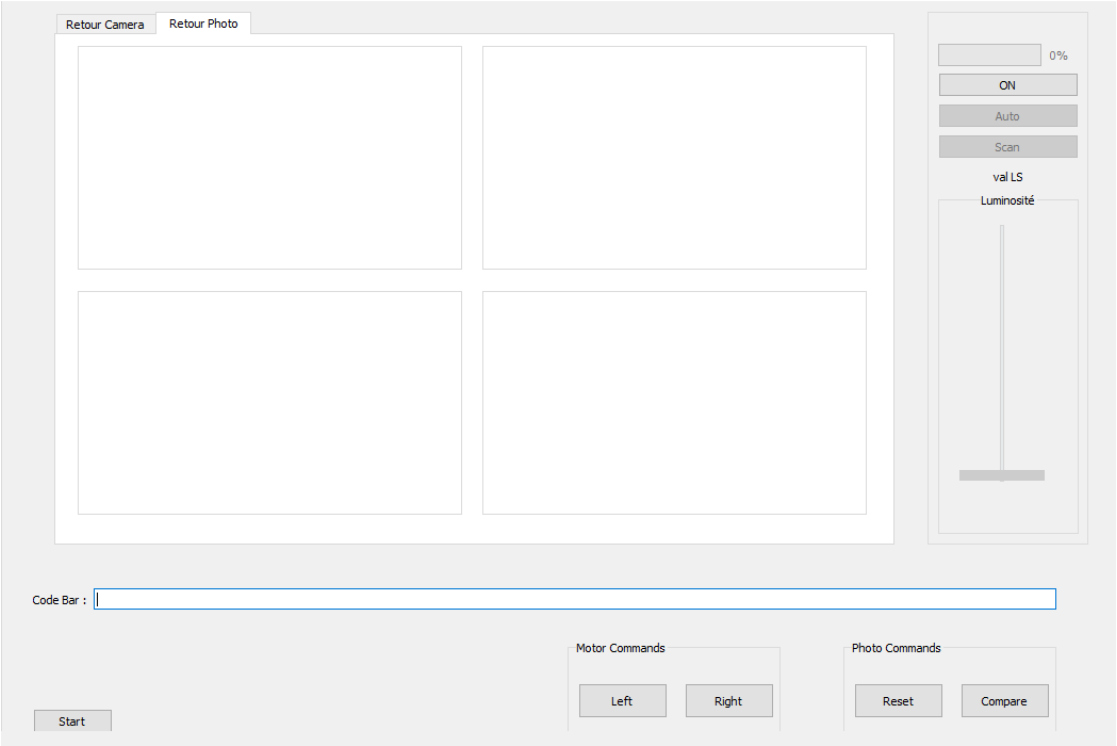
Nous avons maintenant un programme permettant de prendre une photo et d'afficher le rendue de celle-ci dans une interface QT.

Exemple de cliché photo lors des tests :

Développement : Conception de l'IHM :

Le but de cette partie et de pouvoir piloter le moteur sur l'application finale.

Voici le prototype de l'IHM qui a été réalisé par l'ensemble de l'équipe pour l'application finale :



L'IHM doit permettre :

- D'ajuster l'éclairage manuellement ou automatiquement.
- Déplacer le moteur manuellement ou automatiquement.
- Lancer le processus de prises de vues.
- Réinitialiser le processus de prises de vues, si les clichés ne conviennent pas.
- Avoir un retour direct de la caméra pour pouvoir effectuer les différents réglages nécessaires.
- Afficher les 4 prises de vues effectuées.
- Lancer le processus de comparaison (Fenêtre pop-up permettant d'afficher les images prises avec leurs versions antérieures, et saisir une note en cas de différences constatées).
- Saisir et afficher le code-barre d'un produit.

Développement : Intégration de la partie prise de vue :

Lors de cette partie, nous allons voir la mise à niveau du programme afin que celui-ci puisse prendre plusieurs photos à la suite, puis les afficher l'IHM.

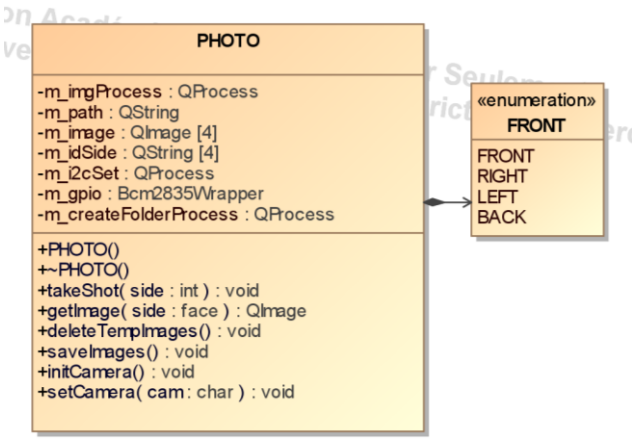
Dans un premier temps, il a fallu faire en sorte que le programme puisse prendre plusieurs photos l'une après l'autre à intervalle régulier tout en assurant la synchronisation des prises de vues avec la rotation du plateau tournant.

Adaptation de la classe PHOTO :

Pour cela, la classe précédente a été modifiée avec les éléments suivants :

Certaines des méthodes précédentes comme **on_pbStartProcess_clicked()**, **on_pbNext_clicked()** … ont été modifier / supprimées.

Voici le nouveau diagramme de classe :



Fichier photo.h :

```
#ifndef PHOTO_H
#define PHOTO_H

#include <QObject>
#include <QProcess>
#include <QMainWindow>

class PHOTO
{
public:
    typedef enum {
        FRONT,
        RIGHT,
        LEFT,
        BACK
    } face;

    PHOTO();
    ~PHOTO();

    void takeShot(int side);
    QImage getImage(face side);
```

```
    void deleteTempImage();
    void saveImages();

private:

    QProcess *m_imgProcess; //Processus permettant de prendre les photos
    const QString m_path = "/var/tmp/img"; //chemain de stoquage temporaire
    QString m_idSide[4] = {"F", "D", "A", "G"}; //indentifiants des differrentes faces

    QImage m_image[4]; //stockage des images pour archivages

};

#endif // PHOTO_H
```

Cette nouvelle classe doit donc permettre de :

- Prendre des photos
- Supprimer les photos qui sont stocker temporairement (avant l'étape d'archivage définit si les photos conviennent à l'opérateur du système).
- Archiver les photos sur un serveur Windows conformément au cahier des charges.

Étant donné que l'on a quatre images à prendre correspondantes aux quatre faces d'un produit, il a été choisi de créer une énumération qui sera utile pour identifier ces différentes faces :

```
public:
    typedef enum {
        FRONT,
        RIGHT,
        LEFT,
        BACK
    } face;
```

Voici l'implémentation de la nouvelle méthode **takeShot()** permettant de réaliser la prise de vue :

```
void PHOTO::takeShot(int side)
{
    QString cmd = "raspistill";
    m_imgProcess->setProgram(cmd);

    QStringList args;
    args.append("-n");
    args.append("t");  //Commande Raspistill avec ses paramètre
    args.append("1");
    args.append("-md");
    args.append("1");
    args.append("-o");
    args.append(m_path + m_idSide[side] + ".jpg");  //Chemin d'accès et nom de l'imageS

    m_imgProcess->setArguments(args);
    m_imgProcess->startDetached(cmd, args); //Execution de la commande raspistill
    qDebug("Je prend une photo !");
    usleep(800000); //Laisse le temps de prendre la photo avant de redémarrer le moteur
}
```

Par rapport à la version de la classe précédente, la modification majeur ici est l'utilisation de la méthode **sratDetached()** là où était utilisé avant, la méthode **start()** de la classe QProcess.

Ce changement se justifie par le fait qu'ici on doit prendre quatre photos à la suite, et donc lancer 4 fois le QProcess. Le problème est que la méthode start lance le même QProcess avec le même PID ce qui provoque une erreur car lorsque l'on veut enchainer avec les photos suivantes, une erreur survient car le QProcess précèdent est déjà en cours d'exécution.

La méthode **startDetached()** permet donc de résoudre ce problème en exécutant plusieurs fois le même processus de manière dissociée.

Pour enregistrer les images, le chemin de destination est construit comme ceci :

```
args.append(m_path + m_idSide[side] + ".jpg");
```

L'attribut **m_path** contient le chemin général : "**/var/tmp/img**", **img** n'est pas un répertoire, mais les noms qu'auront chaque image. Pour compléter ce nom, on vient ajouter une lettre permettant de différencier les images.

Cette lettre correspond à l'initiale de la face de la photo qui vient d'être prise. Ces lettres sont contenues dans l'attribut **m_idSide** :

```
QString m_idSide[4] = {"F", "D", "A", "G"};
```

Remarque : Ici, on n'enregistre pas les photos définitivement, mais seulement temporairement le temps que l'opérateur valide la séquence de prise de vue qui vient d'être réalisée. Ensuite, si les images prises conviennent, on les enregistrera avec une autre méthode détaillée plus loin.

Ensuite, pour récupérer les images qui viennent d'être prises on utilise une autre méthode **getImage()** pour pouvoir récupérer les images prises précédemment en fournissant en argument là face que l'on souhaite récupérer sous forme d'énumération (vu précédemment).

Voici son implémentation :

```
QImage PHOTO::getImage(face side)
{
    QString idSide;
    switch(side) {
        case PHOTO::FRONT :
            idSide = 'F';
            break;
        case PHOTO::RIGHT :
            idSide = 'D';
            break;
        case PHOTO::LEFT :
            idSide = 'G';
            break;
        case PHOTO::BACK :
            idSide = 'A';
            break;
        default:
            idSide = ' ';
    } //switch

    QImageReader reader(m_path + idSide);
    reader.setAutoTransform(true);
```

```
    return reader.read();
}
```

Dans un premier temps, afin de nommer chaque photo, un switch est utilisé pour faire correspondre chaque face avec une lettre qui lui est attribué (expliqué dans la méthode précédente « **takeShot()** »).

Ensuite, on se sert de la lettre que l'on vient de récupérer (la variable idSide) pour reconstruire le chemin ou s'ont enregistrées les images et les retourner dans un **objet QImage**.

Si les images prises ne conviennent pas, ou que l'on souhaite annuler le processus, il faut pouvoir supprimer les images qui sont stockées temporairement.

Pour cela, on utilise la méthode **deleteTempImage()** dont voici l'implémentation :

```
void PHOTO::deleteTempImage()
{
    QString cmd = "rm ";
    for (int i = 0; i < 4; i++) {
        cmd + m_path + m_idSide[i] + ".jpg";
        m_imgProcess->setProgram(cmd);
        m_imgProcess->startDetached(cmd);
    } //for
}
```

Ici nous allons chercher chaque photo et on les supprime à l'aide d'un **QProcess**, mais qui cette fois, vient exécuter la **commande rm** permettant de **supprimer des fichiers sur linux**. On **boucle quatre fois** pour supprimer chaque photo.

Dans le cas où les photos prise sont bonnes, et donc validées par l'opérateur, il faut les enregistrer.

C'est donc le rôle de la méthode **saveImages()** :

```
void PHOTO::saveImages(QString cbar)
{
    for (face i = PHOTO::FRONT; i < PHOTO::LEFT+1; i++) {
        qDebug() << i;
        m_image[i] = this->getImage(i);
    }

    QString date = QDate::currentDate().toString(QString("dd-MM-yyyy"));
    for (int i = 0; i < 4; i++) {
        QString mkdir = "sudo mkdir /home/pi/Desktop/stockPhotoShare/" + cbar +"/";
        m_createFolderProcess->setProgram(mkdir);
        m_createFolderProcess->execute(mkdir);
        mkdir = "sudo mkdir /home/pi/Desktop/stockPhotoShare/" + cbar +"/" + date + "/";
        m_createFolderProcess->setProgram(mkdir);
        m_createFolderProcess->execute(mkdir);
        m_image[i].save("/home/pi/Desktop/stockPhotoShare/"+ cbar + "/" + date + "/" + cbar
 + "-" + date + "-" + m_idSide[i] + ".jpg");
    }
}
```

On commence par **boucler sur l'énumération** pour appeler la méthode **getImage()** (vue précédemment) qui prend en paramètre une constante de l'énumération « **face** » pour rapporter les images stockés temporairement.

Remarque : Pour boucler sur l'énumération comme sur le code ci-dessus, il a fallu surcharger l'opérateur d'incrémentation pour pouvoir incrémenter l'énumération. Voici le code:

```
PHOTO::face& operator++(PHOTO::face& p)
{
    return p = static_cast<PHOTO::face>(p+1);
}

PHOTO::face operator++(PHOTO::face& p, int)
{
    PHOTO::face temp = p;
    ++p;
    return temp;
}
```

Une fois les images rapportées et stockées dans l'attribut **m_image** (qui est un tableau), la prochaine étape est de fabriquer l'arborescence des répertoires et le nom sous lequel seront enregistrées les images définitivement.

Le nom doit être simple, mais en même temps complet, c'est-à-dire contenir des informations précises pour pouvoir facilement retrouver les images. Ceci est important pour l'étape de la comparaison des différents clichés. En effet, il faut être capable de facilement retrouver les images de produits de même référence mais prises à des dates différentes. Pour cela, nous avons aussi choisi d'utiliser l'arborescence suivant pour mieux retrouver chaque image :

« **code_barre/date/date-code_barre-face.jpg** »

La **date** est au format français : « **jours-mois-année** » en **chiffre.**
Le **code barre** est scanné avant de lancer le processus de prise de vues et est passer en argument dans la méthode.
La **face** est représentée par une **lettre** comme vu précédemment.
Enfin, l'image est **enregistrée** au format **jgp**.

On commence donc par exécuter deux **QProcess** permettant de créer nos deux dossiers aux noms du code barre et de la date courante. La commande **mkdir** est utilisée :

```
QString mkdir = "sudo mkdir /home/pi/Desktop/stockPhotoShare/" + cbar +"/";
        m_createFolderProcess->setProgram(mkdir);
        m_createFolderProcess->execute(mkdir);
        mkdir = "sudo mkdir /home/pi/Desktop/stockPhotoShare/" + cbar +"/" + date + "/";
        m_createFolderProcess->setProgram(mkdir);
        m_createFolderProcess->execute(mkdir);
```

La méthode **execute()** permet d'exécuter le QProcess tout en attendant la fin de celui-ci pour passer à la suite.

Pour obtenir la date, on utilise la méthode statique **currentDate()** de la classe **QDate** que l'on convertie en chaine de caractère :

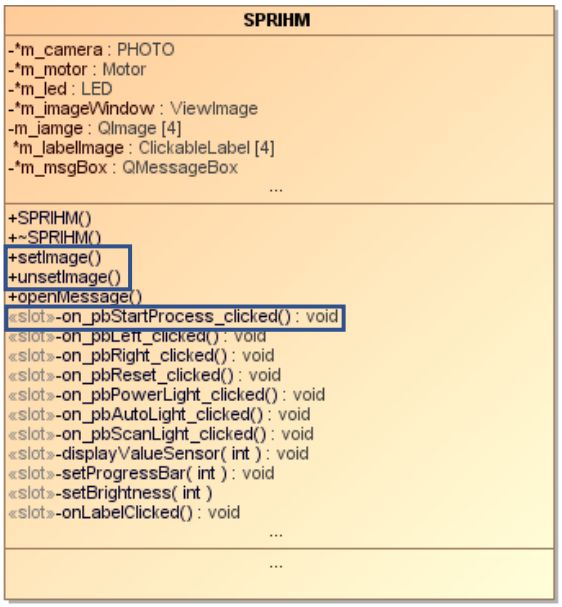
```
QString date = QDate::currentDate().toString(QString("dd-MM-yyyy"));
```

Le chemin est ensuite **construit** avec les images récupérées est **l'attribut m_idSide** pour la face.

Affichage des photos sur l'IHM :

Maintenant que la modification et la mise en application de notre classe **PHOTO** est terminée, nous pouvons l'intégrer à la classe qui gère l'IHM.

Voici le diagramme de classe de l'IHM :



Les méthodes utiles pour la partie prise de vues sont encadrées en bleu

Fichier sprihm.h :

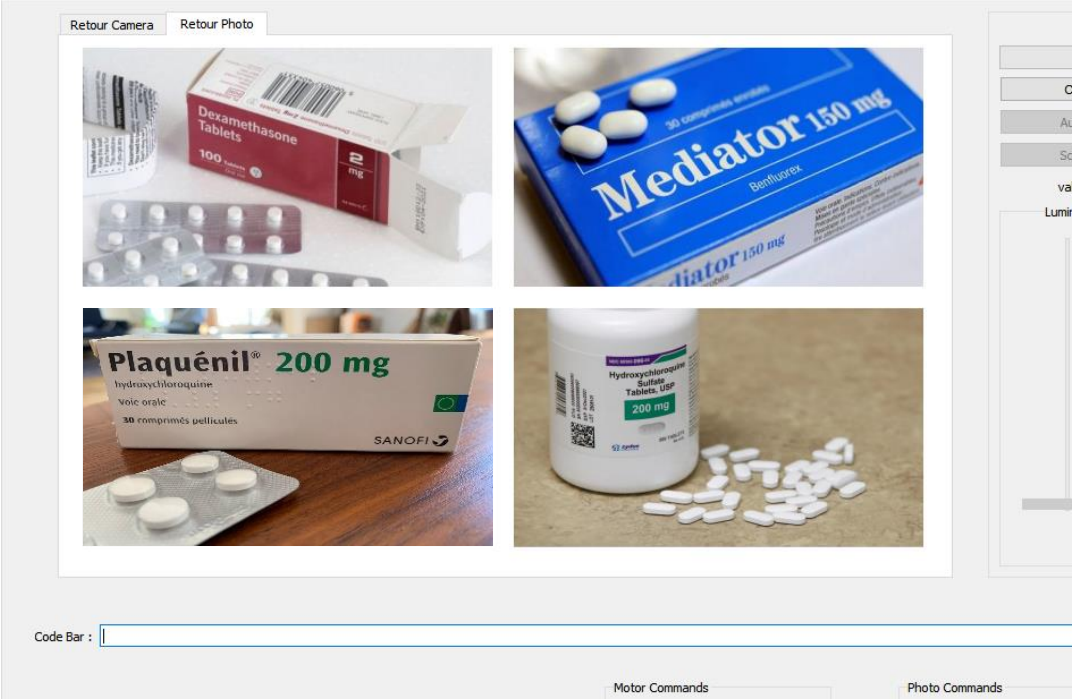
```
#ifndef SPRIHM_H
#define SPRIHM_H

#include <QMainWindow>
#include <QObject>
#include <QWidget>
#include <QPushButton>
#include <QImage>
#include <QLabel>
#include <QMessageBox>
#include <QSlider>

#include "motor.h"
#include "photo.h"
#include "led.h"
#include "clickablelabel.h"
#include "viewimage.h"

QT_BEGIN_NAMESPACE
namespace Ui { class sprIHM; }
QT_END_NAMESPACE

class sprIHM : public QMainWindow
{
    Q_OBJECT
```

```
public:
    sprIHM(QWidget *parent = nullptr);
    ~sprIHM();

    void setImage();
    void unsetImage();
    void openMessage();

private:
    Ui::sprIHM *ui;

    Motor *m_motor;
    PHOTO *m_camera;
    LED *m_led;

    ViewImage *m_imageWindow;

    QPushButton *m_bpLeft;
    QPushButton *m_bpRight;

    QSlider *m_slider;

    QImage m_image[4]; //attribute permettant de stocker les images à afficher
    ClickableLabel *m_labelImg[4];

    QMessageBox *m_msgBox;

signals:

private slots:
    void on_pbBack_clicked();
    void on_pbNext_released();

    void on_pbStartProcess_clicked();

    void on_pbPowerLight_clicked();
    void on_pbAutoLight_clicked();
    void on_pbScanLight_clicked();

    void displayValueSensor(int);
    void setProgressBar(int);

    void setBrightness(int);

    void onLabel1Clicked();
    void onLabel2Clicked();
    void onLabel3Clicked();
    void onLabel4Clicked();
};

#endif // SPRIHM_H
```

La première étape pour pouvoir utiliser la classe PHOTO est de créer une instance de celle-ci dans le constructeur de la classe SPRIHM :

```
m_camera = new PHOTO;
```

On souhaite donc pouvoir afficher les images prises sur l'IHM comme ceci :



Remarque : Les images affichées sur l'image ci-dessus n'ont pas été prises par l'équipe du projet.

Pour réaliser cet affichage, on utilise la méthode **setImage()**. Voici son implémentation :

```
void sprIHM::setImage()
{
    m_image[0] = m_camera->getImage(PHOTO::FRONT);
    m_image[1] = m_camera->getImage(PHOTO::RIGHT);
    m_image[2] = m_camera->getImage(PHOTO::LEFT);
    m_image[3] = m_camera->getImage(PHOTO::BACK);

    for (int i = 0; i < 4; i++) {
        m_labelImg[i]->setPixmap(QPixmap::fromImage(m_image[i]));
    } //for
}
```

On commence par utiliser la méthode **getImage()** (expliquée précédemment dans la classe PHOTO) pour rapporter les images qui ont été faites.
On les affiche ensuite simplement dans les objets **ClickabelLabel**, une classe héritant de **QLabel** expliquée dans la partie de l'étudiant 2.

Nous utilisons une méthode **unsetImage()** afin de pouvoir enlever les image de l'IHM lorsque nous n'avons plus besoin de celles-ci et lorsque l'on supprimera les images stockées temporairement. En effet, supprimer des images alors qu'elles sont affichées ne permet pas de les retirer de l'affichage dans les labels.

Voici le code de cette méthode :

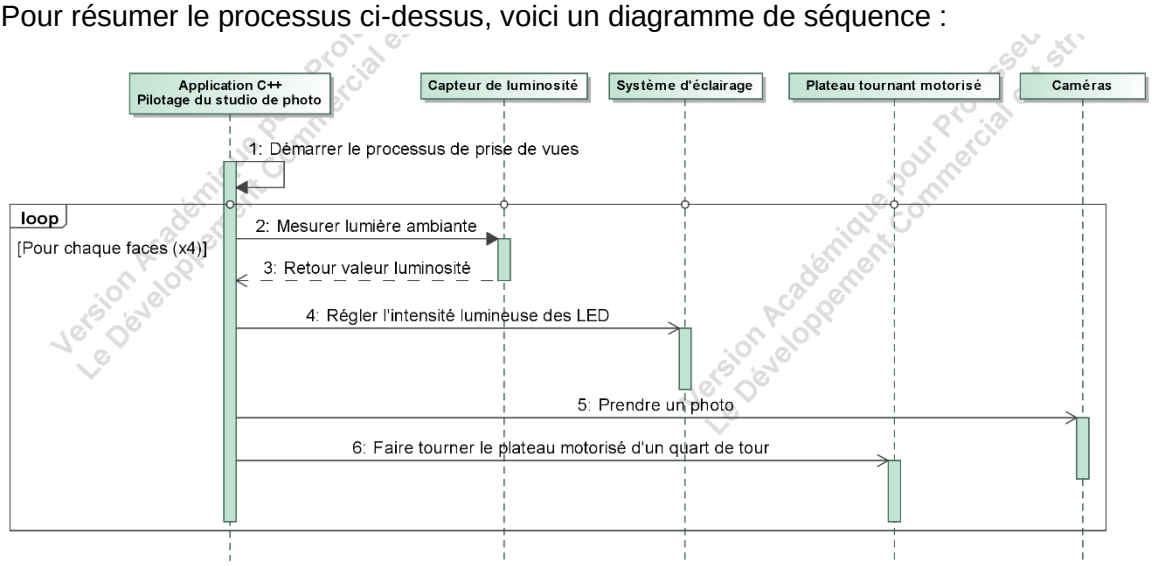
```
void sprIHM::unsetImage()
{
    for (int i = 0; i < 4; i++) {
        m_labelImg[i]->clear();
    } //for
}
```

On vient simplement boucler sur la méthode **clear()** appartenant à la classe **QLabel** permettant d'enlever tout éléments qui seraient présents dans les labels.

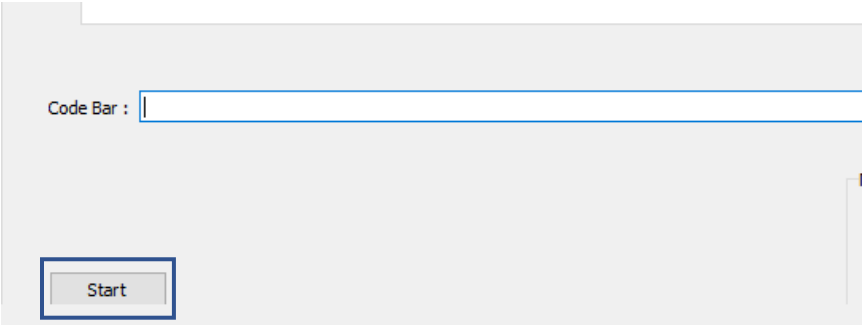
Viens maintenant le moment de prendre les photos.
La prise de vue ce fait lors de **l'enclanchement du processus** comprenant :

- La mesure de la luminosité ambiante et le réglage de la bande LED en conséquence (Partie gestion de l'éclairage de l'étudiant 3)
- La prise des clichés photos des objets disposés sur le plateau tournant.
- La rotation du plateau tournant pour prendre en photo la face suivante. (Partie gestion du plateau tournant de l'étudiant 2)

Pour résumer le processus ci-dessus, voici un diagramme de séquence :



Voici le bouton permettant de lancer ce processus sur l'IHM :



Ce bouton est connecté au slot **on_pbStartProcess_clicked()**.

Voici son implémentation :

```
void sprIHM::on_pbStartProcess_clicked()
{
    for (int i = 0; i < 4; i++){
        if (!m_motor->isRuning()) {
            m_camera->takeShot(i);
        }
        m_motor->quaterTurnR();
    } //for
    setImage();
    openMessage();
}
```

On boucle quatre fois, ce chiffre correspondant au nombre de photo à prendre.
En premier lieu, on teste l'état du moteur (le plateau est en rotation ou non) grâce à la méthode **isRuning()** traitée dans la partie de l'étudiant 2. Si le moteur ne tourne pas, on prend un photo avec la méthode **takeShot()** expliquée précédemment. Puis on fait tourner le moteur.

Lorsque les quatre photos ont été prises, on les affiche grâce à la méthode **setImage()**. Puis on appelle la méthode **openMessage()** permettant d'afficher une boite de dialogue grâce à la classe QMessageBox.

La boite de dialogue est utilisée pour demande à l'opérateur si les photos qui viennent d'être prise conviennent.

L'opérateur a ainsi les possibilités de :

- **Confirmer**, ce qui aura pour effet d'archiver les images définitivement.
- **Recommencer** tout le processus.
- **Annuler**, ce qui aura pour effet de retirer les images de l'affichage et de supprimer les photos qui viennent d'être (et donc stockées temporairement)

Voici l'implémentation de la méthode **openMessage()** :

```
void sprIHM::openMessage()
{
    m_msgBox = new QMessageBox();
    m_msgBox->setText("voulez vous sauvgarder les images");
    //msgBox.setInformativeText("Do you want to save your changes?");
    m_msgBox->setStandardButtons(QMessageBox::Save
    | QMessageBox::Retry | QMessageBox::Cancel);
    m_msgBox->setDefaultButton(QMessageBox::Save);
    int result = m_msgBox->exec();
    delete m_msgBox;

    switch (result) {
        case QMessageBox::Retry :
            QCoreApplication::processEvents();
            emit ui->pbStartProcess->clicked();
            break;
        case QMessageBox::Save :
            m_camera->saveImages();
            break;

        case QMessageBox::Cancel:
            QCoreApplication::processEvents();
            m_camera->deleteTempImage();
            unsetImage();
            break;
    } // switch
}
```

On utilise un **switch** pour agir en fonction de l'option (du bouton) qui a été choisie par l'opérateur.

Nous venons de voir tout ce qui a était fait jusqu'à présent pour la partie prise de vues.

Développement : Sélection des caméras à l'aide de la carte d'extension :

Lors de cette partie, nous allons voir la mise en place et l'utilisation de la sélection des caméras avec la **carte d'extension UC-475 v2.2**.

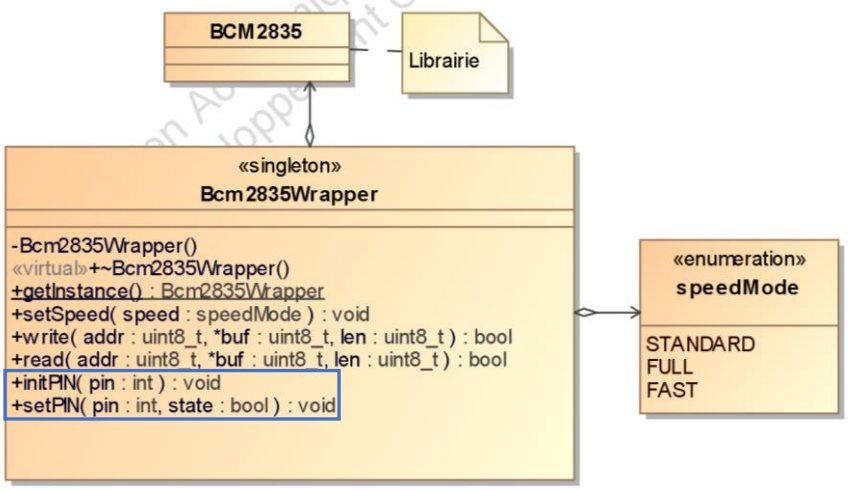
On rappelle que l'on souhaite utiliser **deux caméras** car l'une a une très **bonne qualité d'image** (nécessaire à la comparaison d'image) mais un **champ de vision restreint** pour les objets encombrant. On utilise donc une **deuxième caméra** qui a un **meilleur champ de vision** pour les objets encombrant au détriment d'une **qualité un peu moins élevée**.

La carte d'extension est pilotée par les GPIO de la Raspberry. Pour contrôler les **GPIO**, nous allons utiliser la librairie **BCM2835** permettant de **piloter** tous les **pins** de la RPI.

Conception de la classe Bcm2835Wrapper :

Cette classe a pour objectif de **faciliter l'accès à la librairie BCM2835**.

Voici le diagramme de classe :



Cette classe est dite **singleton**, c'est-à-dire qu'il ne peut y avoir **qu'une seule instance** de cette classe. Cette dernière permettant d'assurer la communication I2C et des GPIO à l'aide de la **librairie BCM2835**, elle a effectivement besoin d'avoir qu'un seul objet instancié pour chaque partie du système.

Nous nous concentrerons uniquement sur les méthodes encadrées en bleu qui concernent le pilotage des **GPIO.** Les méthodes **write()** et **read()** concernant la communication I2C ainsi que la méthode **getInstance()** permettant d'instancier la classe seront expliquées dans la partie **gestion du plateau tournant** de l'étudiant 2.

Fichier Bcm2835Wrapper.h :

```
void initPIN(int);
void setPIN(int, bool);
```

La méthode **initPIN()** nous permet de choisir nos PINS (le 4, le 17 et le 18) et de les définir en tant que sortie.

Fichier Bcm2835Wrapper.cpp :

```
void Bcm2835Wrapper::initPIN(int pin)
{
    bcm2835_gpio_fsel(pin, BCM2835_GPIO_FSEL_OUTP);
}
```

On utilise la méthode **bcm2835_gpio_fsel()** librairie **BCM2835** permettant de définir le **mode** de **communication** des **pin** (**lecture** ou **écriture**).
Voici un extrait de la documentation de cette méthode :

◆ bcm2835_gpio_fsel()

```
void bcm2835_gpio_fsel ( uint8_t pin,
                         uint8_t mode
                       )
```

Sets the Function Select register for the given pin, which configures the pin as Input, Output or one of the 6 alternate functions.

Parameters

[in] **pin** GPIO number, or one of RPI_GPIO_P1_* from **RPiGPIOPin**.

[in] **mode** Mode to set the pin to, one of BCM2835_GPIO_FSEL_* from **bcm2835FunctionSelect**

La méthode **setPIN()** nous permettra de **configurer nos pins** à l'état « **HAUT** » ou « **BAS** » (écriture ou non) afin de **définir** quelle caméra on souhaite sélectionner.

```
void Bcm2835Wrapper::setPIN(int pin, bool status)
{
    if(status){
        bcm2835_gpio_write(pin, HIGH);
    }//if status
    else {
        bcm2835_gpio_write(pin, LOW);
    }
}
```

On utilise une boolean en argument pour choisir la constante (**HIGH** ou **LOW**) que l'on souhaite utiliser.

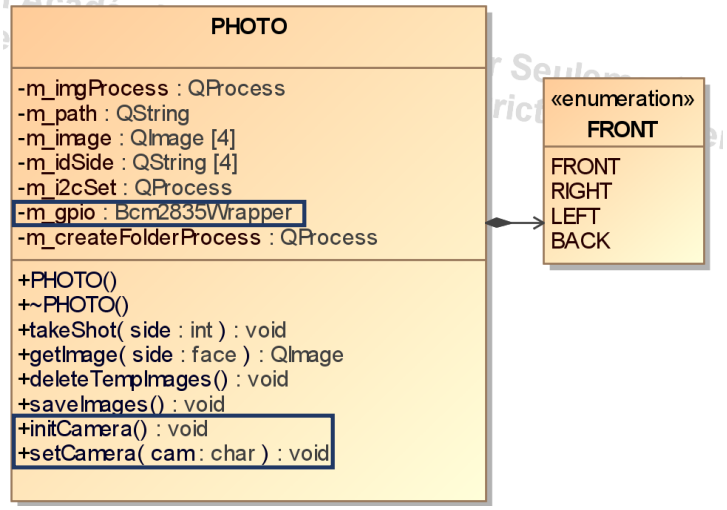
À noter : l'utilisation de la librairie nécessite de l'initialiser. Ceci se fait dans le constructeur de la classe dont le code est développé dans la partie **gestion du plateau tournant** de l'étudiant 2.

Adaptation de la classe PHOTO :

La classe PHOTO a également été modifiée.

Les méthodes **initCamera()** et **setCamera()** y ont été ajoutées.

Diagramme de la classe PHOTO :



Fichier PHOTO.h :

```
void initCamera();
void setCamera(char cam);
```

La méthode **initCamera()** nous permet de définir les PINS nécessaires pour utiliser la carte, Voici l'implémentation de cette méthode :

```
void PHOTO::initCamera()
{
    m_gpio.initPIN(4);
    m_gpio.initPIN(17);
    m_gpio.initPIN(18);
}
```

On utilise la méthode **initPIN()** de la classe **Bcm2835Wrapper** pour initialiser les pins de la carte.

À Noter :
L'attribut **m_gpio** est l'instance de la classe obtenue par la méthode **getInstance()** :

```
PHOTO::PHOTO() :
    m_gpio(Bcm2835Wrapper::getInstance())
{
    m_imgProcess = new QProcess();
    m_i2cSet = new QProcess();
    m_createFolderProcess = new QProcess();
}
```

Constructeur de la classe PHOTO

La méthode **setCamera()** quant à elle va nous permettre de sélectionner une des deux caméras.

Voici l'implémentation de cette méthode :

```
void PHOTO::setCamera(char cam)
{
    QString i2cCmd1 = "i2cset -y 1 0x70 0x00 0x04";
    QString i2cCmd2 = "i2cset -y 1 0x70 0x00 0x05";

    switch(cam){
    case 'A' :
        m_i2cSet->execute(i2cCmd1);
        m_gpio.setPIN(4, false);
        m_gpio.setPIN(17, false);
        m_gpio.setPIN(18, true);
        qDebug() << "Caméra A sélectionée";
        break;

    case 'B':
        m_i2cSet->execute(i2cCmd2);
        m_gpio.setPIN(4, true);
        m_gpio.setPIN(17, false);
        m_gpio.setPIN(18, true);
        qDebug() << "Caméra B sélectionée";
        break;

    default :
        qDebug() << "Erreur";
        break;
    } // switch
}
```

Dans un premier temps, deux variables de type **QString** sont déclarées.

Celles-ci contiennent la commande « **i2cset** » permettant d'écrire dans des registres I2C. On s'en sert ici pour sélectionner le slot de la caméra à utiliser. Le fonctionnement de cette commande est détaillé dans la partie **gestion du plateau tournant** de l'étudiant 2.

Comme on le voit dans l'utilisation de la commande, **l'adresse I2C** de la carte est **0x70**.

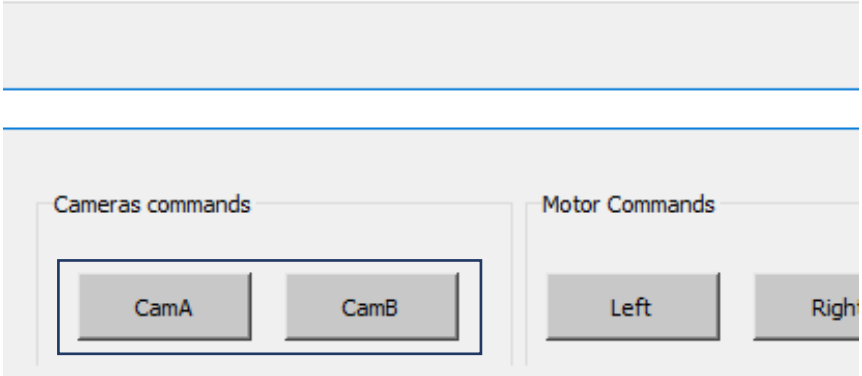
On utilise ensuite un switch qui va contrôler quelle caméra a été sélectionnée en testant la variable **cam** passée en argument de la méthode.

La commande **i2cset** est alors exécutée avec un « **Qprocess** ».

Puis les pins initialisés précédemment sont configurés en fonction de la caméra choisie grâce à la méthode **setPIN()** (méthode de la classe **Bcm22835Wrapper**).

Adaptation de la classe SprIHM :

Nous allons enfin nous attaquer aux modifications apportées à la classe SprIHM. Pour la sélection des caméras, nous avons ajouté à l'interface deux boutons, un pour sélectionner la caméra A, l'autre pour la caméra B :



À partir de là, deux méthodes ont alors été créées. Celles-ci permettent de sélectionner une des deux options du switch de la méthode **setCmera()** (vue précédemment) lorsque l'on clique dessus.

Fichier SprIHM.h :

```
void on_pbCamA_clicked();
void on_pbCamB_clicked();
```

Fichier SprIHM.cpp :

```
void sprIHM::on_pbCamA_clicked()
{
	m_camera->setCamera('A');
}

void sprIHM::on_pbCamB_clicked()
{
	m_camera->setCamera('B');
}
```

On appel la méthode **setCamera()** vue précédemment.

Nous pouvons maintenant sélectionner une des deux caméras selon nos besoins.

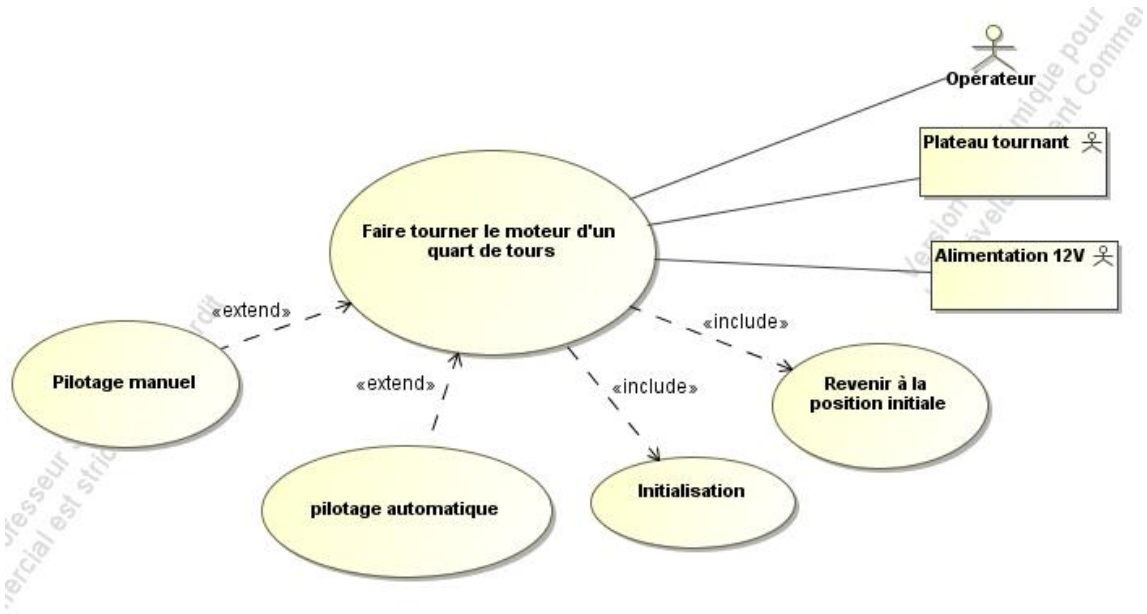
Partie Étudiant 2 : BUFFARD Gabriel (IR2)

Objectifs

- Concevoir l'IHM.
- Mettre en œuvre et piloter le plateau tournant.
- Concevoir les classes C++ permettant la communication I2C et les commandes du plateau tournant.
- Intégrer le développement au système final.

Le choix a été fait de faire fonctionner le plateau tournant avec un moteur pas à pas de 400 pas. Il permet de faire tourner les objets d'un quart de tours automatiquement ou manuellement pour réaliser des prises de vues sous tous les angles.

Diagramme des cas d'utilisations



Fonctionnement d'un moteur pas à pas

Structure et fonctionnement :

Le **contrôle** d'un moteur pas à pas se fait par l'envoi de **signaux électriques**.

Les signaux à générer dépendent de **différents facteurs** comme :

- La **vitesse désirée**.
- Le **nombre de pas** que doit effectuer le moteur.
- La **rampe d'accélération** et de **décélération**.

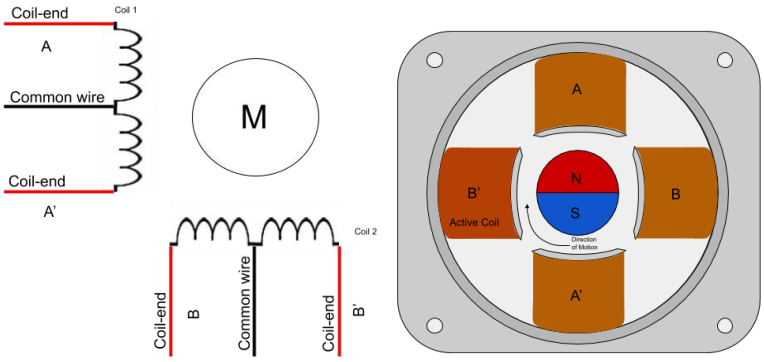
Une rotation complète est **divisée** en plusieurs « étapes » ou positions vers lesquelles le moteur peut naviguer. Ceci permet un contrôle de position du moteur en lui délivrant une commande de déplacement d'un certain nombre de pas.

Un pas correspond à un déplacement d'un certain nombre en degré. Dans notre cas, pour un moteur de 400 pas, le nombre de degrés parcouru peut être calculer comme ceci :

Un tour = **360°** Donc : **360/400 = 0.9** Un pas correspond donc à **0.9°**.

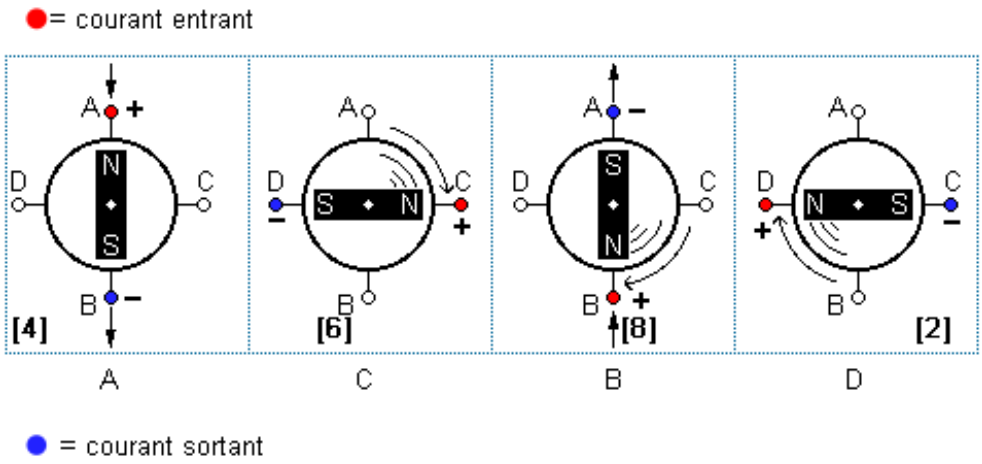
Le nombre de pas dans une rotation étant **connu**, le moteur peut être déplacé à n'importe quelle distance souhaitée en **calculant** le nombre de tours et ensuite le nombre de pas auxquels cette distance correspond.

Les moteurs pas à pas sont constitués d'un **engrenage qui est connecté à un arbre de moteur**, et entouré par plusieurs **électroaimants**. Un **électroaimant**, lorsqu'il est alimenté par un courant électrique, **convertit l'énergie électrique en énergie magnétique**. Ce dernier et constitué d'une bobine, et d'une pièce polaire en matériau **ferromagnétique**.
Le **ferromagnétisme** est le **mécanisme fondamental** par lequel certains **matériaux** (fer, cobalt, nickel, ...) sont **attirés par des aimants** ou **forment des aimants permanents**.



Les **électroaimants** sont divisés en groupes appelés « **phases** ». Chaque électroaimant d'une phase est alimenté en même temps pour **attirer** la dent la plus proche de l'engrenage vers lui.

Lorsqu'un **courant électrique** parcoure un **bobinage**, entraine l'apparition d'un **champ magnétique** et donc de pôle « **nord** » et « **sud** ». C'est sur ce principe de base que repose le fonctionnement de tout moteur électrique :



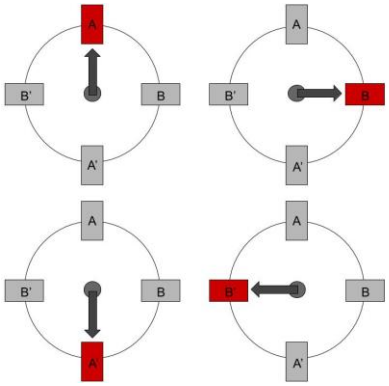
Une fois que l'engrenage s'est déplacé pour s'aligner avec la phase dont les électroaimants sont « **excités** », le moteur a **tourné d'un pas**.

Pour se déplacer d'une autre étape, la phase actuelle est désactivée et l'opération précédente se renouvelle avec la phase suivante.

Principes de bases pour la commande d'un moteur pas à pas :

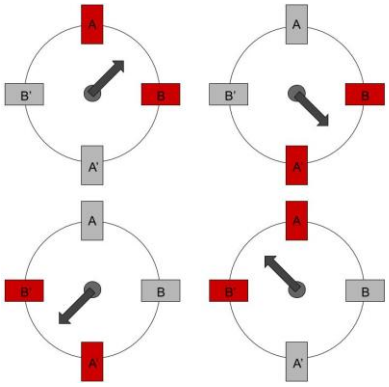
Il existe de nombreuses façons de contrôler les moteurs pas à pas. En voici trois :

- Mode 1 : « One Phase On - Full Step Drive » (aka: « Wave drive ») :
  Ceci est le mode de commande pas à pas le **plus simple**. Ici, une seule phase est activée à la fois :

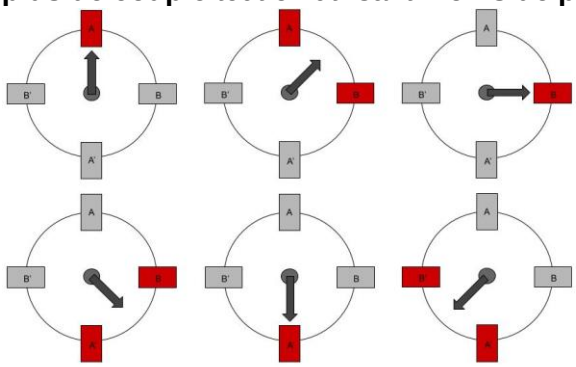


- Mode 2 : « Two Phase On - Full Step Drive » :
  Dans ce mode, **deux phases sont activées à la fois** avec la même résolution qu'une phase activée. Cependant, comme deux phases sont activées à la fois, ce mode offre **plus de couple** au prix de nécessiter **plus de puissance** que les autres

modes de commande :



- <u>Mode 3 : « Half Step Drive »</u>
  Dans ce mode, le moteur alterne entre une et deux phases sous tension à la fois. Cela permet à un demi-pas d'avoir deux fois la résolution du pas complet. Ce mode permet **d'égaliser les avantages** des deux modes précédents en permettant d'avoir **plus de couple** tout en utilisant **moins de puissance** :



Source : https://labjack.com/support/app-notes/digital-IO/stepper-motor-controller

Au siens du projet, le moteur est piloté en « **One phase One** »

Prototypage : Mise en œuvre rapide

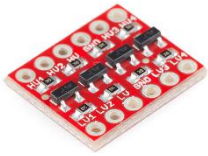
L'étudiant EC1 : Électeur Tony, se chargeant en tout premier lieu de mettre en œuvre la **carte électronique** qui contrôle le moteur, j'ai dû travailler avec du matériel **alternatif mais équivalent** au définitif, pour pouvoir commencer à travailler sur la partie informatique du plateau tournant.

Matériel utilisé :

- Contrôleur PCA9629A :
  Le PCA9629A est un dispositif **CMOS basse puissance** contrôlé par **bus I2C** qui fournit toutes les logiques et commandes nécessaires pour entraîner un **moteur pas à pas à quatre phases**.

- L293D :
  Le L293D permet de fournir des **courants de commande bidirectionnels** allant jusqu'à **600 mA** à des tensions de **4,5 V** à **36 V**. Il permet de piloter des charges inductives telles que des relais, des solénoïdes, des **moteurs pas à pas** à courant continu et bipolaires, ainsi que d'autres charges à courant élevé / haute tension dans les applications d'alimentation positive.

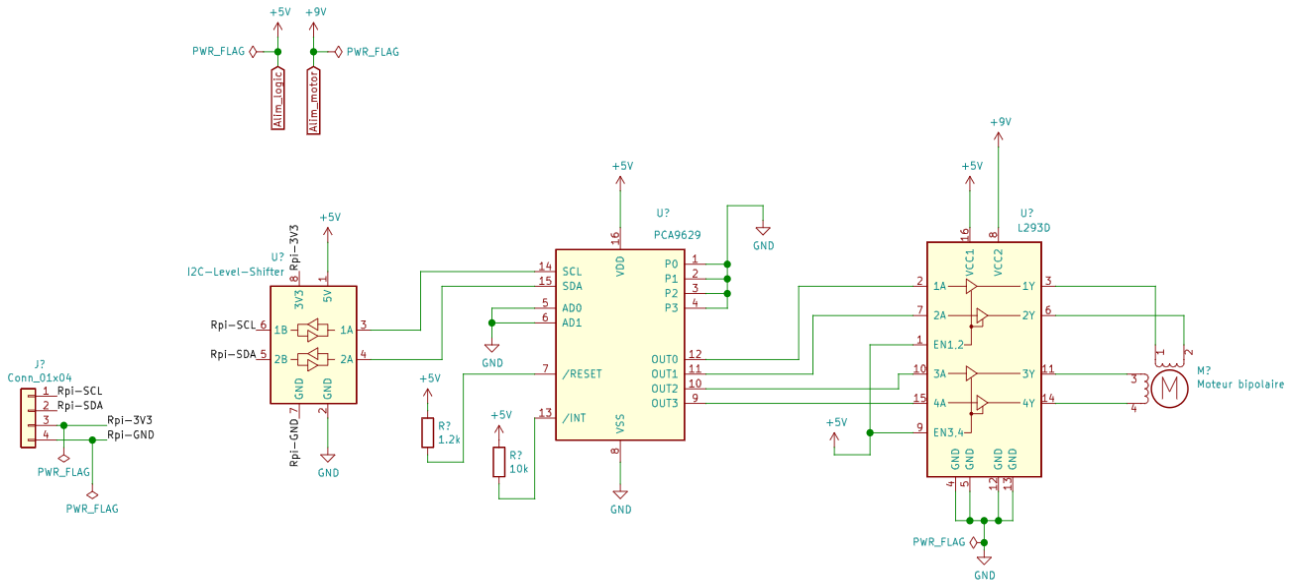
- Level Shifter I2C :
  Permet de **transformer les signaux d'un niveau logique à un autre** (3.3V, 5V…). La Raspberry envoyant du 3.3V pour les signaux I2C (SDA et SCL), Le level shifter est nécessaire pour que le PCA9629A puisse interpréter les signaux en 5V.



- Moteur pas à pas 400 pas 9V :
  Il s'agit de la pièce maîtresse permettant la rotation du plateau tournant.

- Breadboard (plaque de prototypage rapide) :
  Permet d'assurer le câblage entre tous les composants sans avoir à les souder. Cela permet de faire des essais électroniques rapidement.

Schéma du montage :



Mise en œuvre rapide du moteur :

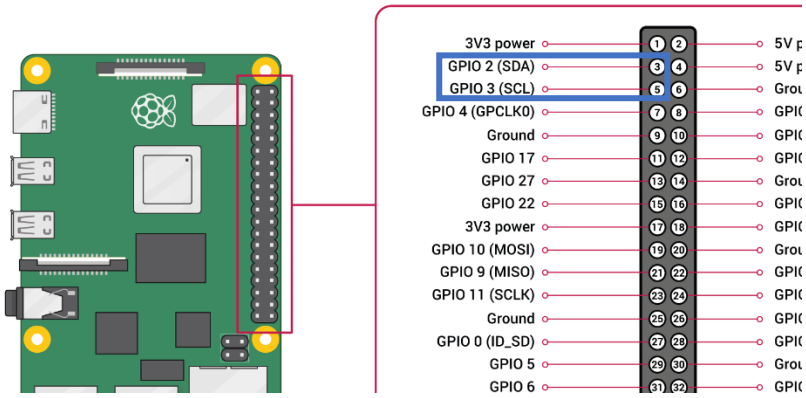
Pour commencer, j'ai utilisé un **script shell** fait depuis la Raspberry à l'aide de la documentation du PCA9629A.

Création du script : `/script$ touch motor.sh`

Le script utilise principalement la commande « **i2cset** » dont voici la signature :
« i2cset [-f] [-y] [-m mask] [-r] i2cbus chip-address data-address [value] ... [mode] »

Source : https://linux.die.net/man/8/i2cset

Cette commande permet de définir **des registres via un bus I2C**. Ici, ce sont les **GPIO SDA et SCL** de la Raspberry qui permettent de générer ces signaux :



Réalisation d'un script permettant de faire faire au moteur un tour complet en quatre temps (quart par quart) :

Le script permet de faire faire au moteur **un tour en quatre temps** (un quart de tour) sans utiliser de rampe d'accélération pour le moment.

En se basant sur la **documentation** suivante : https://www.nxp.com/docs/en/application-note/AN11483.pdf permettant de programmer le contrôleur étape par étape, et en utilisant la **documentation technique** de ce dernier : https://www.nxp.com/docs/en/data-sheet/PCA9629A.pdf , j'ai pu élaborer le script suivant :

- Étape 1 : Configuration de la vitesse du moteur :
  Selon la doc :

4.1.1 Step 1: Set step pulse width in CWPWH/L and CCWPWH/L registers

Set the step pulse width in CWPWH/L and CCWPWH/L registers for motor running speed in clockwise or counter-clockwise direction.

Les registres **CWPWH et CWPWL** permettent de définir le nombre et la durée des impulsions envoyées (« step pulse width »).

En regardant la documentation technique, on trouve une description plus détaillée des registres :

7.3.19 CWPWL, CWPWH — Clockwise step pulse width register

This register determines the step pulse width used for the phase sequence output waveforms during ClockWise (CW) rotation.

Table 26. CWPWL, CWPWH - Clockwise step pulse width control register (address 16h, 17h) bit description

*Legend: * default value.*

Address	Register	Bit	Access	Value	Description
16h	CWPWL	7:0	R/W	00h*	step pulse width, low byte
17h	CWPWH	7:0	R/W	00h*	step pulse width, high byte

This register sets the pulse width value between 3 µs and 3145 ms (±3 %).

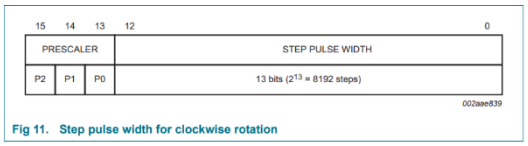


Fig 11. Step pulse width for clockwise rotation

The upper three bits of the register are the prescaler that determines the dynamic range for the step pulse width. Table 27 shows the range for each setting of the prescaler.

Ici, on voit que les deux registres **ont les adresses 0x16 et 0x17** et ont donc une **taille totale de 16 bits**. La documentation nous indique que on peut **écrire dans plusieurs registres en même temps** (sur 16 bits), lorsque ceux-ci vont ensemble. Pour obtenir la nouvelle adresse, il faut **passer le bit de poids fort à 1**.

Donc, la nouvelle adresse = **0x16** = **0b0001 0110** → **0b1001 0110** = **0x96**

Pour la valeur, les **trois premiers bits de poids fort** permettent de sélectionner le « **PRESCALER** ».

Le **PRESCALER** permet de sélectionner la **plage de la durée d'une impulsion**. Voici toutes les plages possibles **codées sur 3 bits** :

Table 27. Prescaler range settings

Prescaler [P2:P0]	Decimal value (D)	2^D	Range
000	0	1	3 µs to 24.576 ms
001	1	2	6 µs to 49.152 ms
010	2	4	12 µs to 98.304 ms
011	3	8	24 µs to 196.608 ms
100	4	16	48 µs to 393.216 ms
101	5	32	96 µs to 786.432 ms
110	6	64	192 µs to 1572.864 ms
111	7	128	384 µs to 3145.728 ms

Plus la plage est faible, plus le moteur tournera rapidement.

Avec les 13 autres bits restants, on définit le **nombre d'impulsions**. Pour la plage de durée minimum, Il faut au **minimum la valeur 1A** pour que le moteur puisse tourner correctement. J'ai obtenu cette valeur expérimentalement. Ceci nous donne alors la **vitesse maximale** à laquelle peut tourner le moteur sans **comportements indésirés**.

Commande complète : **i2cset -y 1 0x20 0x96 0x010a w**
Quelques remarques :

- L'option **-y** permet d'interagir avec le bus i2c en désactivant le mode « interaction ». Par défaut, **i2cset** attend confirmation de l'utilisateur pour interagir avec le bus. Ce mode est utile avec les scripts.

- **1** est le numéro du bus

- **0x20** est l'adresse de l'esclave I2C PCA9629A :

  7.1 Device address

  Following a START condition, the bus master must send the target slave address followed by a read or write operation. The slave address of the PCA9629A is shown in Figure 3. Slave address pins AD1 and AD0 choose one of 16 slave addresses. To conserve power, no internal pull-up resistors are incorporated on AD1 and AD0. Table 4 shows all 16 slave addresses by connecting the AD0 and AD1 to V_{DD}, V_{SS}, SCL or SDA.

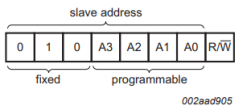
  

  Fig 3. PCA9629A device address

- **0x96** est l'adresse du registre vu précédemment.

- **0x010a** est la valeur que l'on écrit dans le registre nous permettant d'avoir 266 impulsions de 3µs par seconde.

- L'option **w** permet d'indiquer que l'on fait une écriture sur **16 bits** (16-bit Word)

- <u>Étape 2 : Configuration du nombre de pas :</u>

  4.1.2 Step 2: Set number of steps in CWSCOUNTH/L and CCWSCOUNTH/L registers

  Set the number of steps in CWSCOUNTH/L and CCWSCOUNTH/L registers for motor turning steps in clockwise or counter-clockwise direction.

Les registres **CWSCOUNTL et CWSCOUNTH** permettent de configurer **le nombre de pas que le moteur doit faire** :

7.3.17 CWSCOUNTL, CWSCOUNTH — Number of clockwise steps register

This register determines the number of steps the motor should turn in clockwise direction.

Table 24. CWSCOUNTL, CWSCOUNTH - Number of clockwise steps count register (address 12h, 13h) bit description

*Legend: * default value.*

Address	Register	Bit	Access	Value	Description
12h	CWSCOUNTL	7:0	R/W	00h*	number of clockwise steps, low byte
13h	CWSCOUNTH	7:0	R/W	00h*	number of clockwise steps, high byte

En suivant la même logique que précédemment, on trouve l'adresse **0x92** pour écrire sur les deux registres (16 bits).

Puis, il suffit d'indiquer le nombre de pas que le moteur doit faire **sur 16 bits**. Dans mon cas, je veux que mon moteur effectue un quart de tour (il se déplace quart par quart jusqu'à faire un tour complet).

Sachant que j'utilise un moteur de **400 pas**, pour faire un tour complet, le moteur doit se déplacer de 400 étapes. Donc :
400 / 4 = 100. Un quart de tour correspond donc à **100 pas**.

Donc : 100 = 0x64

Remarque : Dans le cas où on utiliserait une **rampe d'accélération** et/ou de **décélération** permettant de faire augmenter et diminuer la vitesse **progressivement**, il faudrait diminuer le nombre de pas car sinon, le moteur tournerait plus loin que prévu car **les rampes utilisent déjà un certain nombre de pas**.
Le bon nombre de pas à définir peut se trouver de façon expérimentale en consultant les registres : **STEPCOUNT0 à STEPCOUNT4**, qui mémorisent le nombre de pas effectué sur 32 bits, après un lancement de rotation.

Commande complète : **i2cset -y 1 0x20 0x92 0x0064 w**

- Étape 3 : Lancer la rotation :
  Le registre **MCNTL** sert **d'interface principale** pour contrôler le moteur (lancer la rotation, le stopper, le redémarrer, choisir le sens de rotation…) :

Table 30. MCNTL - Motor control register (address 1Ah) bit description

*Legend: * default value.*

Address	Register	Bit	Access	Value	Description
1Ah	MCNTL	7	R/W	1	start motor
				0*	stop motor
		6	R/W	1	re-start motor for new speed and operation
				0*	self clear after new speed starts running
		5	R/W	1	emergency stop motor
				0*	self clear after motor stop and bit 7 also clears to 0
		4	W only	1	enable START (bit 7) ignore caused by P0 state
				0*	disable START (bit 7) ignore caused by P0 state
		3	W only		P0 polarity setting for START (bit 7) ignore
				1	set P0 input state is HIGH to ignore START bit 7
				0*	set P0 input state is LOW to ignore START bit 7
		2	R only	0*	reserved
		1:0	R/W	11	rotate counter-clockwise first, then clockwise
				10	rotate clockwise first, then counter-clockwise
				01	rotate counter-clockwise
				00*	rotate clockwise

Ce registre possède **l'adresse 0x1A**, et prend une valeur sur **8 bits**.
Je lui donne donc la valeur **C0** à partir de la documentation, pour le faire tourner en mode rotation horaire.

A noter que pour une rotation antihoraire, il faut répéter les étapes précédentes, mais avec les registres pour la rotation antihoraire. Ils sont facilement identifiables car ils commencent par un C supplémentaire pour « counter-clockwise ». Par exemple, les registres « **CWSCOUNTL et CWSCOUNTH** » deviennent « **CCWSCOUNTL et**

CCWSCOUNTH »

Commande complète : **i2cset -y 1 0x20 0x1a 0xc0**
On peut noter l'absence de l'option **w** car on écrit que sur 8 bits.

- <u>Étape 4 : Stopper le moteur :</u>
  Il suffit de modifier la valeur du registre précédent à **0x00**

  À noter que dans le cas de mon script, cette étape est **dispensable**, car le moteur s'arrête après avoir terminé de se déplacer du nombre de pas défini.

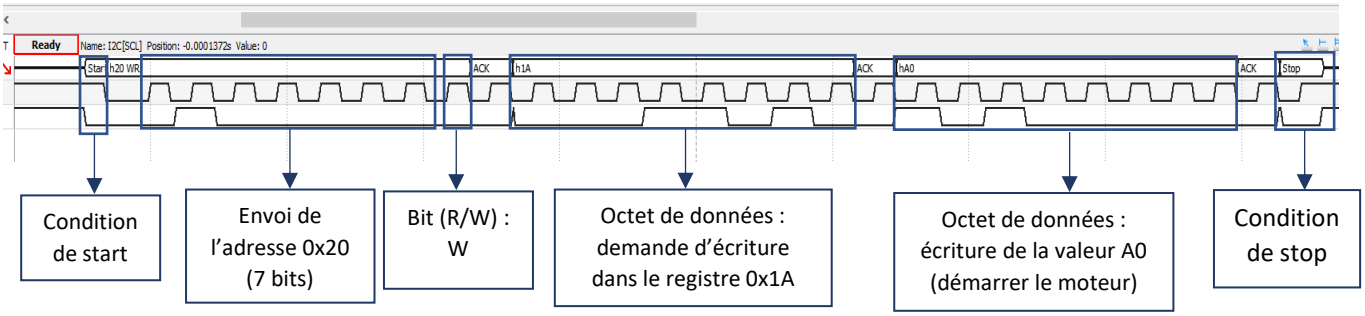
  Commende complète : **i2cset -y 1 0x20 0x1a 0x00**

Voici le script complet :

```
# 1/ Configurer la vitesse de rotation horaire
# . CWPWL & CWPWH : Step pulse width for CW rotation (Low & High bytes)
#     CWPW = 0x010a => Prescaler = 0 & Pulse width = 0x10a
#                   => 266 impulsions de 3µs par seconde
i2cset -y 1 0x20 0x96 0x010a w


# 2/ Configurer le nombre de pas désiré pour la rotation horaire
#   CWSCOUNTL & CWSCOUNTH : Number of steps CW (Low & High bytes)
#   CWSCOUNT = 0x0064 => 100 pas
i2cset -y 1 0x20 0x92 0x0064 w

# 3/ Lancer rotation horaire
# . MCNTL : Motor start/stop and rotate direction control
#     MCNTL[7] = 1 => start motor
#     MCNTL[6] = 1 => re-start motor for new speed and operation
#     MCNTL[1:0] = 00 => rotate clockwise
i2cset -y 1 0x20 0x1a 0xc0


# Attendre fin de rotation (100/266 => t >= 1s)
sleep 1


#Lancer la rotation horaire
i2cset -y 0x20 0x1a 0xc0


# Attendre fin de rotation
sleep 1


#Lancer la rotation horaire
i2cset -y 0x20 0x1a 0xc0


# Attendre fin de rotation
sleep 1


#Lancer la rotation horaire
i2cset -y 0x20 0x1a 0xc0
```

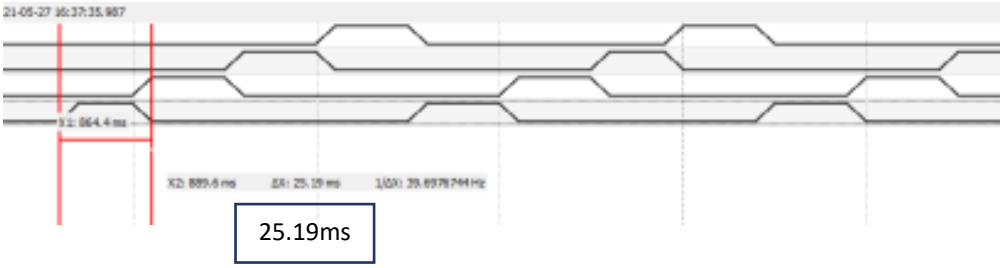
```
# Attendre fin de rotation
sleep 1

# 4/ Stopper le moteur
#    MCNTL : Motor start/stop and rotate direction control
#    MCNTL[7] = 0 => start motor
i2cset -y 1 0x20 0x1a 0x00
```

Voici l'exemple d'une trame I2C générée par la Raspberry pour commander le PCA9629A :



Cette capture de trame a été réalisée grâce à un **Analog Discovery 2**. La trame ci-dessous correspond à la commande permettant de démarrer le moteur.

Voici les impulsions générer par le PCA9629A en sortie :



Un **curseur** a été placé pour faire une **mesure approximative** de la **durée d'un pas** qui est environ égale à **25ms** ce qui est proche de la vitesse configurée pouvant aller jusqu'à **24.5ms**.

De plus, la **succession des pas** nous montre que l'on tourne dans le **sens horaire**.

À noter que sur **l'application finale**, il faudra sûrement **mettre en œuvre les rampes d'accélération et de décélération** pour éviter que les objets sur le plateau tournant **tombent, due à une vitesse trop brutale** au démarrage du moteur. Ainsi, l'accélération se ferait progressivement.

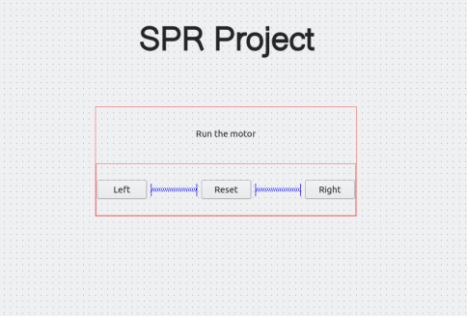
La mise en œuvre rapide du moteur étant faite, il reste à **traduire le script précédent en C++** pour être capable de faire la même chose depuis une **interface graphique**, et ainsi, intégrer le développement au reste du système (prise de vue, éclairage…).

Prototypage : Conception d'une IHM en C++

L'objectif ici est de transformer le script de la partie précédente en **C++** pour être capable de contrôler le moteur depuis une **IHM** (interface homme machine).

Pour la conception du programme, je vais utiliser le célèbre **Framework Qt** et travailler depuis **QT Creator**.

Voici l'IHM réalisée depuis **QT Designer** :



La première étape est de coder la classe permettant la **communication I2C** entre la RPI et le PCA9629A.

Pour pouvoir assurer cette communication, nous utilisons la **librairie bcm2835**, qui est une librairie en **C compatible C++**, permettant **d'accéder aux GPIO** de la RPI.

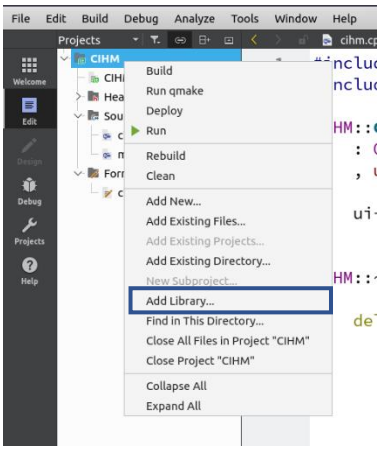
Installation de la librairie :

```
# Télécharger la dernière version de la librairie, puis faire les commandes suivantes :
tar zxvf bcm2835-1.xx.tar.gz
cd bcm2835-1.xx
./configure
make
sudo make check
sudo make install
```

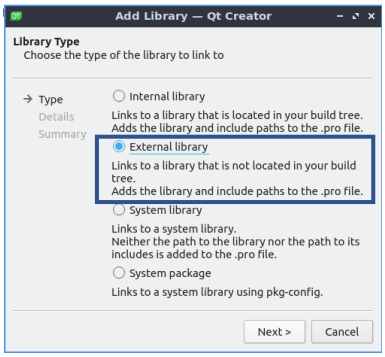
Source : https://www.airspayce.com/mikem/bcm2835/

Importer la librairie dans QT Creator :

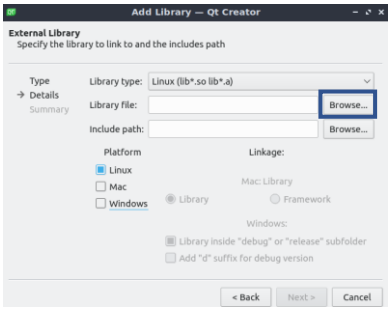
1)



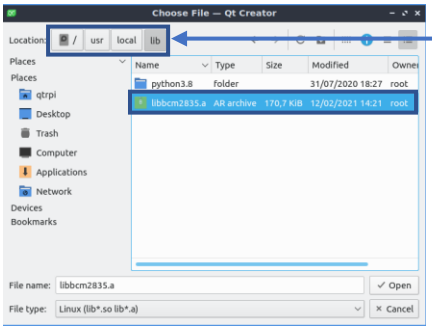
2)



3)

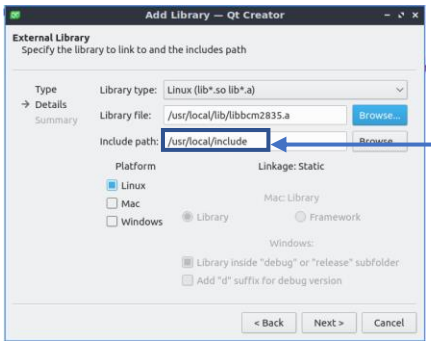


4)



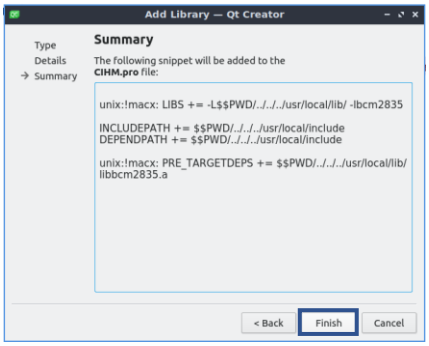
Chemin d'accès à la librairie

5)



Si tout s'est bien passé, le champ « include path » doit s'autocompléter

6)



Une fois la librairie installée, je peux maintenant coder la classe suivante :

Réalisation de la classe Bcm2835Wrapper :

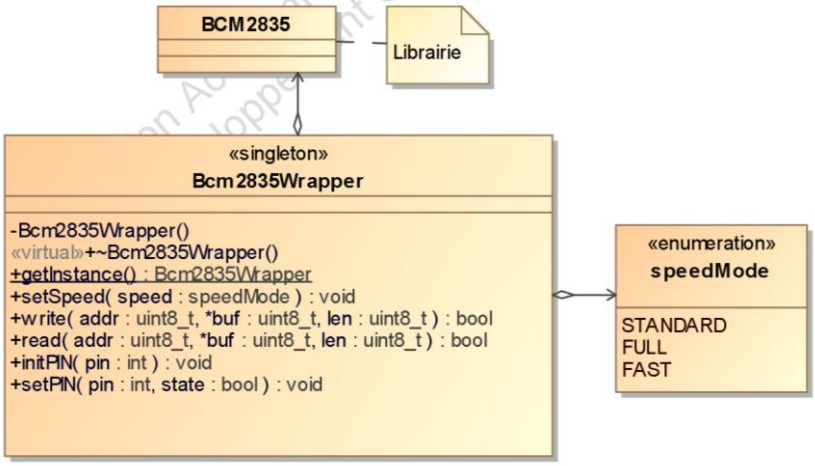


Diagramme de classe «Bcm2835Wrapper »

La classe **Bcm2835Wrapper** est une classe dite « **singleton** ».

Cela signifie qu'il ne peut y avoir **qu'une seule instance** de cette classe. Cette dernière permettant d'assurer la communication I2C et des GPIO à l'aide de la librairie BCM2835, elle a effectivement besoin d'avoir qu'un seul objet instancié pour chaque partie du projet.

De plus, cette méthode permet d'utiliser **la même instance dans tout le code**. Cela sera utile lorsque l'on fera l'intégration du moteur avec le reste du système, notamment la partie éclairage, qui utilise aussi le **protocole I2C**.

De plus, la librairie **nécessite une initialisation** avant de pouvoir utiliser ses méthodes. Ainsi, l'initialisation se fait une seule fois.

Ainsi, il ne suffira que **d'une seule instanciation de cette classe pour pouvoir faire fonctionner tous les équipements**.

Le code de la classe :

Fichier Bcm2835Wrapper.h :

```
#ifndef BCM2835WRAPPER_H
#define BCM2835WRAPPER_H

#include <QObject>
#include <bcm2835.h>

/*
 * Classe Bcm2835Wrapper
 * Gère la communication I2C avec la RPI
 * Utilise la librairie BCM2835
 */

class Bcm2835Wrapper {

private:
    Bcm2835Wrapper();

public:
    // Rendre inaccessibles les constructeurs de copie/déplacement et opérateurs d'affectat
ion/déplacement
    Bcm2835Wrapper(Bcm2835Wrapper const&) = delete;
    Bcm2835Wrapper(Bcm2835Wrapper&&) = delete;
    Bcm2835Wrapper& operator=(Bcm2835Wrapper const&) = delete;
    Bcm2835Wrapper& operator=(Bcm2835Wrapper &&) = delete;

    virtual ~Bcm2835Wrapper();

    typedef enum {STANDARD, FULL, FAST} speedMode;

    static Bcm2835Wrapper& getInstance();

    void setSpeed(speedMode speed);
    bool write(uint8_t addr, uint8_t* buf, int len);
    bool read(uint8_t addr, uint8_t* buf, int len);

    void initPIN(int);
    void setPIN(int, bool);

};

#endif // Bcm2835Wrapper_H
```

On fait en sorte que les **constructeurs de copie** et **les opérateurs d'affectation** soient **inaccessibles** en leurs assignant la valeur « **delete** ». Ainsi, on interdit toutes instanciations dynamiques de cette classe à l'aide des constructeurs surchargés.

Le **constructeur** classique, quant à lui est en **private** pour le rendre **inaccessible**.

La **méthode statique** « **getInstance()** » permet donc d'obtenir l'instance unique de cette classe. Voici son implémentation :

```
Bcm2835Wrapper& Bcm2835Wrapper::getInstance() {
    static Bcm2835Wrapper instance;
    return instance;
}
```

On créer un attribut statique du type de la classe, et on le retourne.

Les méthodes « **read()** » et « **write()** » permettent de **lire et écrire** dans les **registres I2C** en utilisant la librairie **BCM2835**.

Extrait de la documentation de la librairie BCM2835 :

◆ bcm2835_i2c_read()

```
uint8_t bcm2835_i2c_read ( char *    buf,
                           uint32_t  len
                         )
```

Transfers any number of bytes from the currently selected I2C slave. (as previously set by

See also

bcm2835_i2c_setSlaveAddress)

Parameters

[in] **buf** Buffer of bytes to receive.

[in] **len** Number of bytes in the buf buffer, and the number of bytes to received.

Returns

reason see bcm2835I2CReasonCodes

◆ bcm2835_i2c_write()

```
uint8_t bcm2835_i2c_write ( const char *  buf,
                            uint32_t      len
                          )
```

Transfers any number of bytes to the currently selected I2C slave. (as previously set by

See also

bcm2835_i2c_setSlaveAddress)

Parameters

[in] **buf** Buffer of bytes to send.

[in] **len** Number of bytes in the buf buffer, and the number of bytes to send.

Returns

reason see bcm2835I2CReasonCodes

Source : http://www.airspayce.com/mikem/bcm2835/group__i2c.html

Voici l'implémentation des méthodes **read()** et **wite()** :

```
bool Bcm2835Wrapper::read(uint8_t addr, uint8_t* buf, int len)
{
    bcm2835_i2c_setSlaveAddress(addr);
    uint8_t status = bcm2835_i2c_read(reinterpret_cast<char*>(buf), len);
    return status == BCM2835_I2C_REASON_OK ? true : false;
}

bool Bcm2835Wrapper::write(uint8_t addr, uint8_t* buf, int len)
{
    bcm2835_i2c_setSlaveAddress(addr);
    uint8_t status = bcm2835_i2c_write(reinterpret_cast<char*>(buf), len);
    return status == BCM2835_I2C_REASON_OK ? true : false;
}
```

Ces méthodes permettent donc de simplifier l'accès à la librairie BCM2835, et retournent une valeur boolean (true si l'écriture ou la lecture est un succès, ou false dans le cas contraire si la communication ne s'est pas établie correctement).

La librairie a de besoin d'initialiser la communication pour pouvoir fonctionner, et ce, grâce aux méthodes **bcm2835_init()** et **bcm2835_i2c_begin()** que l'on vient appeler dans le constructeur de cette classe. Voici le code du constructeur de la **classe Bcm2835Wrapper** :

```
Bcm2835Wrapper::Bcm2835Wrapper() {
    if ( bcm2835_init() && bcm2835_i2c_begin()) {
        setSpeed(Bcm2835Wrapper::STANDARD);
    } else {
        qDebug() << "Erreur initialisation librairie I2C - BCM2835" << endl;
    } //else
}
```

Si la communication s'initialise correctement, on vient définir la vitesse de communication grâce à la méthode **setSpeed()** dont voici le code :

```
void Bcm2835Wrapper::setSpeed(speedMode speed) {
    int br;

    switch(speed) {
        case Bcm2835Wrapper::STANDARD :
            br = 100000;
            break;
        case Bcm2835Wrapper::FULL :
            br = 400000;
            break;
        case Bcm2835Wrapper::FAST :
            br = 1000000;
            break;
        default:
            br = -1;
            break;
    } //switch

    if(br > 0) {
        bcm2835_i2c_set_baudrate(br);
    } //if
}
```

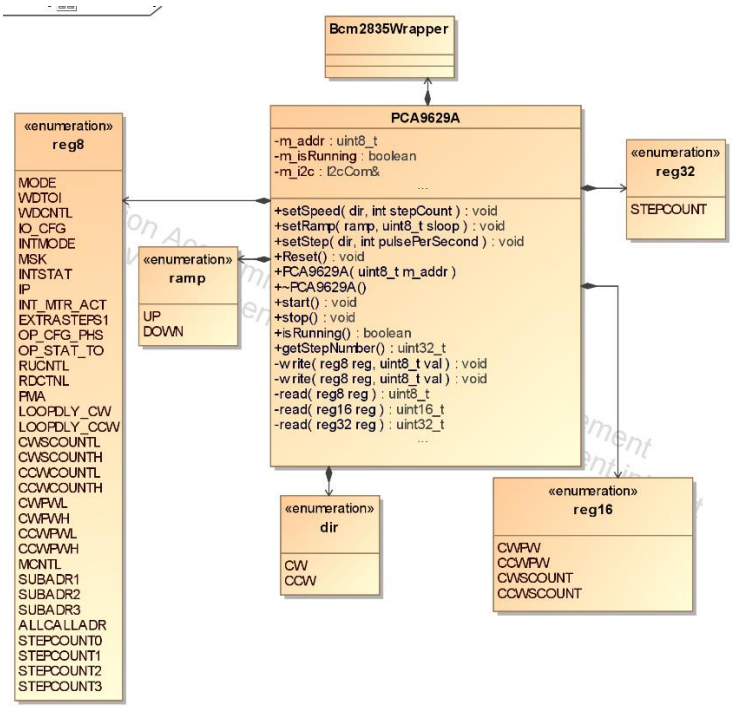
On utilise la méthode **bcm2835_i2c_set_baudrate()** pour définir la vitesse de communication dont la valeur est définie grâce à une énumération.

Dans le constructeur, on initialise cette vitesse en **STANDARD** ce qui correspond à une **vitesse de transmission de données** de **100Kbits/s** qui suffira largement pour notre application.

La prochaine étape est de concevoir la classe qui permettrait **d'interagir avec le PCA9629A** par l'intermédiaire de la classe « **Bcm2835Wrapper** ».

Réalisation de la classe PCA9629A

Voici le diagramme de classe :



Cette classe permet donc de faire la même chose que le **script shell** mit en œuvre précédemment à la différence qu'il est prévu d'utiliser les rampes d'accélération et de décélération ici.

Des **énumérations sont utilisées** pour représenter tous les registres de la documentation technique du contrôleur, et ce, même s'ils ne sont pas tous utilisés.

Le code de la classe :

Fichier PCA9629A.h :

```
#ifndef PCA9629A_H
#define PCA9629A_H

#include <QObject>
#include "bcm2835wrapper.h"

/*
 * Classe PCA9629A
 * Gère tous les registres PCA9629A via une communication I2C
 * Utilise la Classe Bcm2835Wrapper
 */

class PCA9629A {
public:
    /** keyword to select direction of rotation */
    typedef enum {
        CW = 0, /**< Clockwise direction */
        CCW /**< ConterClockwise direction */
    } dir;

    typedef enum {
        UP,
        DOWN
    } ramp;

    PCA9629A(uint8_t addr = 0x20);
    virtual ~PCA9629A();

    void Reset();
    void start(dir dir);
    void stop();
    void setStep(dir dir, int stepCount);
    uint16_t setSpeed(dir dir, int pulsePerSecond);
    void setRamp(ramp ramp, uint8_t slope);
    bool isRunning();
    uint32_t getStepNumber();

private:
    const double  STEP_RESOLUTION = 1/(3e-6);
    typedef enum {
        MODE, /**< 0x00   Mode register                                    */
        WDTOI, /**< 0x01   WatchDog Time-Out Interval register              */
        WDCNTL, /**< 0x02   WatchDog Control register                      */
        IO_CFG, /**< 0x03   I/O Configuration register                     */
        INTMODE, /**< 0x04   Interrupt Mode register                       */
        MSK, /**< 0x05   Mask interrupt register                          */
        INTSTAT, /**< 0x06   Interrupt Status register                     */
        IP, /**< 0x07   Input Port register                                */
        INT_MTR_ACT, /**< 0x08   Interrupt motor action control register   */
        EXTRASTEPS0, /**< 0x09   Extra steps count for INTP0 control register */
        EXTRASTEPS1, /**< 0x0A   Extra steps count for INTP1 control register */
        OP_CFG_PHS, /**< 0x0B   Output Port Configuration and Phase control register */
        OP_STAT_TO, /**< 0x0C   Output state and time-out control register */
```

```
        RUCNTL, /**< 0x0D   Ramp-up control register                                */
        RDCNTL, /**< 0x0E   Ramp-down control register                              */
        PMA, /**< 0x0F   Perform multiple of actions control register            */
        LOOPDLY_CW, /**< 0x10   Loop delay timer for CW to CCW control register         */
        LOOPDLY_CCW, /**< 0x11   Loop delay timer for CCW to CW control register         */
        CWSCOUNTL, /**< 0x12   Number of clockwise steps register (low byte)           */
        CWSCOUNTH, /**< 0x13   Number of clockwise steps register (high byte)          */
        CCWSCOUNTL, /**< 0x14   Number of counter-clockwise steps register (low byte)   */
        CCWSCOUNTH, /**< 0x15   Number of counter-clockwise steps register (high byte)  */
        CWPWL, /**< 0x16   Clockwise step pulse width register (low byte)          */
        CWPWH, /**< 0x17   Clockwise step pulse width register (high byte)         */
        CCWPWL, /**< 0x18   Counter-clockwise step pulse width register (low byte)  */
        CCWPWH, /**< 0x19   Counter-clockwise step pulse width register (high byte) */
        MCNTL, /**< 0x1A   Motor control register                                  */
        SUBADR1, /**< 0x1B   I2C-bus subaddress 1                                    */
        SUBADR2, /**< 0x1C   I2C-bus subaddress 2                                    */
        SUBADR3, /**< 0x1D   I2C-bus subaddress 3                                    */
        ALLCALLADR, /**< 0x1E   All Call I2C-bus address                                */
        STEPCOUNT0, /**< 0x1F   Step counter registers: STEPCOUNT0                      */
        STEPCOUNT1, /**< 0x20   Step counter registers: STEPCOUNT1                      */
        STEPCOUNT2, /**< 0x21   Step counter registers: STEPCOUNT2                      */
        STEPCOUNT3 /**< 0x22   Step counter registers: STEPCOUNT3                     */
    } reg8;

    /* register names for 2 bytes accessing */
    typedef enum {
        CWPW = CWPWL | 0x80, /**< Step pulse width for CW rotation */
        CCWPW = CCWPWL | 0x80, /**< Step pulse width for CCW rotation */
        CWSCOUNT = CWSCOUNTL | 0x80, /**< Number of steps CW */
        CCWSCOUNT = CCWSCOUNTL | 0x80, /**< Number of steps CCW */
    } reg16;

        /* register names for 4 bytes accessing */
    typedef enum {
        STEPCOUNT = STEPCOUNT0 | 0x80, /**< Step pulse width for CW rotation */
    } reg32;

    /* prescaler range setting */
    typedef enum {
        PRESCALER_FROM_40_TO_333333,    /*< Prescaler range from   3us(333333pps) to   24.5
76ms(40   pps) */
        PRESCALER_FROM_20_TO_166667,    /*< Prescaler range from   6us(166667pps) to   49.1
52ms(20   pps) */
        PRESCALER_FROM_10_TO_83333,     /*< Prescaler range from  12us( 83333pps) to   98.3
04ms(10   pps) */
        PRESCALER_FROM_5_TO_41667,      /*< Prescaler range from  24us( 41667pps) to  196.6
08ms( 5   pps) */
        PRESCALER_FROM_2_5_TO_20833,    /*< Prescaler range from  48us( 20833pps) to  393.2
16ms( 2.5 pps) */
        PRESCALER_FROM_1_27_TO_10416,   /*< Prescaler range from  96us( 10416pps) to  786.4
32ms( 1.27pps) */
        PRESCALER_FROM_0_64_TO_5208,    /*< Prescaler range from 192us(  5208pps) to 1572.8
64ms( 0.64pps) */
        PRESCALER_FROM_0_32_TO_2604,    /*< Prescaler range from 384us(  2604pps) to 3145.7
28ms( 0.32pps) */
    } prescaler_range;

    uint8_t m_addr;
    Bcm2835Wrapper& m_i2c;
    bool m_isRunning;

    void initRegs();
    void write(reg8 reg, uint8_t val);
    void write(reg16 reg, uint16_t val);
    uint8_t read(reg8 reg);
```

```
    uint16_t read(reg16 reg);
    uint32_t read(reg32 reg);
};

#endif // PCA9629A_H
```

Les méthodes privées « **read()** » et « **write()** » **redéfinissent** les méthodes de la classe **Bcm2835Wrapper** (en utilisant les méthodes de cette dernière) pour être **adaptées** au **PCA9629A**, puis utilisées dans les autres méthodes de la classe. Voici leurs implémentations :

```
void PCA9629A::write(reg8 reg, uint8_t val) {
    uint8_t cmd[ 2 ];

    cmd[ 0 ] = reg;
    cmd[ 1 ] = val;

    m_i2c.write(m_addr, cmd, 2);
    // i2cset -y 1 0x20 0xe 0x12
    qDebug() << "i2cset -
y 1 0x" << hex << (int)m_addr << " 0x"  << hex << (int)cmd[0] << " 0x" << hex << (int)cmd[1] << endl;
}

void PCA9629A::write(reg16 reg, uint16_t val) {
    uint8_t cmd[ 3 ];

    cmd[ 0 ] = reg;
    cmd[ 1 ] = val & 0x00ff;
    cmd[ 2 ] = val >> 8;

    m_i2c.write(m_addr, cmd, 3);

    //  i2cset -y 1 0x20 0x1a 0x1234 w => ecrit 0x20|W 0x1A 0x34 0x12
    qDebug() << "i2cset -
y 1 0x" << hex << (int)m_addr << " 0x"  << (int)cmd[0] << " 0x" << hex << (int)cmd[1] << hex << (int)cm
d[2] << " w" << endl;
}

uint8_t PCA9629A::read(reg8 reg) {
    uint8_t val;
    uint8_t cmd = (uint8_t)reg;

    // Sélectionner le registre à lire
    m_i2c.write(m_addr, &cmd, 1);

    // Lire le registre
    m_i2c.read(m_addr, &val, 1);

    return val;
}

uint16_t PCA9629A::read(reg16 reg) {
    uint8_t val[ 2 ];
    uint8_t cmd = (uint8_t)reg;

    // Sélectionner le registre à lire
    m_i2c.write(m_addr, &cmd, 1);

    // Lire le registre
    m_i2c.read(m_addr, val, 2);

    return *((uint16_t *)val);
}

uint32_t PCA9629A::read(reg32 reg) {
    uint8_t val[ 4 ];
    uint8_t cmd = (uint8_t)reg;

    // Sélectionner le registre à lire
    m_i2c.write(m_addr, &cmd, 1);
```

```
    // Lire le registre
    m_i2c.read(m_addr, val, 4);

    return *((uint32_t *)val);
}
```

Les signatures des méthodes **write** : `write(reg16 reg, uint16_t val)` permet d'avoir un accès simple au contrôle du PCA9629A. En effet, les arguments sont le registre dans lequel on souhaite écrire, et la valeur à écrire dans celui-ci. Ces méthodes sont définies plusieurs fois car comme on l'a vu précédemment, il y a certains registres dans lesquels on peut écrire sur **8 bits (1 octet),** et d'autre sur **16 bits (2 octets)**. Il en est de même pour les méthodes **read**, mais à la seule différence qu'il y a un registre lisible sur **32 bits** permettant de connaitre le nombre total de pas effectué par le moteur depuis le premier lancement sans réinitialisation.

À l'intérieur de ces méthodes, on utilise **l'attribut m_i2c** pour accéder aux méthodes de la classe **Bcm2835Wrapper**. Cet attribut est instancié grâce à la méthode **getInstance()** (vu précédemment) dans le constructeur :

```
PCA9629A::PCA9629A(uint8_t addr) : m_addr(addr), m_i2c(Bcm2835Wrapper::getInstance()), m_is
Running(false) {
    Reset();
    initRegs();
}
```

Ensuite les méthodes **read()** et **write()** sont réutilisées dans les autres méthodes. Voici par exemple la méthode **setRamp** :

```
void PCA9629A::setRamp(ramp ramp, uint8_t slope) {
    uint8_t cmd = 0x20;
    cmd |= slope;
    write((ramp == UP) ? RUCNTL : RDCNTL, cmd);
}
```

Implémentation de la méthode permettant de définir les rampes d'accélération ou de décélération pour le moteur.

Voici maintenant l'implémentations des méthodes essentielles :

- **setStep()** : Cette méthode permet de définir le nombre de pas que doit parcourir le moteur :

  ```
  void PCA9629A::setStep(dir dir, int stepCount) {
      write((dir == CW) ? CWSCOUNT : CCWSCOUNT, stepCount);
  }
  ```

  Le premier argument que prend cette méthode est une valeur de l'énumération **dir** permettant de **définir la direction de rotation**. Ensuite, on teste cette valeur pour choisir dans quel registre on définit le nombre de pas car il y en a un pour le sens horaire et un autre pour le sens antihoraire (ceci a déjà été expliqué dans une partie précédente). Le deuxième argument et simplement le nombre de pas sous forme d'entier.

- **setSpeed()** : Cette méthode permet de définir la vitesse de rotation du moteur :

  ```
  uint16_t PCA9629A::setSpeed(dir dir, int pulsePerSecond) {
      uint8_t prescaler = 0;
      uint8_t ratio;

      ratio = (1.0/24.576e-3) / pulsePerSecond;
  ```

```
    prescaler = (ratio & 0x01) ? 1 : prescaler;
    prescaler = (ratio & 0x02) ? 2 : prescaler;
    prescaler = (ratio & 0x04) ? 3 : prescaler;
    prescaler = (ratio & 0x08) ? 4 : prescaler;
    prescaler = (ratio & 0x10) ? 5 : prescaler;
    prescaler = (ratio & 0x20) ? 6 : prescaler;
    prescaler = (ratio & 0x40) ? 7 : prescaler;

    uint16_t stepPulseWidth;

    stepPulseWidth = STEP_RESOLUTION / ((1 << prescaler) * pulsePrSecond);

    if (stepPulseWidth & 0xE000) { //error( "pps setting: out of range" );
        stepPulseWidth = 0x1FFF;
        printf("the pps forced in to the range that user specified.. %fpps\r\n", (float) STEP_
RESOLUTION / ((float) 0x1FFF * (float) (1 << prescaler)));
    } //if
    if (!stepPulseWidth) { //error( "pps setting: out of range" );
        stepPulseWidth = 0x1;
        printf("the pps forced in to the range that user specified.. %fpps\r\n", (float) STEP_
RESOLUTION / (float) (1 << prescaler));
    } //if

    stepPulseWidth |= (prescaler << 13);
    write((dir == CW) ? CWPW : CCWPW, stepPulseWidth);

    return ( stepPulseWidth);
}
```

La vitesse de rotation est calculée grâce aux formules fournis dans la documentation.

- **isRuning()** : Cette méthode retourne une valeur boolean selon l'état du moteur : **true si actif,** et **false si non actif** :

```
bool PCA9629A::isRunning() {
    uint8_t mcntl = read(MCNTL);

    return (mcntl & 0x80) ? true : false;
}
```

Cette méthode va aller lire dans le registre **MCNTL** (qui ici est définit grâce à une énumération) du PCA9629A qui a été vu dans la partie prototypage.
Voici un rappel de la documentation :

when both bits [7:6] of this register are set to 1. The type of operation control in bit [1:0] is not allowed to change when re-starting motor.

Table 30. MCNTL - Motor control register (address 1Ah) bit description
*Legend: * default value.*

Address	Register	Bit	Access	Value	Description
1Ah	MCNTL	7	R/W	1	start motor
				0*	stop motor
		6	R/W	1	re-start motor for new speed and operation
				0*	self clear after new speed starts running
		5	R/W	1	emergency stop motor
				0*	self clear after motor stop and bit 7 also clears to 0
		4	W only	1	enable START (bit 7) ignore caused by P0 state
				0*	disable START (bit 7) ignore caused by P0 state
		3	W only		P0 polarity setting for START (bit 7) ignore
				1	set P0 input state is HIGH to ignore START bit 7
				0*	set P0 input state is LOW to ignore START bit 7
		2	R only	0*	reserved
		1:0	R/W	11	rotate counter-clockwise first, then clockwise
				10	rotate clockwise first, then counter-clockwise
				01	rotate counter-clockwise
				00*	rotate clockwise

8.21.1 MCNTL[7]: start/stop motor

Ce registre permet de contrôler l'état du moteur (le démarrer dans une direction ou dans l'autre, l'arrêter…). On vient donc lire ce registre avec la méthode « **read** » (de la classe **PCA9629A** qui est implémenter grâce à la classe **Bcm2835Wrapper**). Et on stock le résultat dans une variable « **mcntl** » qui correspond au nom du registre :

```
uint8_t mcntl = read(MCNTL);
```

Puis, on effectue un **ET bit à bit** avec la valeur hexadécimale **0x80** pour savoir si le moteur tourne ou non.
En effet, selon la documentation ci-dessus, le bit numéro 7 (8ème car ça commence à 0) permet de commander l'état du moteur. Si on isole ce bit en binaire, on obtient : **0b10000000** soit **0x80.** Puis on retourne le résultat sous forme d'un boolean (**false** si le **8ème bite** est à **0**, et **true** s'il est à **1**).

```
return (mcntl & 0x80) ? true : false;
```

On utilise ici la **syntaxe ternaire** pour faire toute l'opération proprement en une ligne (cela évite d'écrire un **if(){}**).

- **start()** : Cette méthode permet de démarrer la rotation du moteur dans un sens ou dans l'autre :

```
void PCA9629A::start(dir dir) {
    write(MCNTL, 0x80 | dir);
}
```

Comme précédemment, on écrit dans le registre **MCNTL** permettant de contrôler le moteur. L'énumération **dir** a pour valeur **CW** (**clockwise**) qui vaut **0** et **CCW** (counter-clockwise) qui vaut 1. On fait ensuite un **OU logique** la valeur **0x80** permettant de **passer à 1** le bit de démarrage du moteur, et la valeur de de l'énumération (0 ou 1) permettant de contrôler le **premier bit** qui définit la direction de la rotation :

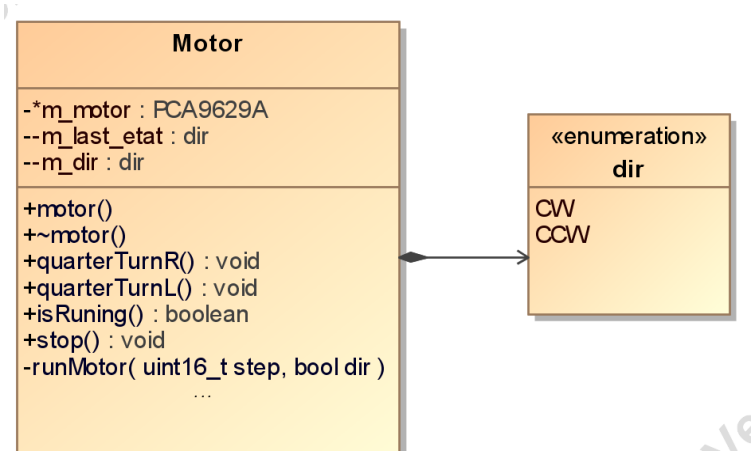
2	R only	0*	reserved
1:0	R/W	11	rotate counter-clockwise first, then clockwise
		10	rotate clockwise first, then counter-clockwise
		01	rotate counter-clockwise
		00*	rotate clockwise

La dernière étape est de créer une classe permettant de faire tourner le moteur d'un quart de tour en utilisant des méthodes simples.

Réalisation de la classe Motor :

L'objectif de cette classe est de simplifier l'accès à la classe PCA9629A vu précédemment en permettant de faire simplement tourner le moteur d'un quart de tours à droite ou à gauche.

Voici le diagramme de classe :



Le code de cette classe :

Fichier motor.h :

```
#ifndef MOTOR_H
#define MOTOR_H

#include "pca9629a.h"

/*
 * Classe Motor
 * Gère la rotation du moteur pour faire un quart de tour
 * Utilise la classe PCA9629A
 */

class Motor
{
public:
    Motor();
    ~Motor();

    void quaterTurnL();
    void quaterTurnR();
    void stop();
    bool isRuning();
    void reset();

private:
    typedef enum {
        CW,
        CCW
    } dir;

    PCA9629A *m_motor;
    dir m_last_etat;
    dir m_dir;
    uint32_t m_stepNumber;
    void runMotor(uint16_t step = 0x64, bool dir = false);
};

#endif // MOTOR_H
```

Pour accéder à la classe **PCA9629A**, on utilise un **attribut** nommé **m_motor** instancié dans le constructeur dont voici l'implémentation :

```
Motor::Motor()
{
    //instensiation de la classe PCA9629A
    m_motor = new PCA9629A;
    //Définition des ramps
    //m_motor->setRamp(PCA9629A::UP, 0x0);
    //m_motor->setRamp(PCA9629A::DOWN, 0x0);
    //Définition de la vitesse
    m_motor->setSpeed(PCA9629A::CW, 0xFF);
    m_motor->setSpeed(PCA9629A::CCW, 0xFF);
    m_stepNumber = 0;
}
```

À la suite de ça, on définit aussi la vitesse de rotation du moteur et on initialise le nombre de pas à parcourir à 0.

Il y a donc deux méthodes pour faire pour faire tourner le moteur soit à droite, soit à gauche, d'un quart de tours :

```
void quaterTurnL();
void quaterTurnR();
```

La seule différence est la direction. Pour **éviter de répéter la logique** permettant de faire tourner le moteur, on utilise une **méthode intermédiaire** :

```
void runMotor(uint16_t step = 0x64, bool dir = false);
```

Cette méthode est privée car elle ne sert que pour le fonctionnement interne de la classe.

Elle prend pour paramètres le nombre de pas à effectuer qui par défaut vaut **0x64** qui correspond à 100 pas (Pour un **moteur 400 pas** : **400 / 4 = 100** pour un quart de tours).

Le deuxième paramètre permet de définir la direction de la rotation par une valeur boolean.

Voici l'implémentation de cette méthode :

```
void Motor::runMotor(uint16_t step, bool dir)
{
    if (!dir){
        m_motor->setStep(PCA9629A::CCW, step);
        m_motor->start(PCA9629A::CCW);
    } else {
        m_motor->setStep(PCA9629A::CW, step);
        m_motor->start(PCA9629A::CW);
    } // else

    //Boucle infinie -> tant que le moteur tourne
    while(m_motor->isRunning()) {
        sleep(1);
    } //while
}
```

On teste la direction pour définir le nombre de pas et faire tourner le moteur dans un sens ou dans l'autre.

Ensuite, on utilise une **boucle while** qui s'exécutera **tant que le moteur tourne**. Cela sera utile pour la **partie intégration** en **bloquant le reste du code** pendant que le moteur tourne, notamment pour la **synchronisation plateau tournant et caméra**.

Cette méthode est ensuite appelée dans les méthodes **quaterTurnL()**, et **quaterTurnR()** :

```
void Motor::quaterTurnL()
{
    m_dir = CCW;
    runMotor(0x1C4, m_dir);
    m_last_etat = CCW; //variable d'etat pour le reset
}

void Motor::quaterTurnR()
{
    m_dir = CW;
    runMotor(0x1C4, m_dir);
    m_last_etat = CW; //variable d'etat pour le reset
}
```

À noter qu'on fournit pour le nombre de pas à effectuer, la valeur **0x1C4,** soit **452** en décimal qui correspond au nombre de pas que doit faire le moteur pour faire faire un quart de tour au plateau tournant lorsque celui-ci est monté avec le moteur.

Cette valeur a été déterminée expérimentalement comme la plus optimale.

Pour ramener le moteur à sa position initiale, on utilise la méthode **reset()** dont voici le code:

```
void Motor::reset()
{
    //Récupérer le nombre de pas parcouru
    m_stepNumber = m_motor->getStepNumber();

    if (m_stepNumber > 0) { //si le moteur a tourné
        //Quel est le dernier sens dans lequel le moteur a tourné
        if (m_last_etat == CCW) {
            runMotor(m_stepNumber, CW);
        } else {
            runMotor(m_stepNumber, CCW);
        } //else
    } //if
}
```

On commence par lire le nombre de pas qui a été parcouru (autant de fois que l'on a appelé les méthodes **quaterTurnL** et **quaterTurnerR**). On teste ensuite que cette valeur n'est pas nul (si non, c'est que le moteur n'a pas tourné). Si le test est vrai, on regarde qu'elle est la dernière direction dans laquelle le moteur a tourné pour pouvoir revenir dans le sens inverse avec le nombre de pas qui a été parcouru.

Pour finir, la méthode **isRuning()** permet de retourner l'état du moteur. On utilise la méthode **isRuning** de la classe **PCA9629A** pour implémenter celle-ci :

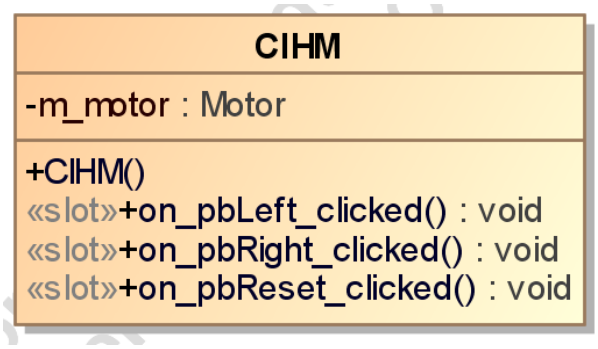
```
bool Motor::isRuning()
{
    return m_motor->isRunning();
}
```

La dernière étape est d'intégrer le moteur avec l'IHM réalisée.

Réalisation de la classe CIHM :

Cette classe a pour objectif de gérer l'IHM.

Voici le diagramme de classe :



Fichier cihm.h :

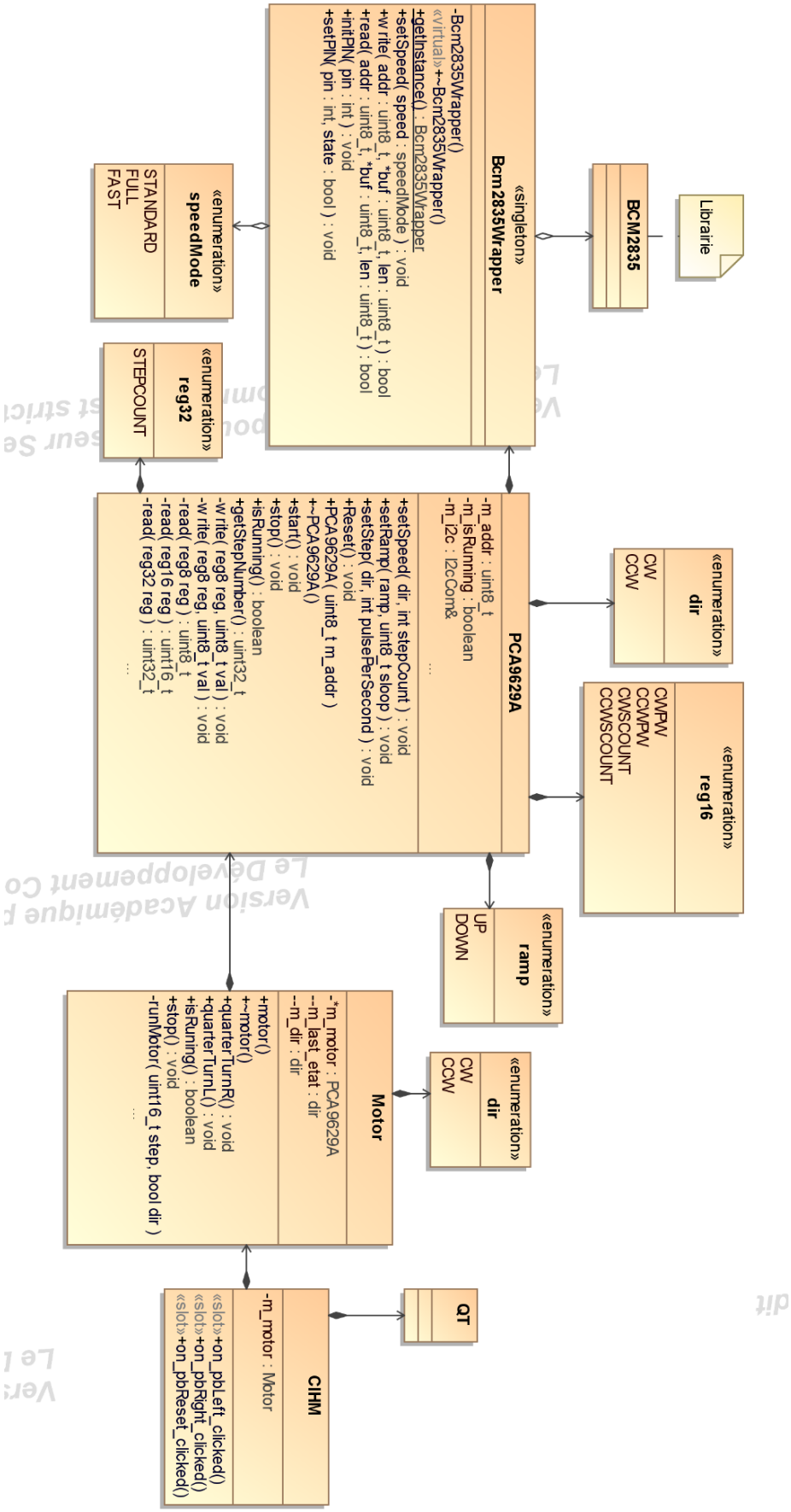
```
#ifndef CIHM_H
#define CIHM_H

#include "motor.h"

#include <QMainWindow>

QT_BEGIN_NAMESPACE
namespace Ui { class CIHM; }
QT_END_NAMESPACE

class CIHM : public QMainWindow
{
    Q_OBJECT

public:
    CIHM(QWidget *parent = nullptr);
    ~CIHM();

private slots:
    void on_pbLeft_clicked();
    void on_pbRight_clicked();
    void on_pbReset_clicked();

private:
    Ui::CIHM *ui;
    motor* m_motor;
};
#endif // CIHM_H
```

On utilise le **système de slots** de Qt permettant de **réceptionner des signaux** (un bouton appuyer par exemple).

Ces **slots** ont été **générés automatiquement** via l'interface de **Qt-Designer**. Leurs implémentations sont plutôt simples :

```
void CIHM::on_pbLeft_clicked()
{
    m_motor->quaterTurnL();
}

void CIHM::on_pbRight_clicked()
{
    m_motor->quaterTurnR();
}

void CIHM::on_pbReset_clicked()
{
    m_motor->reset();
}
```

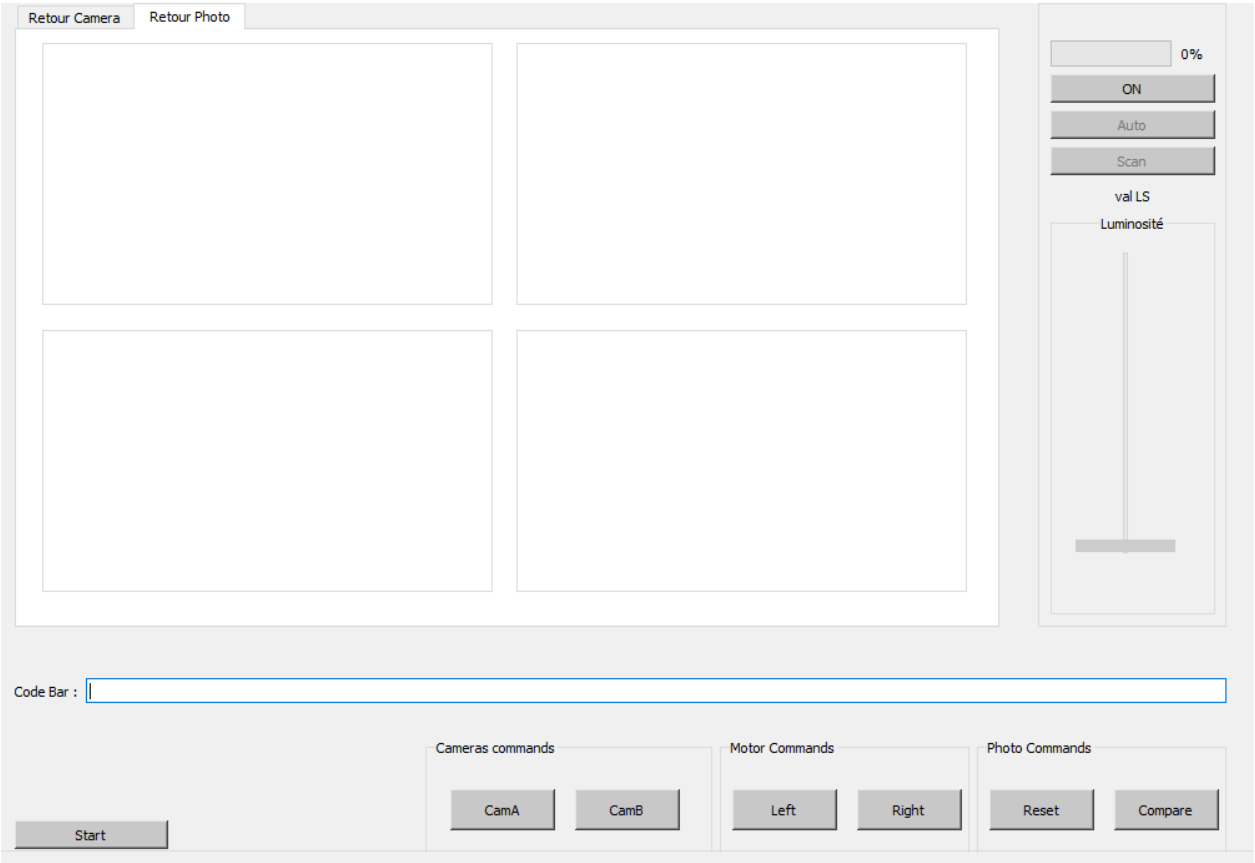
Diagramme de classe complet :



Développement : Conception de l'IHM :

Le but de cette partie et de pouvoir piloter le moteur sur l'application finale.

Voici le prototype de l'IHM qui a été réalisé par l'ensemble de l'équipe pour l'application finale :



L'IHM doit permettre :

- D'ajuster l'éclairage manuellement ou automatiquement.
- Déplacer le moteur manuellement ou automatiquement.
- Lancer le processus de prises de vues.
- Réinitialiser le processus de prises de vues, si les clichés ne conviennent pas.
- Avoir un retour direct de la caméra pour pouvoir effectuer les différents réglages nécessaires.
- Afficher les 4 prises de vues effectuées.
- Lancer le processus de comparaison (Fenêtre pop-up permettant d'afficher les images prises avec leurs versions antérieures, et saisir une note en cas de différences constatées).
- Saisir et afficher le code-barre d'un produit.

Développement : Intégration de la partie plateau tournant :

Pour rappel, le moteur doit pouvoir être piloter de deux manières différentes :

- Manuellement via les boutons présents sur l'IHM pour faire un quart de tours à droite ou à gauche.

- Automatiquement en enclenchant le processus de prise de vues via le bouton start

Pour ce faire, le code de la partie prototypage a été repris. C'est-à-dire les classes :

- **Bcm2835Wrapper** : Classe dite « singleton » permettant d'assurer la communication I2C par l'intermédiaire de la librairie BCM2835 permettant d'interagir avec les GPIO de la RPI.

- **PCA9629A** : Classe permettant de contrôler le PCA9629A, le contrôleur du moteur pas à pas, via une communication I2C en utilisant la classe **Bcm2835Wrapper**.

- **Motor** : Classe simplifiant l'accès à la classe PCA9629A. Cette classe permet de faire tourner le moteur simplement d'un quart de tours à droite ou à gauche.

Remarque : par rapport à la partie prototypage, la **classe Motor** a été modifiée en enlevant la méthode « reset » qui avait pour fonction de réinitialiser le moteur.

Ici, cette méthode n'est pas nécessaire étant donné que le plateau n'a pas de fin de course (n'est pas bloqué au bout d'un tour). De plus, durant le processus de prise de vue, le plateau fait un tour complet et revient donc à sa position initiale de lui-même.

Voici le diagramme des classes complet :

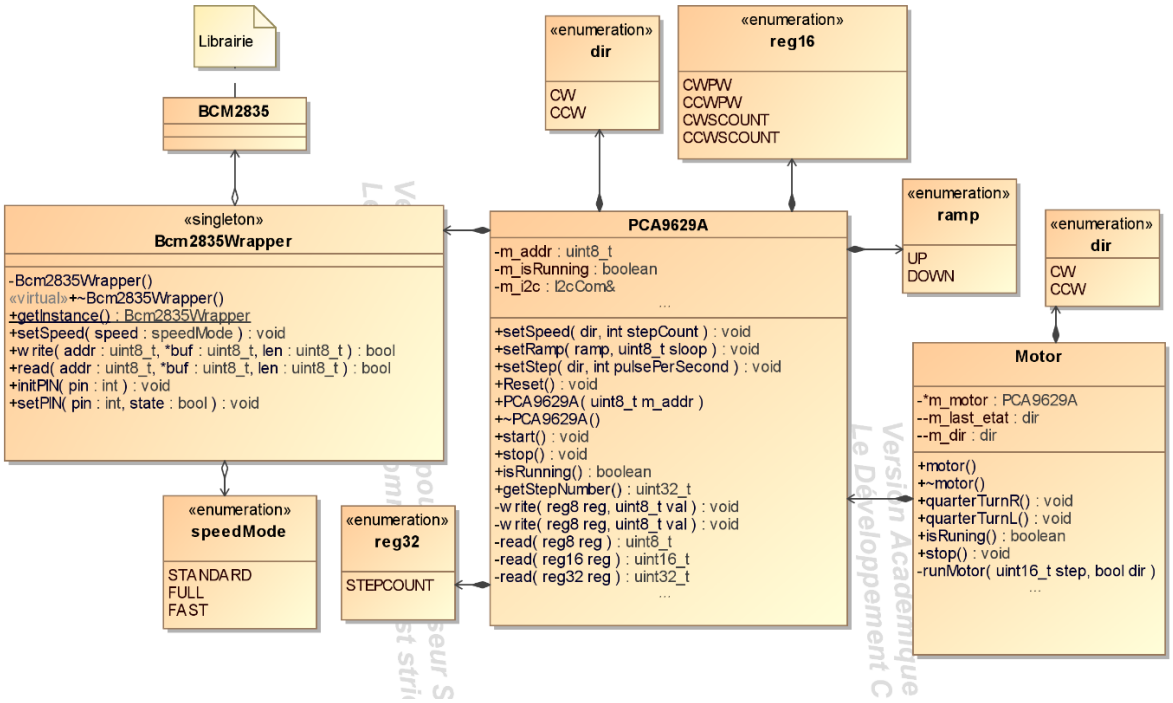
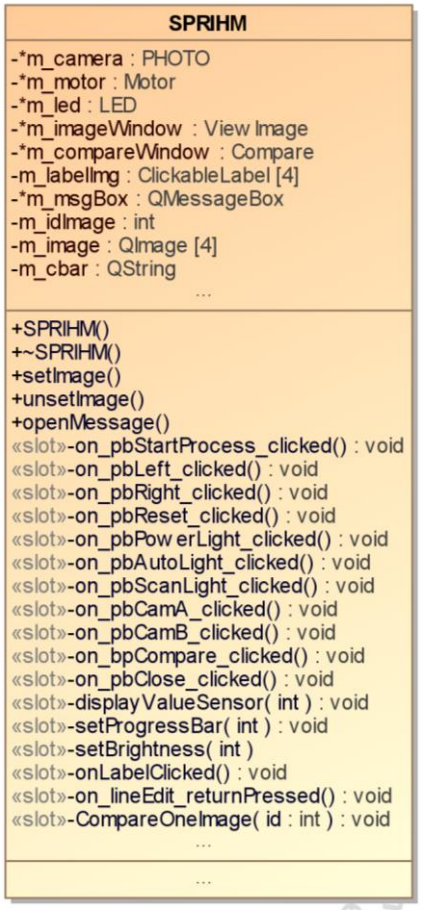


Diagramme de classe de la partie moteur avec quelques énumérations retirées pour simplifier.

Voici le diagramme de classe de l'IHM :



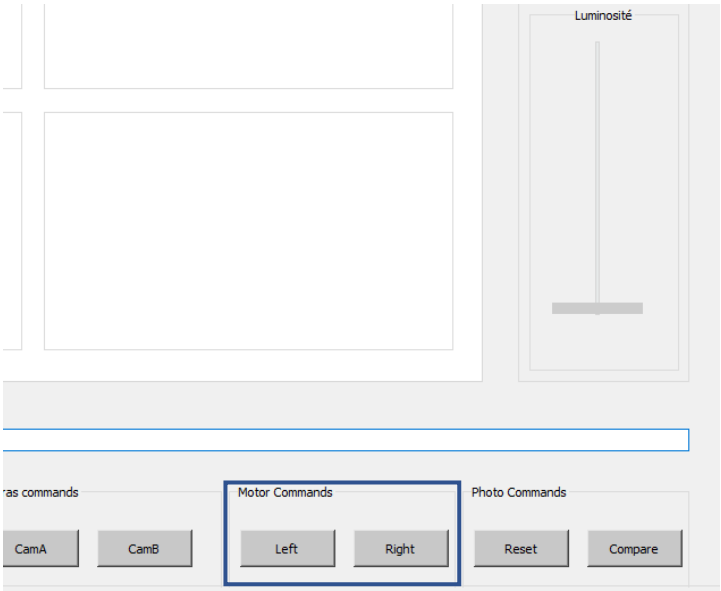
Fichier SPRIHM.h :

```
#ifndef SPRIHM_H
#define SPRIHM_H

#include <QMainWindow>
#include <QObject>
#include <QWidget>
#include <QPushButton>
#include <QImage>
#include <QLabel>
#include <QMessageBox>
#include <QSlider>

#include "motor.h"
#include "photo.h"
#include "led.h"
#include "clickablelabel.h"
#include "viewimage.h"
#include "compare.h"

QT_BEGIN_NAMESPACE
namespace Ui { class sprIHM; }
QT_END_NAMESPACE

class sprIHM : public QMainWindow
{
    Q_OBJECT
```

```
public:
    sprIHM(QWidget *parent = nullptr);
    ~sprIHM();

    void setImage();
    void unsetImage();
    void openMessage();

private:
    Ui::sprIHM *ui;

    Motor *m_motor;
    PHOTO *m_camera;
    LED *m_led;

    ViewImage *m_imageWindow;
    Compare *m_compareWindow;

    QPushButton *m_bpLeft;
    QPushButton *m_bpRight;

    QSlider *m_slider;

    QImage m_image[4];
    ClickableLabel *m_labelImg[4];

    QMessageBox *m_msgBox;

    QString m_cbar;
    int m_idImage;

private slots:
    void on_pbBack_clicked();
    void on_pbNext_released();

    void on_pbStartProcess_clicked();

    void on_pbPowerLight_clicked();
    void on_pbAutoLight_clicked();
    void on_pbScanLight_clicked();

    void displayValueSensor(int);
    void setProgressBar(int);

    void setBrightness(int);

    void onLabel1Clicked();
    void onLabel2Clicked();
    void onLabel3Clicked();
    void onLabel4Clicked();
    void on_pbReset_clicked();

    void on_lineEdit_returnPressed();
    void on_bpCompare_clicked();

    void CompareOneImage(int);

    void deleteWindow(int);
    void on_pbClose_clicked();
    void on_pbCamA_clicked();
    void on_pbCamB_clicked();
};

#endif // SPRIHM_H
```

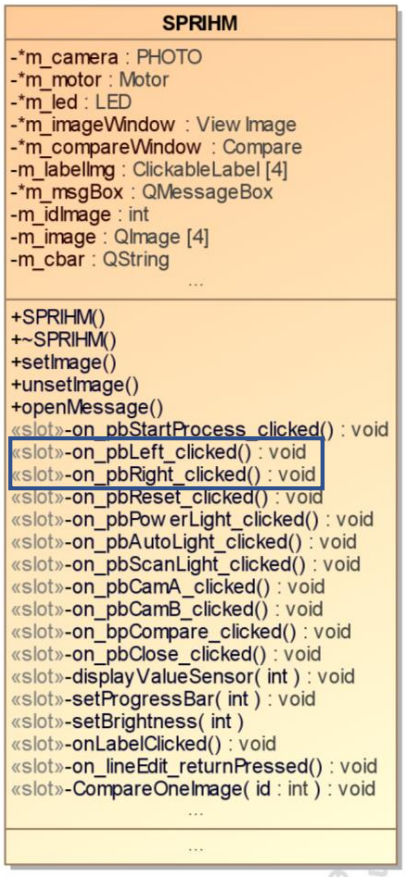
Pilotage manuel du moteur :

C'est tout d'abord le pilotage manuel du moteur qui a été mis en place. Comme dit précédemment, il y'a deux boutons pour piloter le moteur :



Ces deux boutons doivent permettre de faire tourner le moteur d'un quart de tours gauche et à droite.

Pour **connecter les boutons aux méthodes** de la classe **Motor**, on utilise les slots des boutons générés depuis Qt-Designer :



Les slots “**on_pbLeft_clicked()**” et “**on_pbRight_clicked()**” permettent de faire tourner le moteur respectivement à gauche et à droite.

Voici leur implémentation :

```
void sprIHM::on_pbLeft_clicked()
{
    m_motor->quaterTurnL();
}

void sprIHM::on_pbRight_clicked()
{
    m_motor->quaterTurnR();
}
```

Pilotage automatique du moteur :

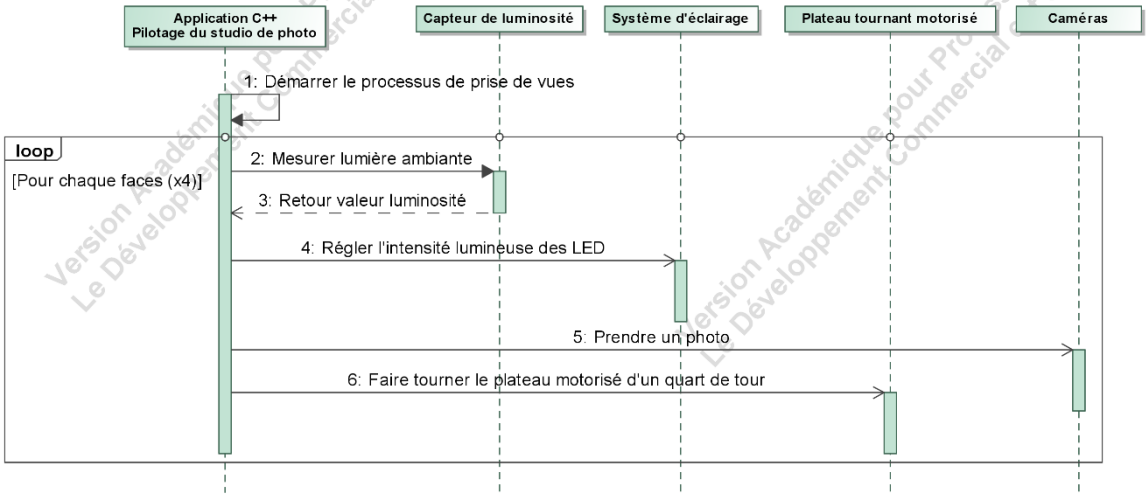
Le moteur doit pouvoir tourner automatiquement lorsque le processus de prises vues est enclenché.

Ce processus comprend :

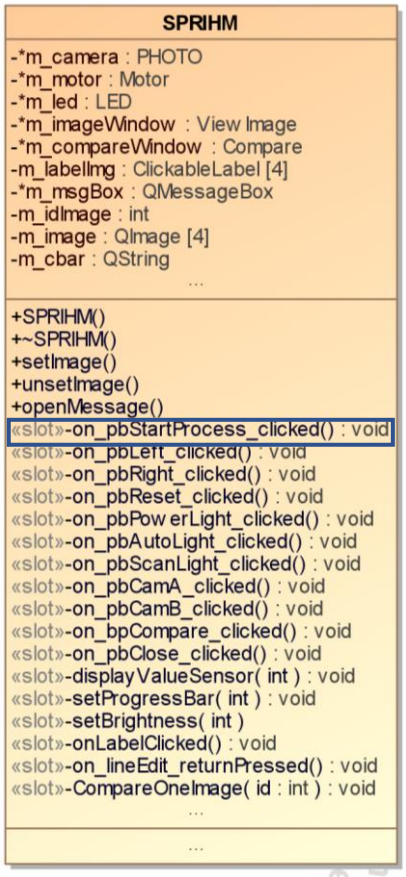
- La mesure de la luminosité ambiante et le réglage de la bande LED en conséquence (Partie gestion de l’éclairage de l’étudiant 3)
- La prise des clichés photos des objets disposés sur le plateau tournant. (Partie gestion de la prise de vue de l’étudiant 1)
- La rotation du plateau tournant pour prendre en photo la face suivante.

Et ainsi de suite jusqu'à avoir pris 4 photos.

Pour résumer le processus ci-dessus, voici un diagramme de séquence :



Comme pour le pilotage automatique, le lancement de ce processus se fait par un slot généré depuis **Qt-Designer** :



Voici l'implémentation de ce slot :

```
void sprIHM::on_pbStartProcess_clicked()
{
    for (int i = 0; i < 4; i++){
        if (!m_motor->isRuning()) {
            m_camera->takeShot(i);
        }
        m_motor->quaterTurnR();

    } //for
    setImage();
    openMessage();
}
```

On commence par une **boucle for** qui a 4 tours à faire, ce nombre correspondant aux 4 photos à prendre d'un produit (les 4 faces).

À l'intérieur, on commence par utiliser la méthode « **isRunning()** » retournant une valeur **boolean** (**true** ou **false**) en fonction de l'état du moteur : actif ou non (le code de cette méthode a été expliqué dans la partie prototypage).

Si le moteur ne tourne pas, on peut prendre une photo avec la méthode :

```
takeShot();
```

Cette méthode est expliquée dans la partie « prise de vue » traitée par l'étudiant 1.

Une fois la photo prise, on fait tourner le moteur d'un quart de tour à droite.

Partie Étudiant 3 DEMETRESCO Mathias (IR3)

Objectifs

- Concevoir l'IHM
- Concevoir le protocole de communication avec EC1 pour le pilotage de l'éclairage
- Mettre en œuvre et piloter l'éclairage
- Récupérer les données du capteur de luminosité
- Concevoir la classe C++ de l'éclairage.
- Intégrer le développement au système final

Le but de cette partie est d'éclairer le mieux possible les produis disposés sur le plateau tournant, pour cela, l'utilisation de LED est nécessaire afin d'obtenir un éclairage optimal pour tous les angles du produis.

Plus la photo sera claire, plus il sera simple de détecter les différences.

La lumière verra son intensité varier, grâce à la fonction **PWM** ***((****Pulse Width Modulation)* ***ou*** *MLI* ***(****Modulation de Largeur d'Impulsion****))*** de la bande de LED.

Il sera possible au client de changer, selon ses besoins, la luminosité de celle-ci.

Pour cela, il faudra utiliser le logiciel commun aux 3 parties : Un bouton nommé « Auto » permettant d'avoir la luminosité optimale (calculée grâce au capteur de lumière) théorique sera disponible, un autre bouton « Manuel » permettant de contrôler un S*lider (objet de l'interface)* manuellement, celui-ci ajustera la luminosité de la bande de LED en fonction de la position du curseur (haut/bas).

Un tissu réfléchissant sera aussi placé de chaque côté de la boite dans le but réfléchir la lumière émise par les LED pour que l'objet ai le moins de point sombre possible.

Tous les équipements seront placés de manière optimale et organisé.

Le matériel :

Pour ce qui est du matériel, nous utiliserons :

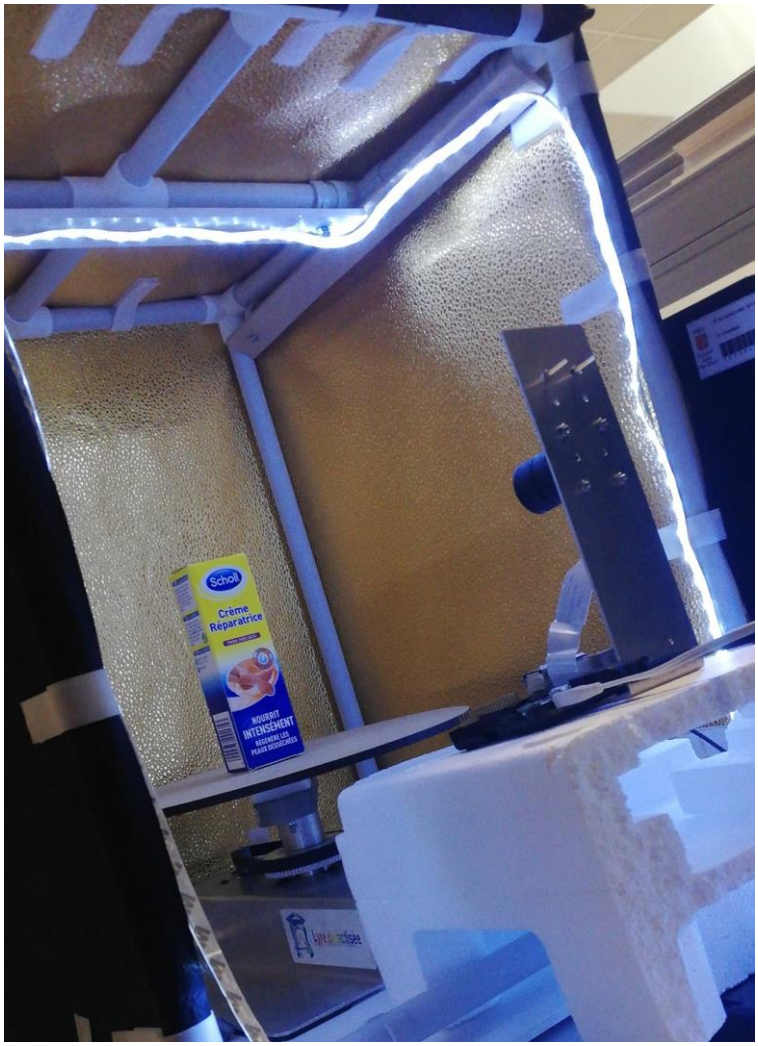
- Une bande de LED blanche pilotée en **PWM** :

- Un tissu réfléchissant doré :

- Un capteur de luminosité VEML7700 ADA4162 fonctionnant en I2C :

- Une Raspberry PI 3 :

Voici le tout en image : (ceci est un montage temporaire, créé pour effectuer nos tests)



Le capteur de luminosité VEML7700 ADA4162 I2C

Module basé sur un capteur VEML7700 permettant de mesurer la luminosité ambiante. Ce capteur est prévu pour une utilisation avec un microcontrôleur type Arduino, Raspberry Pi ou compatible via le bus I2C.

Connectivité : L'utilisation de ce module nécessite la soudure du connecteur 5 broches inclus en fonction de l'utilisation.

Caractéristiques :

- **Alimentation** : 3,3 ou 5 Vcc
- **Interface** : I2C
- **Adresse I2C** : 0x10 (non modifiable)
- **Plage de mesure** : 0 à 120000 lux sur 16 bits
- **Sortie régulatrice** : 3,3 Vcc/100 mA MAX
- **Dimensions** : 17 x 17 x 4 mm
- **Poids** : 1 gramme

L'unité de mesure « Lux (lx) » permet de mesurer l'éclairement :

Un **lux** est l'éclairement d'une surface qui reçoit, d'une manière uniformément répartie, un flux lumineux d'un lumen par mètre carré.

Trame I2C du capteur d'après la documentation :

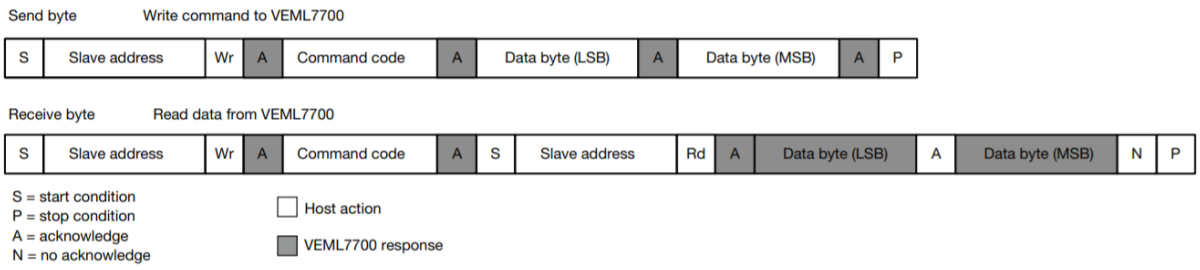


Diagramme des cas d'utilisation :

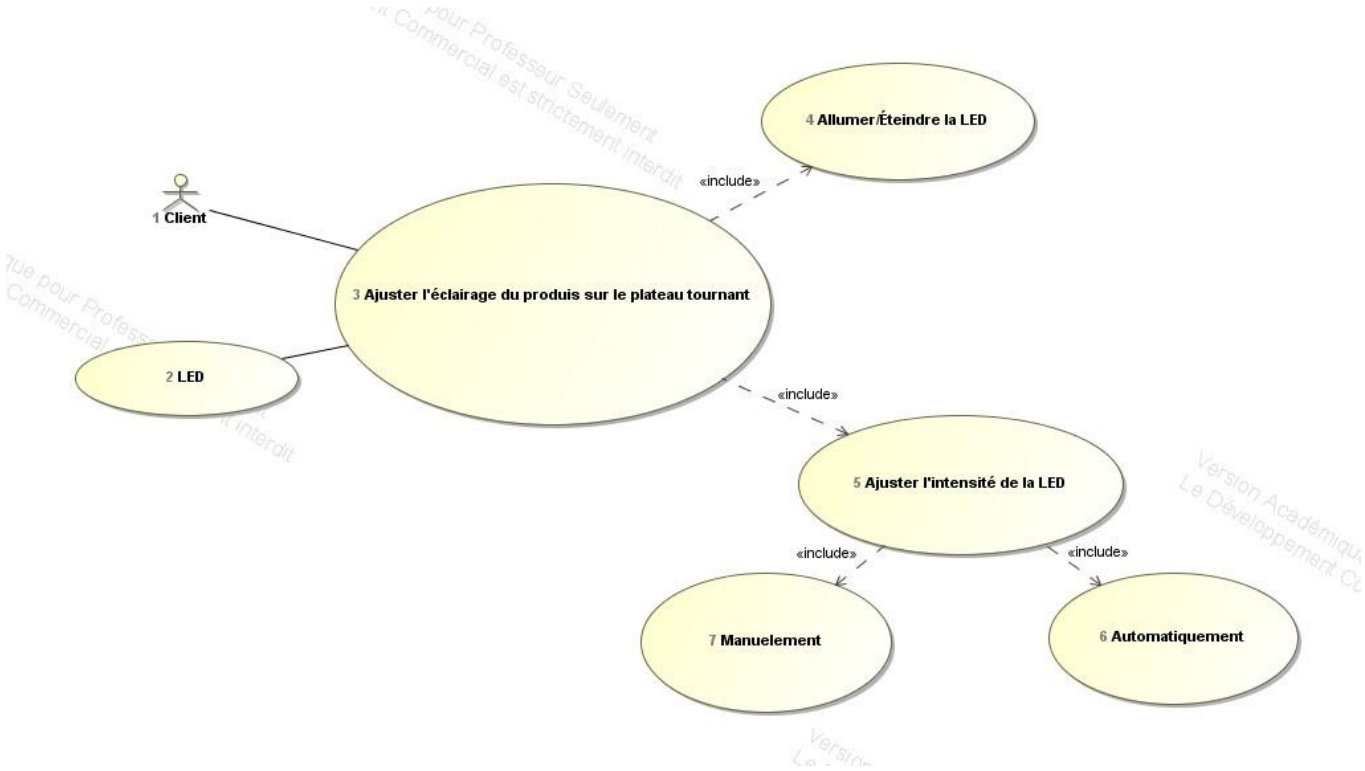
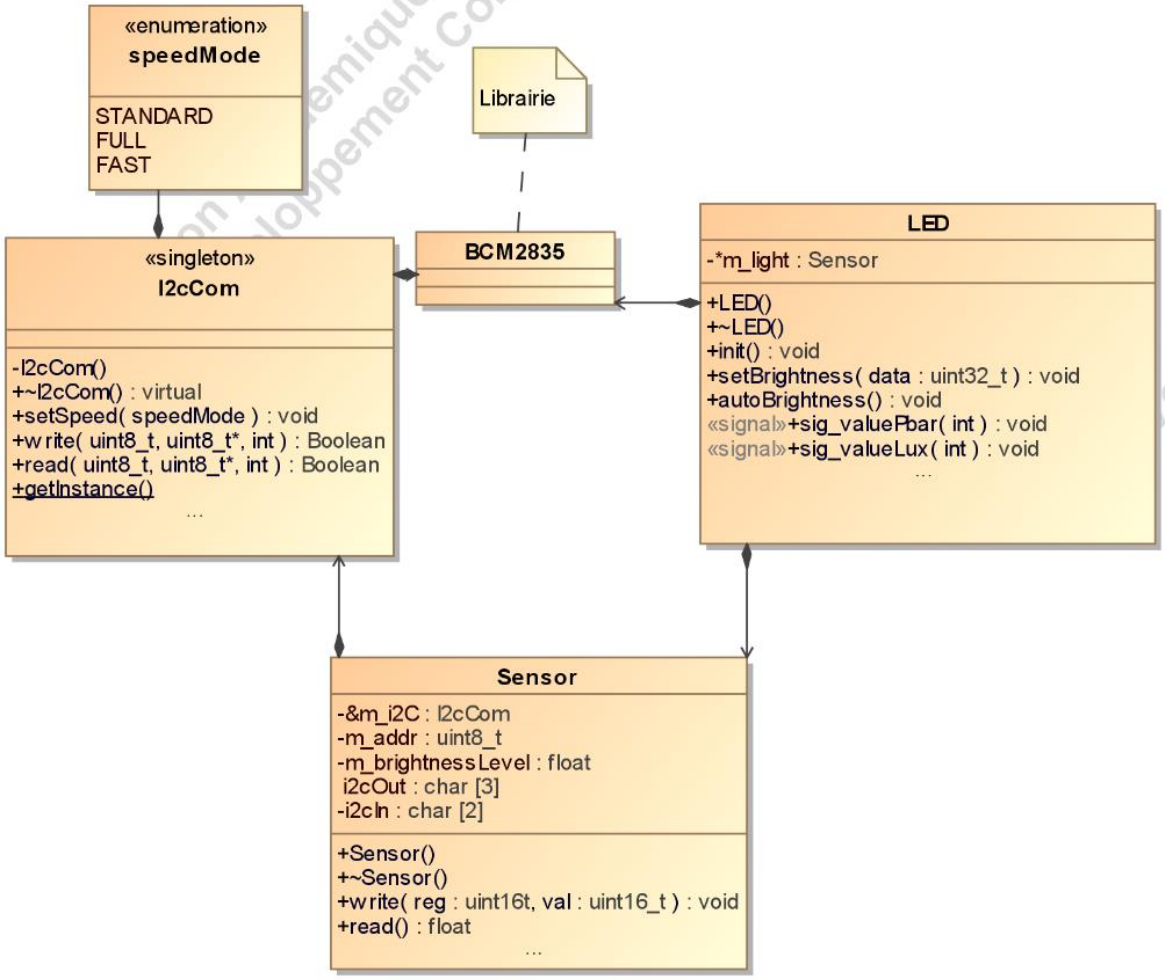


Diagramme de classe de la partie « Éclairage »



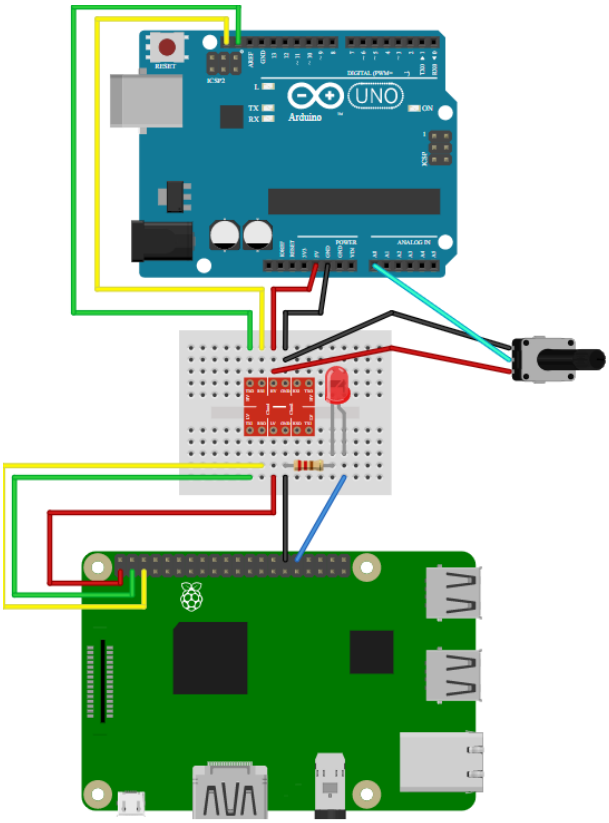
Prototypage :

Dans le but de simuler un changement d'intensité de la bande de LED, l'utilisation d'une LED rouge générique était nécessaire.

Grace à la librairie « **bcm2835** », j'ai pu avoir accès à un programme **d'exemple**, exécutable via la Raspberry.

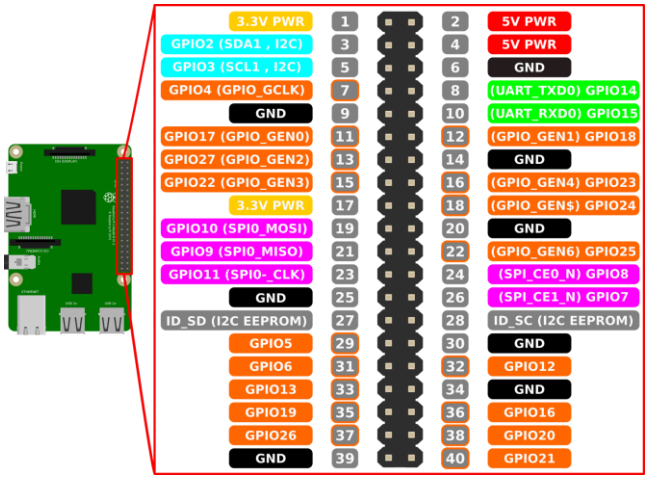
Avant d'exécuter ce programme, il faut bien évidemment **câbler** la Raspberry avec la LED rouge, voici le **schéma** créé :

Dans ce schéma, j'utilise :



- Une **Arduino Uno** dans le but d'utilisé une extension contenant un capteur de luminosité (nous le verrons plus tard).
- Un **Level Shifter I2C** est aussi nécessaire afin de convertir les signaux **3,3V** de la Raspberry pour communiquer avec l'Arduino qui elle fonctionne en **5V.**
- Une **LED générique** et une **résistance 220 ohm,** pour des raisons sécurité.
- Un **Potentiomètre**, qui, après son initialisation dans le programme, nous permettra de faire varier l'intensité de la LED manuellement.
- Une **Breadboard**, permettant de connecter les composants de maniéré non définitive

La LED est branchée sur le **pin 32** de la RPI, celui-ci correspond au **GPIO 12** (PWM compatible).



Analyse des fonctions :

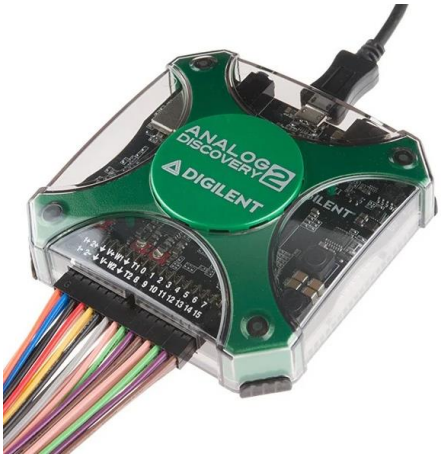
Voici le programme en question, il permet de faire varier l'intensité de la LED automatiquement les **fonctions** sont commentées (lignes vertes) :

```
#include <bcm2835.h>
#include <stdio.h>

// PWM output on RPi Plug P1 pin 12 (which is GPIO pin 18)
// in alt fun 5.
// Note that this is the _only_ PWM pin available on the RPi IO headers
#define PIN RPI_GPIO_P1_12
// and it is controlled by PWM channel 0
#define PWM_CHANNEL 0
// This controls the max range of the PWM signal
#define RANGE 1024

int main(int argc, char **argv)
{
    if (!bcm2835_init())
        return 1;

    // Set the output pin to Alt Fun 5, to allow PWM channel 0 to be output th
ere
    bcm2835_gpio_fsel(PIN, BCM2835_GPIO_FSEL_ALT5);

    // Clock divider is set to 16.
    // With a divider of 16 and a RANGE of 1024, in MARKSPACE mode,
    // the pulse repetition frequency will be
    // 1.2MHz/1024 = 1171.875Hz, suitable for driving a DC motor with PWM
    bcm2835_pwm_set_clock(BCM2835_PWM_CLOCK_DIVIDER_16);
    bcm2835_pwm_set_mode(PWM_CHANNEL, 1, 1);
    bcm2835_pwm_set_range(PWM_CHANNEL, RANGE);
```

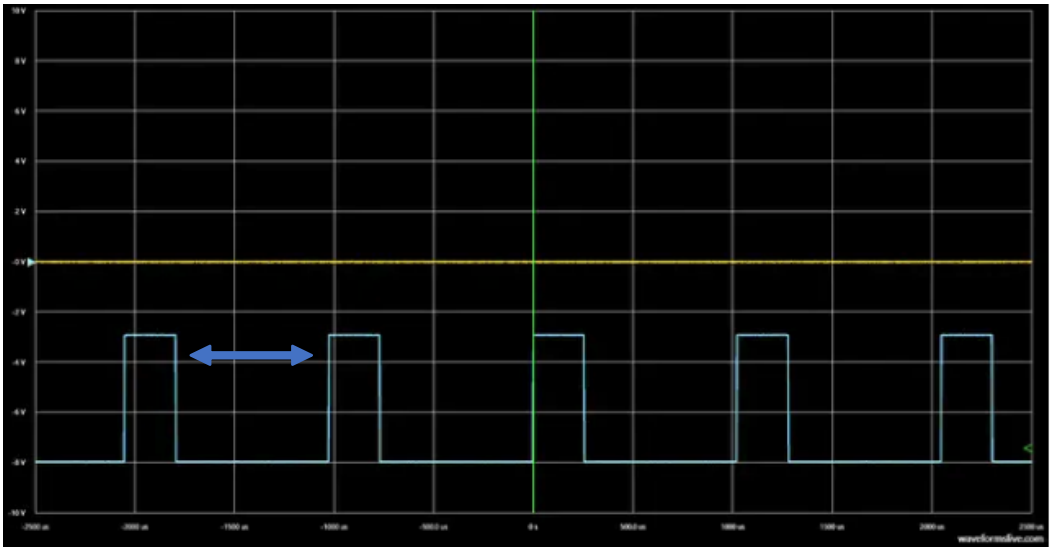
```
    // Vary the PWM m/s ratio between 1/RANGE and (RANGE-1)/RANGE
    // over the course of a few seconds
    int direction = 1; // 1 is increase, -1 is decrease
    int data = 1;
    while (1)
    {
        if (data == 1)
            direction = 1;   // Switch to increasing
        else if (data == RANGE-1)
            direction = -1;  // Switch to decreasing
        data += direction;
        bcm2835_pwm_set_data(PWM_CHANNEL, data);
        bcm2835_delay(1);
    }

    bcm2835_close();
    return 0;
}
```

Analyse du signal émis une fois le programme lancé :

Grace à un **Analogue Discovery 2 :**



Visualisation des signaux **PWM** via leur logiciel **Digilent Waveform** :



Le signal bleu visible montre la fréquence de clignotement de la LED (en Hz). Plus cette fréquence est élevée, plus l'intensité de la LED augmentera, à l'inverse, plus la fréquence diminue, moins l'intensité sera élevée. À une fréquence nulle, la LED s'éteint.

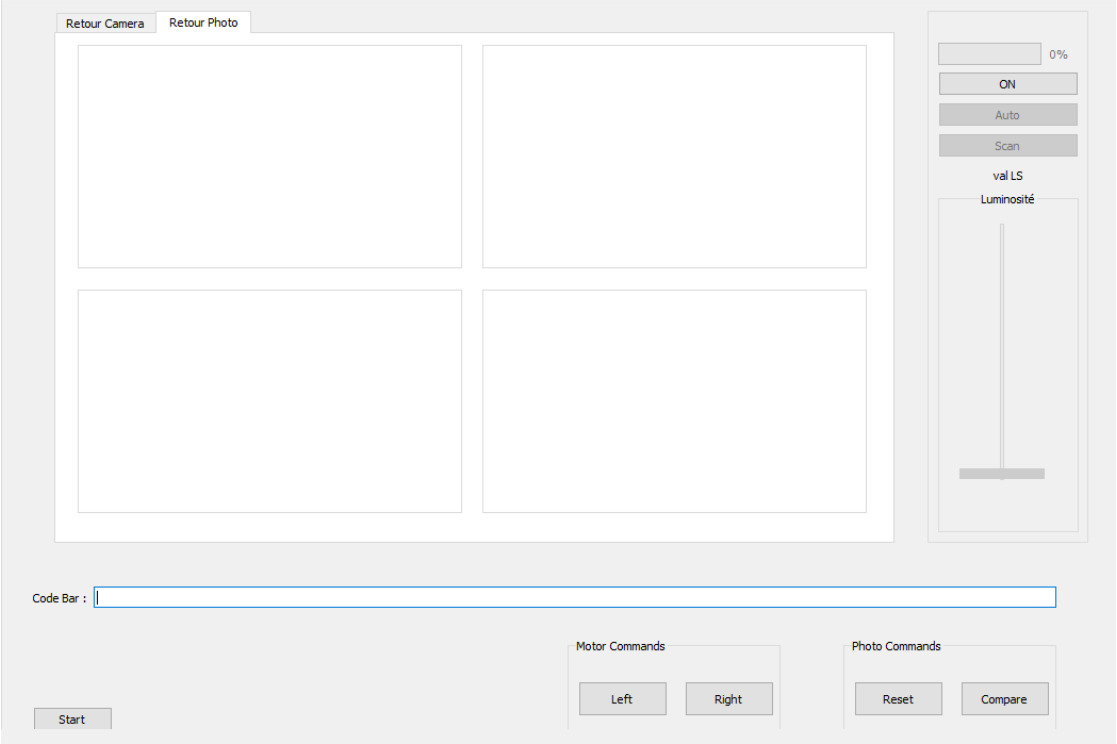
Le programme ci-dessus permet donc de varier cette fréquence. Si l'on visualise le signal pendant que le programme est en cours, le signal bleu verra sa durée varier.

Plus précisément, l'écart entre les signaux rectangles (représenté par la flèche bleu) ci-dessus va varier.

Développement : Conception de l'IHM :

Le but de cette partie et de pouvoir piloter le moteur sur l'application finale.

Voici le prototype de l'IHM qui a été réalisé par l'ensemble de l'équipe pour l'application finale :



L'IHM doit permettre :

- D'ajuster l'éclairage manuellement ou automatiquement.
- Déplacer le moteur manuellement ou automatiquement.
- Lancer le processus de prises de vues.
- Réinitialiser le processus de prises de vues, si les clichés ne conviennent pas.
- Avoir un retour direct de la caméra pour pouvoir effectuer les différents réglages nécessaires.
- Afficher les 4 prises de vues effectuées.
- Lancer le processus de comparaison (Fenêtre pop-up permettant d'afficher les images prises avec leurs versions antérieures, et saisir une note en cas de différences constatées).
- Saisir et afficher le code-barre d'un produit.

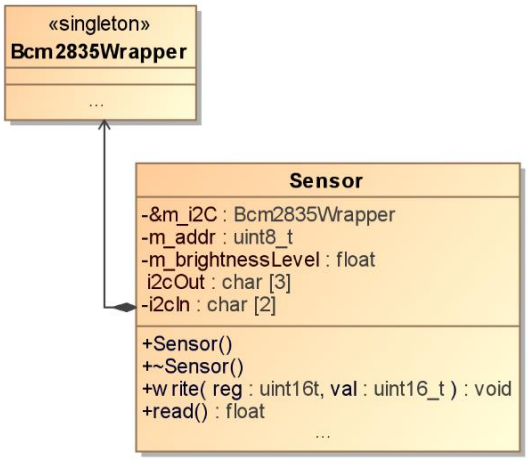
Développement : Conception des classes de la partie éclairage

L'objectif de cette partie est de développer les classes finales qui assureront la gestion de l'éclairage.

Développement de la classe Sensor :

La première étape est de gérer le capteur de luminosité qui nous servira lors du pilotage de la bande LED.

Voici le diagramme de classe :



Cette classe utilise la classe **Bcm2835Wrapper** qui a été développée dans la partie « **gestion du plateau tournant** » de l'étudiant 2.

Pour rappel, c'est un classe singleton permettant d'assurer la **communication** avec les **GPIO** de la **RPI** (et donc assurer la communication I2C) en utilisant la librairie **BCM2835**.

Fichier sensor.h :

```
#ifndef LIGHTSSI2C_H
#define LIGHTSSI2C_H

#include <QObject>
#include "bcm2835wrapper.h"

#define MODE_READ 0
#define MODE_WRITE 1

#define MAX_LEN 32

/*
 * Classe Sensor
 * Gère le capteur de luminosité via une communication I2C
 * Utilise la classe Bcm2835Wrapper (BCM2835)
 */

class Sensor
{
    //Q_OBJECT
public:
    Sensor();
    ~Sensor();
```

```
    //void write(uint8_t reg, uint8_t val);
    void write(uint16_t reg, uint16_t val);

    float read();

private:
    Bcm2835Wrapper &m_i2c;
    uint8_t m_addr;

    float m_brightnessLevel;
    char i2cOut[3];
    char i2cIn[2];

};

#endif // LIGHTSSI2C_H
```

La première étape est de récupérer l'instance de la classe **Bcm2835Wrapper** dans l'attribut **m_i2c**.

Ceci se fait dans l'implémentation du constructeur :

```
Sensor::Sensor() : m_i2c(Bcm2835Wrapper::getInstance())
{
    m_addr = 0x10;
}
```

Comme on le voit, on initialise l'attribut **m_addr** qui correspond à l'adresse de l'esclave **I2C** du **capteur de luminosité**.

D'après la documentation du capteur, son **adresse I2C** est « **0x10** » en Hexadécimal.

La méthode **write()** permet d'écrire dans les **registres I2C** du capteur. Ici, on initialise le capteur. Voici le code de cette méthode :

```
void Sensor::write(uint16_t reg, uint16_t val)
{
    uint8_t tps[3];

    tps[0] = reg;
    tps[1] = val & 0x00ff;
    tps[2] = val  >> 8;

    m_i2c.write(m_addr, tps, 3);
}
```

Enfin, pour **lire** la valeur de **la luminosité ambiante**, on a une méthode **read()** :

```
float Sensor::read()
{
    i2cOut[0] = 0x04;
    i2cOut[1] = 0x00;
    i2cOut[2] = 0x00;
    bcm2835_i2c_setSlaveAddress(m_addr);
    if (bcm2835_i2c_write_read_rs(i2cOut,1,i2cIn,2) != BCM2835_I2C_REASON_OK){
        qDebug() << "lecteure I2C pas ok";
    } //if
    m_brightnessLevel = i2cIn[0] + 256*i2cIn[1];
    m_brightnessLevel *= 0.0576;

    return m_brightnessLevel;
}
```

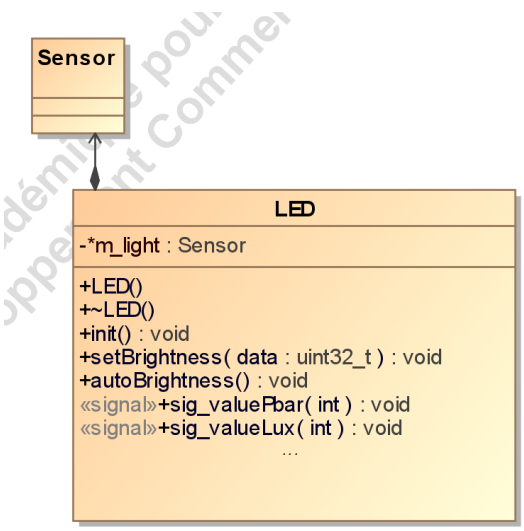
La fonction « **`bcm2835_i2c_setSlaveAddress(m_addr)`** » de la librairie bcm2835 permet la **connexion I2C** avec le capteur de luminosité en renseignant l'adresse de celui-ci.

La fonction « **`bcm2835_i2c_write_read_rs(i2cOut,1,i2cIn,2)`** » permet à la fois de **demander** la luminosité au capteur et à la fois de **lire** celle-ci : « i2cOut » **renseigne** les informations à renvoyer par le capteur, tandis que « i2cIn » **reçois** ces informations et nous permet de les **exploiter** grâce à un calcul donné dans la documentation.

Développement de la classe LED :

La deuxième étape est de développer la classe qui doit assurer le pilotage de la bande LED.

Voici le diagramme de classe :



Fichier led.h :

```
#ifndef PWMCOM_H
#define PWMCOM_H

#define PIN  RPI_V2_GPIO_P1_32
#define PWM_CHANNEL 0
#define RANGE 1024

#include <QObject>
#include <bcm2835.h>
#include "sensor.h"

/*
 * Classe LED
 * Gère la bande led en PWM avec la librairie BCM2835
 */

class LED : public QObject
{
    Q_OBJECT

public:
    LED();
    ~LED();

    void init();  //Méthode d'initialisation de la bande LED
    void setBrightness (uint32_t data);
```

```
    void autof();

private:
    Sensor *m_light;

signals:
    void valuePbar(int);
    void valueLux(int);
};

#endif
```

Cette classe utilise la **librairie BCM2835** pour piloter la LED grâce au **GPIO PWM** de la raspberry.

La méthode **init()** permet d'initialiser la librairie :

```
void LED::init()
{
    bcm2835_init();
    bcm2835_gpio_fsel(12, BCM2835_GPIO_FSEL_ALT0);
    bcm2835_pwm_set_clock(BCM2835_PWM_CLOCK_DIVIDER_256);
    bcm2835_pwm_set_mode(PWM_CHANNEL, 1, 1);
    bcm2835_pwm_set_range(PWM_CHANNEL, RANGE);
}
```

On utilise donc les méthodes de la librairie pour initialiser cette dernière.

On appelle ensuite cette méthode dans le constructeur de cette classe :

```
LED::LED()
{
    this->init();
    m_light = new Sensor;
}
```

On vient également instancier la classe **Sensor** développée précédemment.

Pour **piloter la luminosité** de la LED, on a une méthode simple :

```
void LED::setBrightness(uint32_t data)
{
    bcm2835_pwm_set_data(PWM_CHANNEL, data);
}
```

On vient encore ici utiliser la librairie **BCM2835**

Enfin, on a une dernière méthode permettant de gérer automatiquement l'intensité lumineuse de la LED grâce au capteur de luminosité :

```
void LED::autof() {

    int lightIntensity = (int)m_light->read();
    emit valueLux(lightIntensity);

    switch (lightIntensity) {
        case 0 ... 30:
            setBrightness(1000);
            emit valuePbar(100);
            break;
        case 31 ... 70:
            setBrightness(800);
```

```
            emit valuePbar(80);
            break;
        case 71 ... 100:
            setBrightness(500);
            emit valuePbar(50);
            break;
        case 101 ... 200:
            setBrightness(400);
            emit valuePbar(40);
            break;
        case 201 ... 500:
            setBrightness(300);
            emit valuePbar(30);
            break;
        case 501 ... 2000:
            setBrightness(100);
            emit valuePbar(10);
            break;
        default :
            setBrightness(0);
            emit valuePbar(0);
            break;
    } //switch
```

Cette méthode commence par aller lire la valeur de la luminosité ambiante retournée par le capteur grâce à la méthode **read()** de la classe **Sensor** vu précédemment.

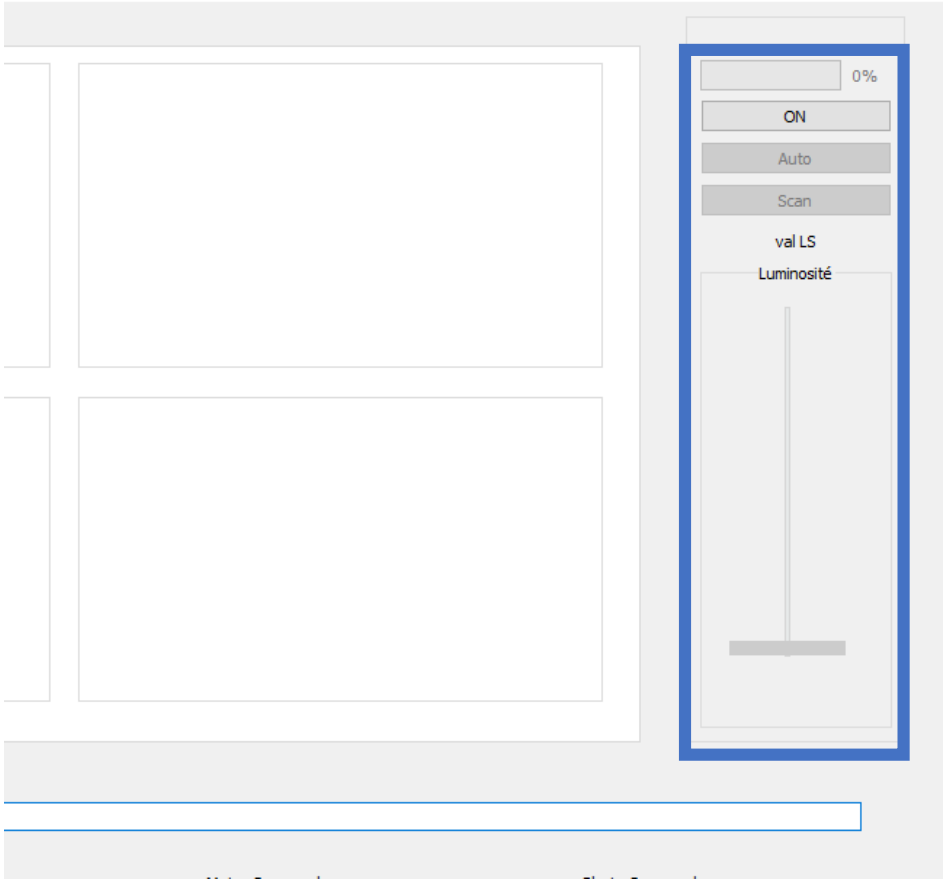
Ensuite, on utilise un **switch** pour définir la valeur de la luminosité en fonction de la luminosité ambiante.

Remarque : la syntaxe « **…** » dans le **case** permet de définir une intervalle de valeur, on peut traduire ca par : « Dans le cas où la valeur de lightIntensity est comprise entre 0 et 30 … »

Au moment de définir la luminosité, on émet également un signal qui contient aussi la valeur de la luminosité mais **divisé par 10**. Ce signal sera ensuite réceptionné par la classe gérant l'IHM pour mettre à jour l'interface graphique avec la valeur de l'intensité lumineuse de la LED.

Développement : Intégration de la partie éclairage à l'IHM

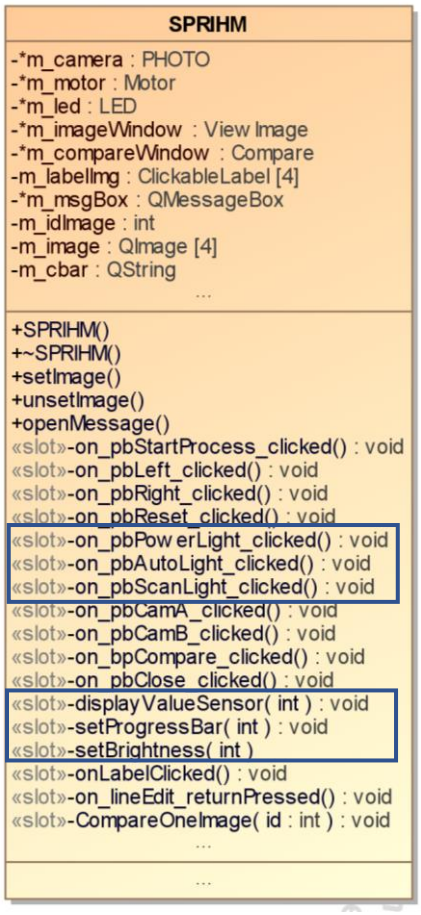
Voici la partie permettant de gérer l'éclairage sur l'IHM :



Il y'a donc 5 éléments importants :

- **Un bouton ON :** Permettant d'éteindre ou d'allumer la bande LED
- **Un bouton Auto :** Permettant de choisir le mode de réglage de la luminosité (automatiquement via le capteur de luminosité, ou manuellement via le slider)
- **Un bouton scan :** Permettant de mettre à jour l'intensité lumineuse en allant lire la valeur du capteur de luminosité lorsque le mode automatique est enclenché.
- **Un slider :** Permettant de contrôler manuellement l'intensité lumineuse de la LED lorsque le mode manuel est enclenché.
- **Une barre de progression :** Permet de visualiser le niveau de l'intensité lumineuse actuel de la LED.

Diagramme de classe de l'IHM :



Les **méthodes** (slots) encadrés en bleu sont associées aux différents éléments de l'IHM présenté ci-dessus.

Voici la gestion des cinq éléments de l'IHM cités précédemment permettant de piloter la bande LED :

1) **Le bouton « Power »** :
   Le bouton power est connecté au slot **on_pbPowerLight_clicked()** dont voici l'implémentation :

```
void sprIHM::on_pbPowerLight_clicked()
{
    QString etatBp = ui->pbPowerLight->text();

    if (etatBp == "ON") {
        m_slider->setValue(4);
        emit ui->slider->valueChanged(4);

        ui->pbPowerLight->setText("OFF");
        ui->progressBar->setEnabled(true);
        ui->pbAutoLight->setEnabled(true);
        ui->slider->setEnabled(true);

    } else if (etatBp == "OFF"){
        this->setBrightness(0);

        ui->pbPowerLight->setText("ON");
```

```
        ui->slider->setEnabled(false);
        ui->progressBar->setEnabled(false);
        ui->pbAutoLight->setEnabled(false);
        ui->pbScanLight->setEnabled(false);
    } // else if
}
```

Ce bouton va **activer** les objets via la fonction « `setEnabled(true/false)` » car, lors du lancement de l'IHM, la bande de LED sera **éteinte**, il faudra donc **l'allumer manuellement** avec le bouton « ON ». Celui-ci va aussi allumer la bande de LED à 40% de sa valeur.

Une fois **appuyé**, le bouton va voir son texte être **remplacé** par « OFF », et une fois réappuyer, le bouton va désactiver tous les objets et va voir son texte être **remplacé** par « **ON** », et ainsi de suite.

2) Le bouton « Auto » :

Ce bouton est connecté au slot **on_pbAutoLight_clicked()** :

```
void sprIHM::on_pbAutoLight_clicked()
{
    QString etatBp = ui->pbAutoLight->text();

    if (etatBp == "Auto") {
        ui->slider->setEnabled(false);
        ui->pbScanLight->setEnabled(true);
        ui->pbAutoLight->setText("Manual");

    } else if (etatBp == "Manual"){
        ui->slider->setEnabled(true);
        ui->pbScanLight->setEnabled(false);
        ui->pbAutoLight->setText("Auto");
    } //else if
}
```

Encore une fois, on vient simplement mettre à jour l'IHM.

3) Le bouton « Scan » :

```
void sprIHM::on_pbScanLight_clicked()
{
    m_led->autof();
}
```

On appelle simplement la méthode **autof()** expliquée précédemment.

4) Le « Slider » :

Le slider est connecté au slot **setBrightness()** :

```
void sprIHM::setBrightness(int value)
{
    int brightnessPBar = value * 10;
    int brightnessLED = value * 100;

    if (brightnessLED <= 100) {
        m_led->setBrightness(0);
        setProgressBar(0);

    } else {
        m_led->setBrightness(brightnessLED);
        setProgressBar(brightnessPBar);
    } // else
}
```

Ici on règle la luminosité de la LED, et l'indicateur de luminosité (QProgressBar) *

5) La barre de progression :

La barre de progression utilise le slot **setProgressBar()**

```
void sprIHM::setProgressBar(int value)
{
    ui->progressBar->setValue(value);
}
```

Cette méthode prend en paramètre un entier allant de 0 à 100 permettant qui définira la valeur de la barre de progression.

Ce slot est connecté au **signal émit** par la classe **LED** :

```
switch (lightIntensity) {
        case 0 ... 30:
            setBrightness(1000);
            emit valuePbar(100);
            break;
```

*Implémentation de la fonction **autof()***

La valeur émise est ensuite récupérée via le slot « `setProgressBar` » grâce à un « **connect** » dans le constructeur de la classe SPRIHM:

```
connect(m_led, &LED::valuePbar, this, &sprIHM::setProgressBar);
```

Utilisation d'une classe Hystérésis :

La classe « **Hystérésis** » a pour but de d'empêcher un possible scintillement.

Celui-ci peut être causé par le capteur de luminosité, par exemple, si le capteur relevé une valeur de 100 et que le seuil de changement de luminosité (pour passer de 30% à 40%) est de 100 aussi, alors la luminosité passera à 40%, mais si le capteur relève une autre valeur, par exemple 99, alors la luminosité de la bande de LED va repasser 30%.

Si cela se produis durant la prise de vue, alors les photos prisent auront une luminosité différente des autres.

Pour pallier à ce problème, nous allons donc utiliser un « Hystérésis ».

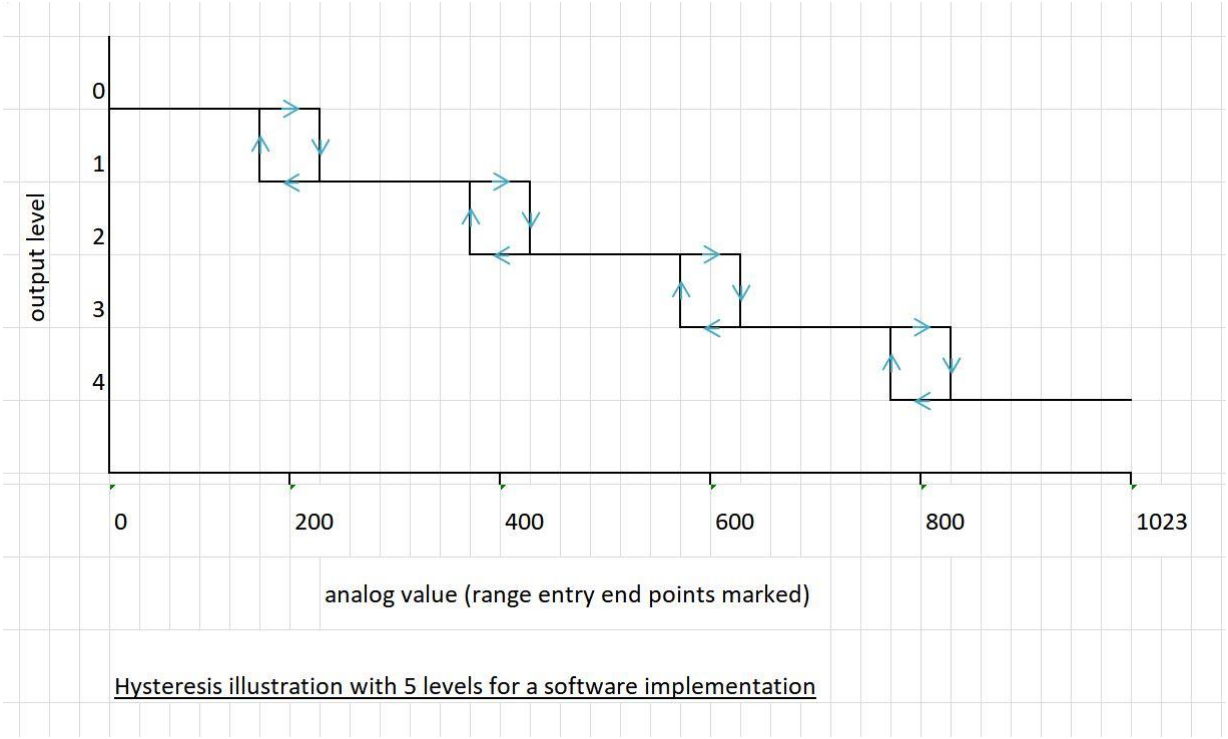
Voici la méthode utilisée :

```
#include "hystfilter.h"

HystFilter::HystFilter(uint16_t inputValues, uint16_t outputValues, uint16_t margin) :
  _inputValues(inputValues),
  _outputValues(outputValues),
  _margin(margin),
  _currentOutputLevel(0)
{  }


uint16_t HystFilter::getOutputLevel(uint16_t inputLevel) {

  // récupère la limite inferieure et supperieure de currentOutputLevel
  uint16_t lb = (float) ((float) _inputValues / _outputValues) * _currentOutputLevel;
  if (_currentOutputLevel > 0) lb -= _margin;   // subtract margin

  uint16_t ub = (((float)((float) _inputValues / _outputValues) * (_currentOutputLevel + 1)
) - 1);
  if (_currentOutputLevel < _outputValues) ub += _margin;  // add margin
  // teste si l'entrée est en dehors des marges extérieures pour la valeur de sortie actuel
le
  // si oui, calcule le nouveau niveau de sortie
  if (inputLevel < lb || inputLevel > ub) {
  // determine le nouveau niveau de sortie
    _currentOutputLevel = (((float) inputLevel * (float) _outputValues) / _inputValues) ;
  }
  return _currentOutputLevel;
}
```

Cette méthode a besoin de 3 paramètres pour fonctionner :

- « inputValue » qui représente le nombre total de valeur possible
- « outputValue » qui représente le nombre de seuil, proportionnellement au nombre total de valeur : pour une valeur totale de 100, si on choisis 5, alors cela donnera 5 seuil de 20.
- « margin » qui représente la marge du seuil le plus proche.

Voici un diagramme montrant le fonctionnement de la méthode :



Dans ce diagramme, les valeurs « inputValue », « outputValue » et « margin » serait :

- inputValue = 1024
- outputValue = 5
- margin = 33

Partie Étudiant 4 ÉLECTEUR Tony (EC1)

Objectifs

- Mettre en œuvre la carte de commande du plateau tournant
- Effectuer la saisie du schéma et le routage de ce Hat.
- Produire les fichiers Gerber afin que la fabrication du PCB soit sous-traitée.
- Câbler le PCB de la carte et effectuer les essais.
- Documenter la mise en service de la carte finale.

Avancement du projet

Cette partie est d'abord séparée pour le moteur et l'éclairage :

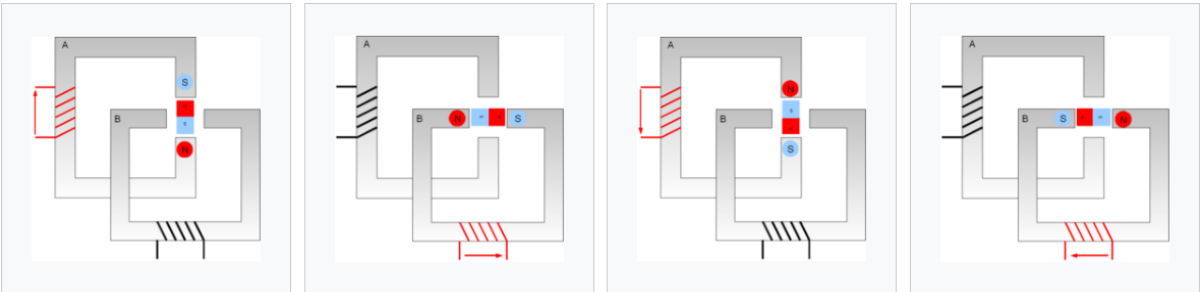
La partie moteur :

Pour cette partie nous allons utiliser un moteur pas à pas pour permettre de faire tourner les objets d'un quart de tours automatiquement ou manuellement pour réaliser des prises de vues sous tous les angles.

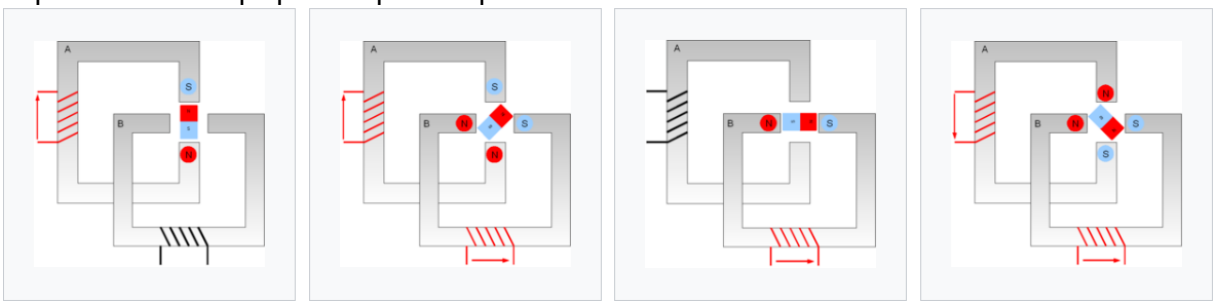
Le principe du moteur pas à pas est de faire de petit mouvement précis piloté par l'envoi d'impulsions électrique, l'intensité et la fréquence du signal permettra de contrôler la vitesse et le nombre de pas à effectuer.

Ce moteur est constitué de 2 bobines et d'un aimant, Ils ont plusieurs modes de fonctionnement mais tout le même principe de base.

Pour la première façon, il suffit d'alimenter une bobine qui est constituée de 2 pôles opposés et l'aiment va s'aligner à la bobine puis de changer et d'alimenter l'autre bobine l'aimant va donc tourner pour s'aligner avec la 2 eme bobine, on appelle ça le Full-Step



Le deuxième, le Half Step fonctionne en demi pas c'est le même principe mais l'on allume les 2 bobines en même temps pour que l'aimant s'arrête entre les 2 bobines avant d'arrêter la précédente ce qui permet plus de points d'arrêt.



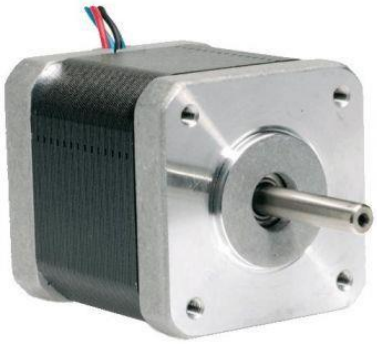
Le nombre de pas d'un moteur pas à pas, représente l'angle minimum que le moteur peut faire en une impulsion électrique et le nombre de pas qu'il faut pour faire un tour complet.

Le moteur pas à pas est donc le meilleur choix pour nous pour prendre des photos de 4 points de vue avec des rotation assez lente et précise.

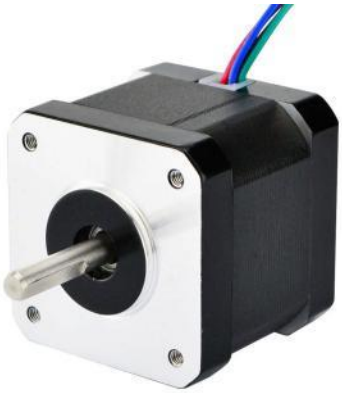
Possibilité de moteur pas à pas retenu après tests selon les exigences du projet (à approfondir) :

Nema 17HD6403 :

-Moteur bipolaire 200 pas
-Alimentation 12V



Nema 17HM15-0904S :
-Moteur bipolaire 400 pas
-Alimentation 5V



Pour le moteur nous avons choisi de le piloter en I2C nous allons donc utiliser un PCA9629A qui est un bus I2C pour contrôler le moteur, le PCA9629A peut être programmé pour arrêter, redémarrer ou inverser le sens de rotation du moteur.

Nous avons aussi pris le TB6612FNG qui cert d'interface de puissance pour l'alimentation du moteur.

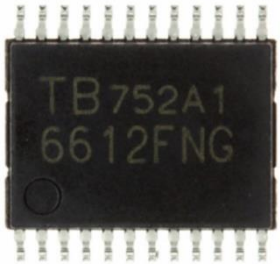
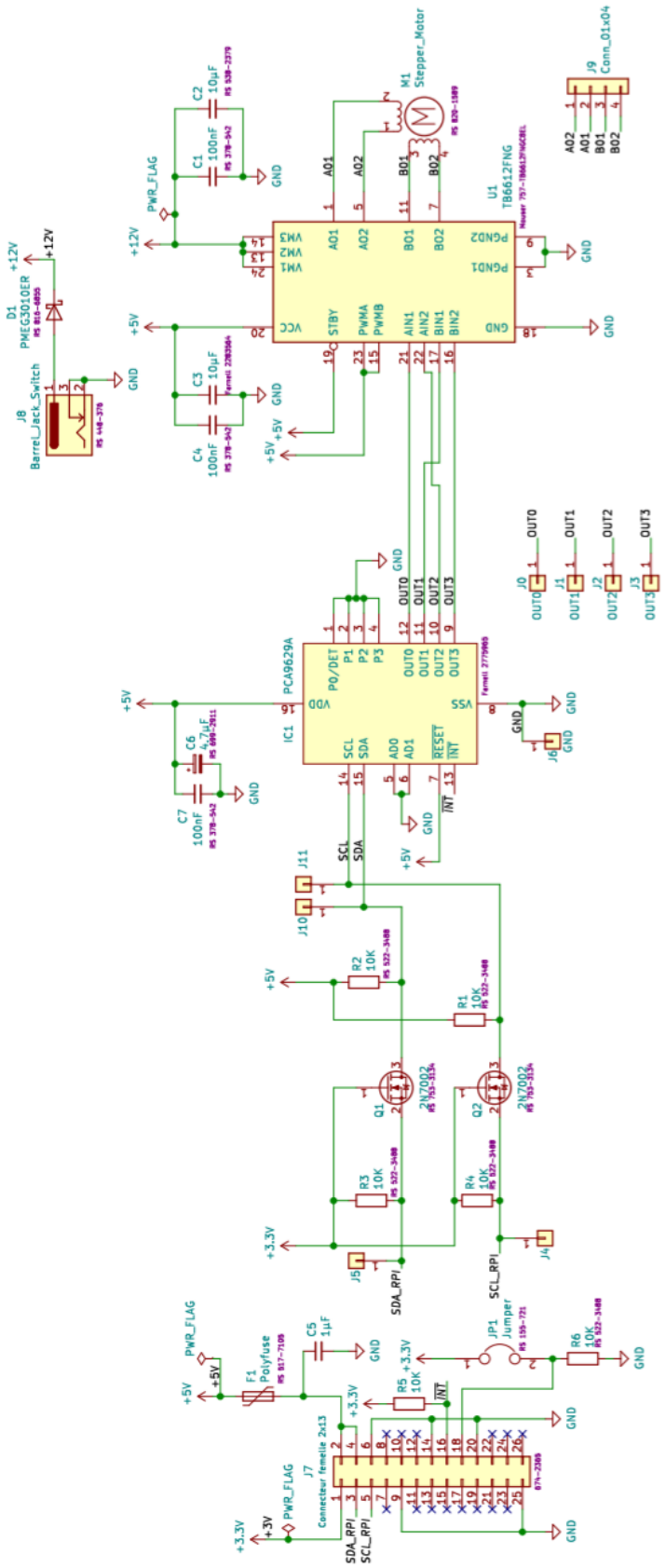


Schéma de la carte pilote moteur de l'année 2020 :



Pour les tests avec l'IHM des étudiants IR nous avons utilisé le moteur pas à pas 5V **Nema 17HM15-0904S.**

La partie éclairage :

Nous avions 2 possibilités pour cette partie :
-La première est un éclairage extérieur à l'aide de projecteurs
-La deuxième est de l'intérieur a l'aide de bande LED

Pour cette partie à la suite de tests de luminosité et aux conditions de l'entreprise nous avons choisis d'utiliser une bande LED contrôlée en PWM (modulation de largeur d'impulsion) et un tissu réfléchissant pour une meilleur luminosité et une contrainte de place avec les conditions données par l'entreprise les projecteurs prenaient trop de place.

N'ayant pas encore le matériel prévu, j'ai d'abord effectué des tests avec une bande LED 5V alimentée et contrôlée par une carte Arduino Uno.

J'ai fait 2 programmes :
L'un pour simuler le contrôle de luminosité automatique avec une LDR

```
int LDR = A5;
int Led = 3;

void setup() {
pinMode(Led,OUTPUT);
pinMode(LDR,INPUT);
}

void loop() {
  int value = analogRead(LDR);
  int value2 = map(value, 0, 1023, 255, 0);
  analogWrite(Led,value2);
}
```

L'autre pour simuler le contrôle manuel avec un potentiomètre

```
int Led = 3;
int Pot = A0; //potentiomètre


void setup() {
pinMode(Led,OUTPUT);
pinMode(Pot,INPUT);
}

void loop() {
  int value = analogRead(Pot);
  int value2 = map(value, 0, 1023, 0, 255);
  analogWrite(Led,value2);
}
```

Photo du montage test :

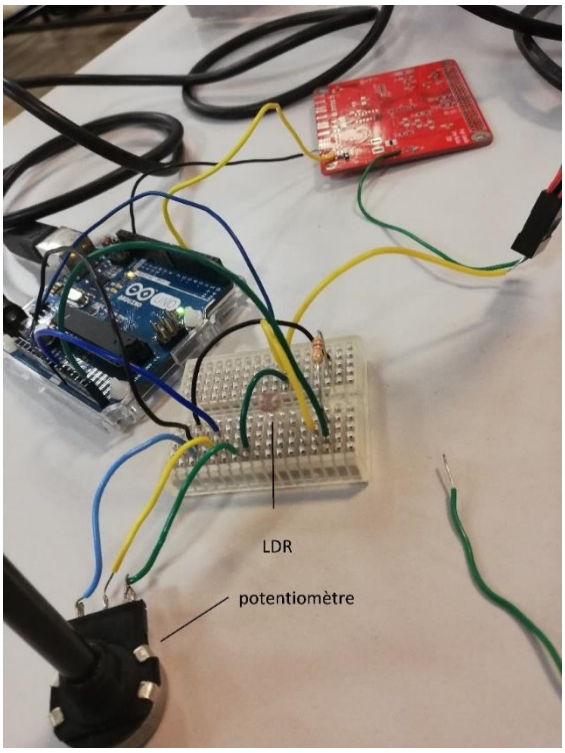
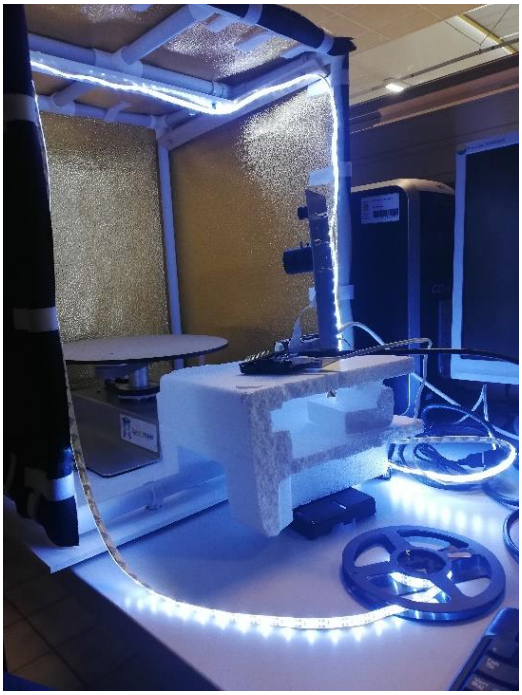


Photo du montage test :



Nous avons ensuite reçu 2 bandes LED :

-Ruban LED cold white SMD 5050

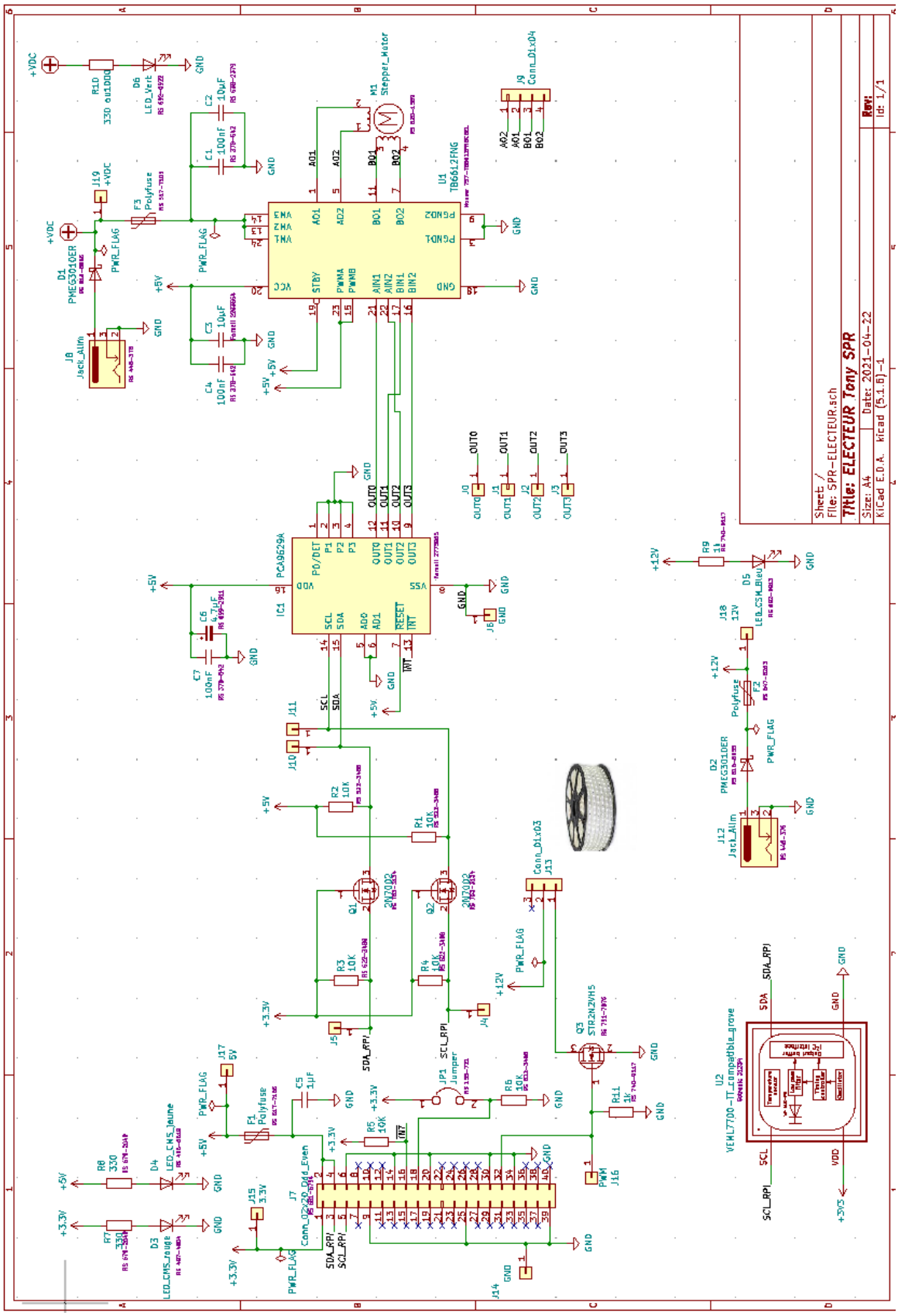
-Ruban LED cold white SMD 5630

À la suite de tests effectués pour chaque bandes LED nous avons choisis d'utiliser la bande LED SMD 5050 qui offre un meilleur rendu de luminosité pour le système.

Nous avons rencontré un problème avec la bande LED contrôlée avec l'IHM. Nous avons remarqué que la tension de la PWM ne descendait pas à 0 quand on la pilotait avec la Raspberry donc la bande LED ne s'éteignait pas, pour résoudre ce problème nous avons changé le transistor pour en prendre un avec un VGSth min de 0.6V nous avons donc pris un **Transistor MOS STR2N2VH5.**

Création du nouveau Hat Rpi :

Pour commencer la création du nouveau Hat Rpi j'ai utilisé le schéma fait l'année précédente en y rajoutant ce qu'il manquait.



Par rapport a celui de l'année précédente j'y ai rajouté la partie éclairage afin de piloter la bande LED en plus de la partie moteur ainsi qu'un nouveau point d'alimentation pour piloter la bande LED. J'y ai aussi ajouté des Leds pour vérifier le bon fonctionnement des différentes alimentations ainsi que différents points de test et des fusibles pour contrôler son

fonctionnement à tous moments. ET le dernier ajout est une sortie pour récupérer les informations du capteur de lumière.

Voici ci-dessous la liste des composants utilisé pour cette carte :

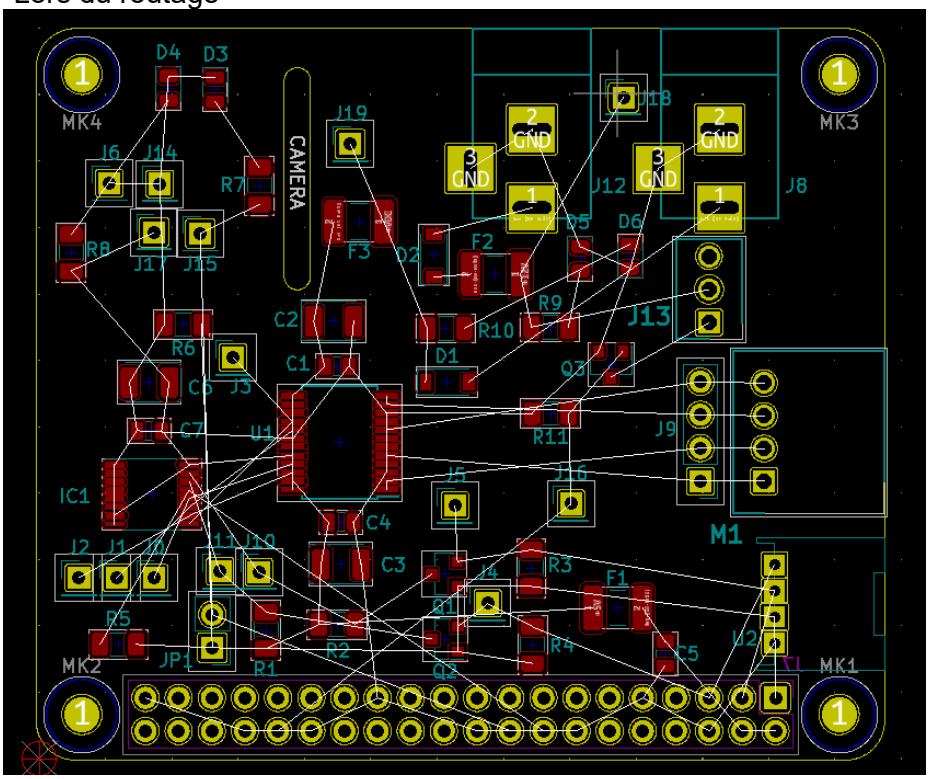
Source:
Preojet SPR
ELECTEUR Tony
Liste des composants :

Item	Qty	Reference(s)	Value	Code commande
1	3	C1, C4, C7	100nF	RS 378-542
2	2	C2, C3	10µF	RS 538-2379
3	1	C5	1µF	RS 264-4450
4	1	C6	4.7µF	RS 699-2911
5	2	D1, D2	PMEG3010ER	RS 816-6855
6	1	D3	LED_CMS_rouge	RS 497-4804
7	1	D4	LED_CMS_jaune	RS 486-0519
8	1	D5	LED_CSM_Bleu	RS 692-0913
9	1	D6	LED_Vert	RS 692-0922
10	3	F1, F2, F3	Polyfuse	RS 517-7105
11	1	IC1	PCA9629A	Farnell 2775965
12	15	J0, J1, J2, J3, J4, J5, J6, J10, J11, J14, J15, J16, J17, J18, J19	Conn_01x01	Gotronic 08000
13	1	J7	Conn_02x20	RS 681-6794
14	2	J8, J12	Jack_Alim	RS 448-376
15	1	J9	Conn_01x04	RS 820-1551
16	1	J13	Conn_01x03	RS 532-096
17	1	JP1	Jumper	RS 155-721
18	1	M1	Stepper_Motor	RS 820-1589
19	2	Q1, Q2	2N7002	RS 753-3134
20	1	Q3	STR2N2VH5	RS 791-7876
21	6	R1, R2, R3, R4, R5, R6	10K	RS 522-3488
22	2	R7, R8	330	RS 679-2049
23	2	R9, R11	1k	RS 740-9117
24	1	R10	330 ou1000	
25	1	U1	TB6612FNG	Mouser 757-TB6612FNGC8EL
26	1	U2	VEML7700-TT_compatible_grove	Gotronic 31234

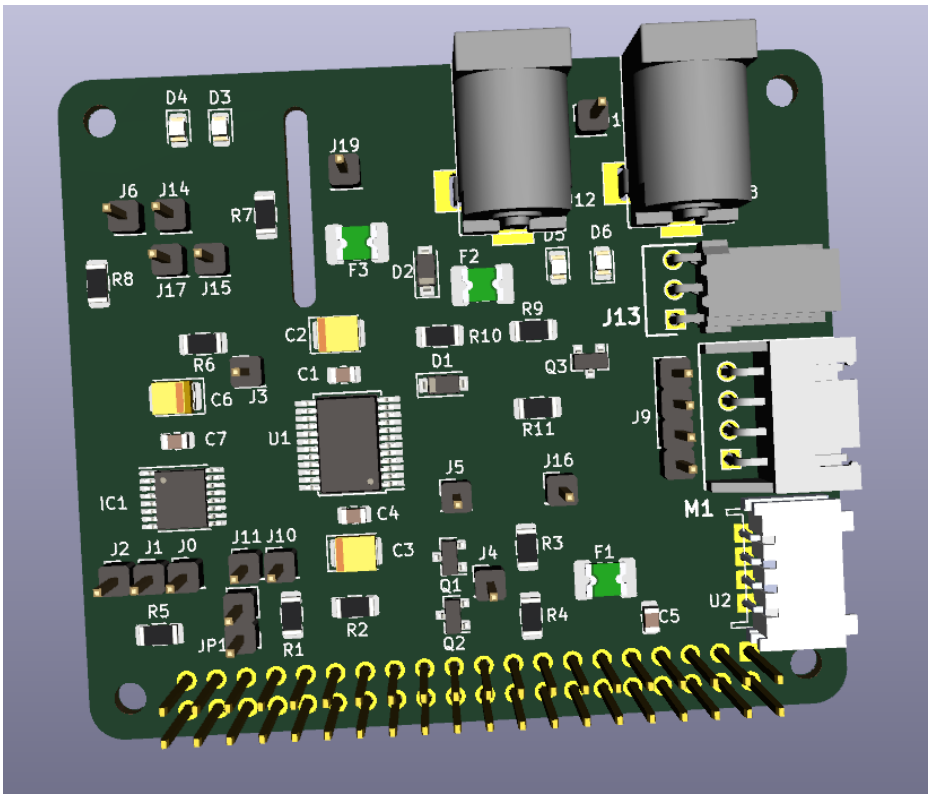
J’ai ensuite commencé le routage de la carte :

Voici un premier aperçu de la carte avec les composants placés (quelques changements sont encore nécessaires) :

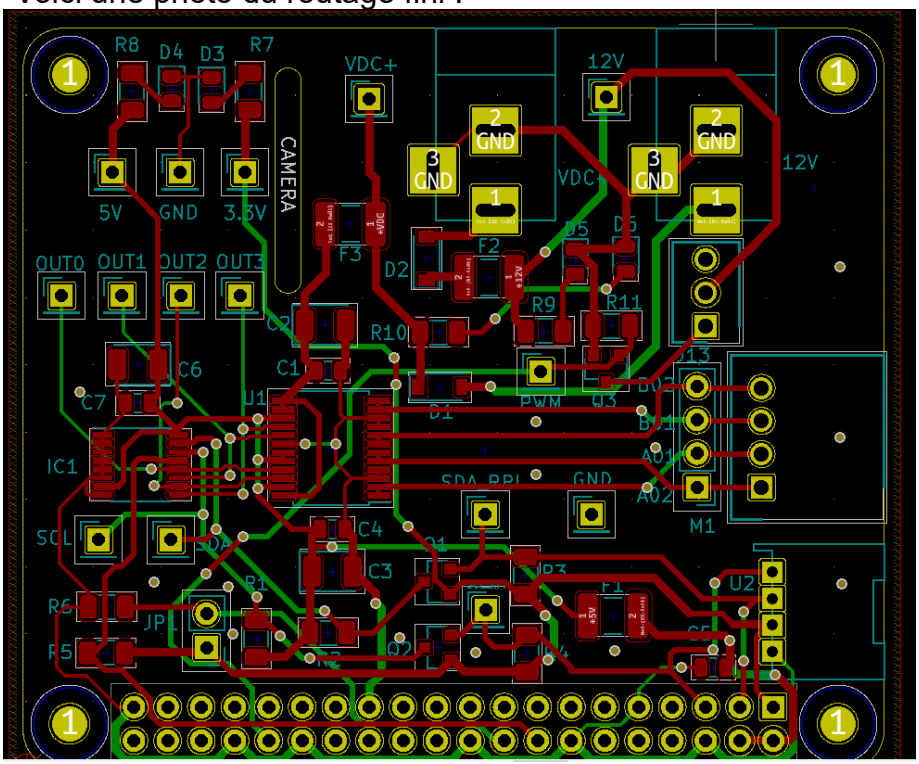
-Lors du routage



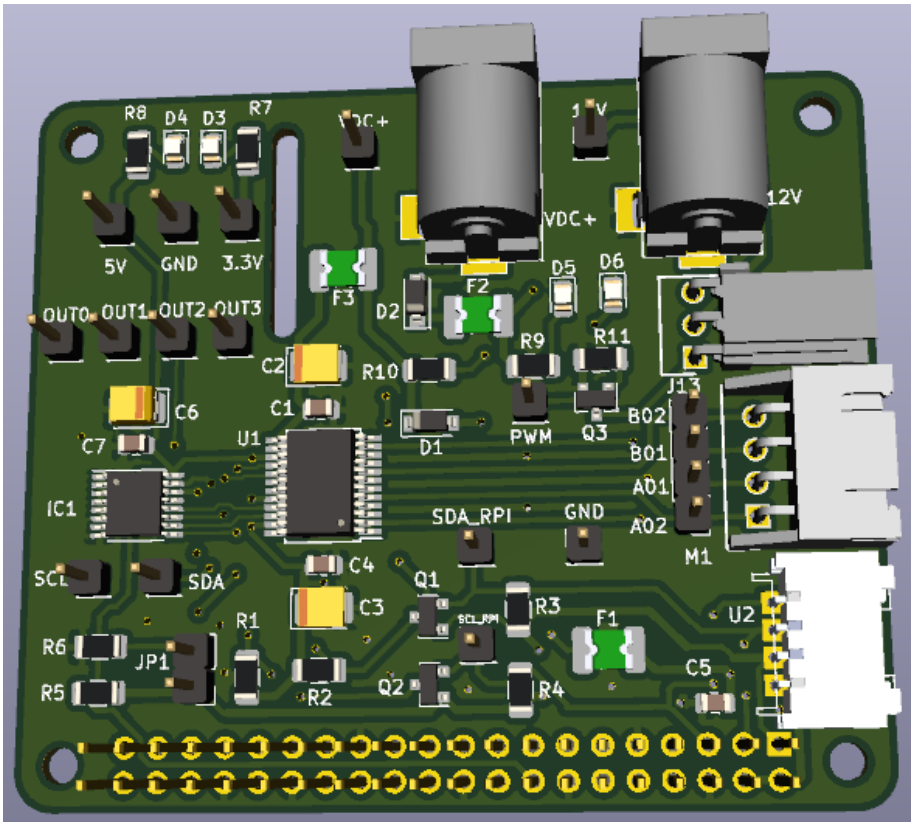
-Visualisation 3D



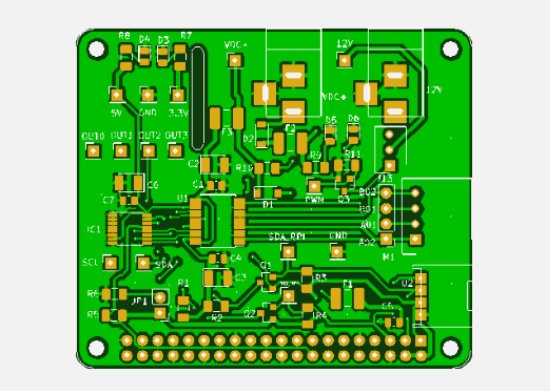
-Voici une photo du routage fini :

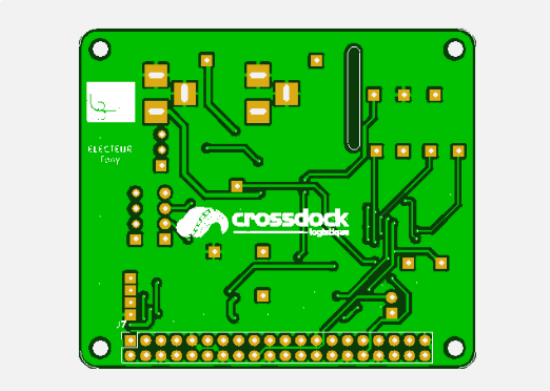


-Et de la visualisation 3D :



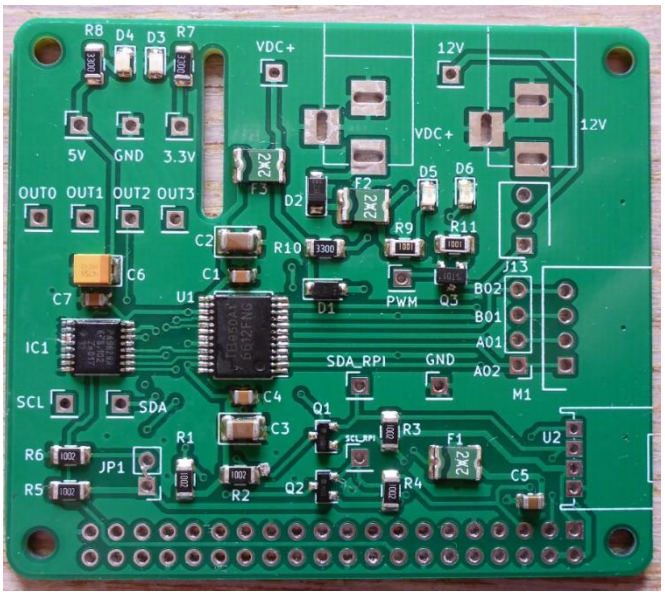
-Après la création de fichier GERBER
Voici le rendu de la simulation de la création de PCB sur JLCPCB :



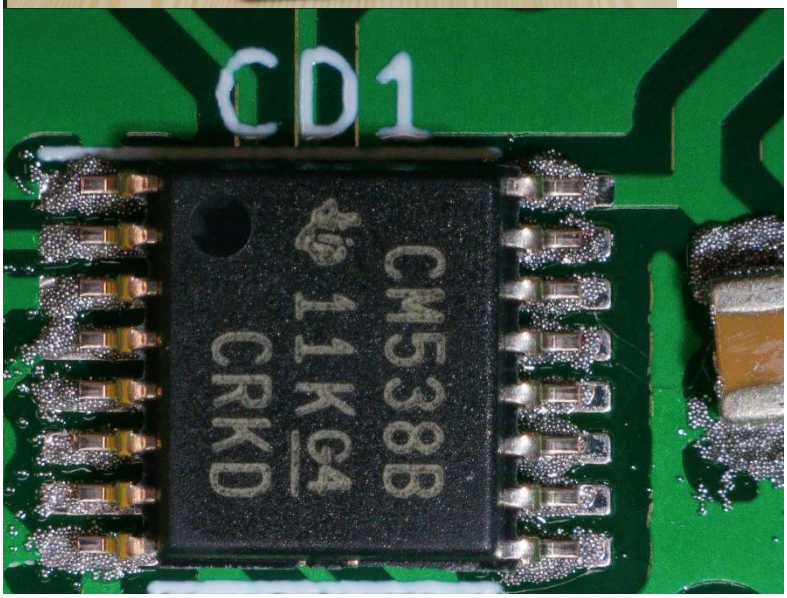


À la suite de ça nous avons reçu le PCB ainsi que le stencil (nous avons remarqué a la réception du PCB que l'appellation des 2 Jack-alim était inversés) :

Nous avons commencé la soudure au four pour les CMS et brasé pour les composants en CMS :



On a vérifié les soudures à l'aide d'un microscope :



Application finale :

Objectifs

Cette partie fait suite aux parties de tous les étudiants. L'objectif est de développer ici le travail commun à tous les étudiants, à savoir la **comparaison d'image**, l'optimisation de **l'expérience utilisateur**…

Comparaison des images

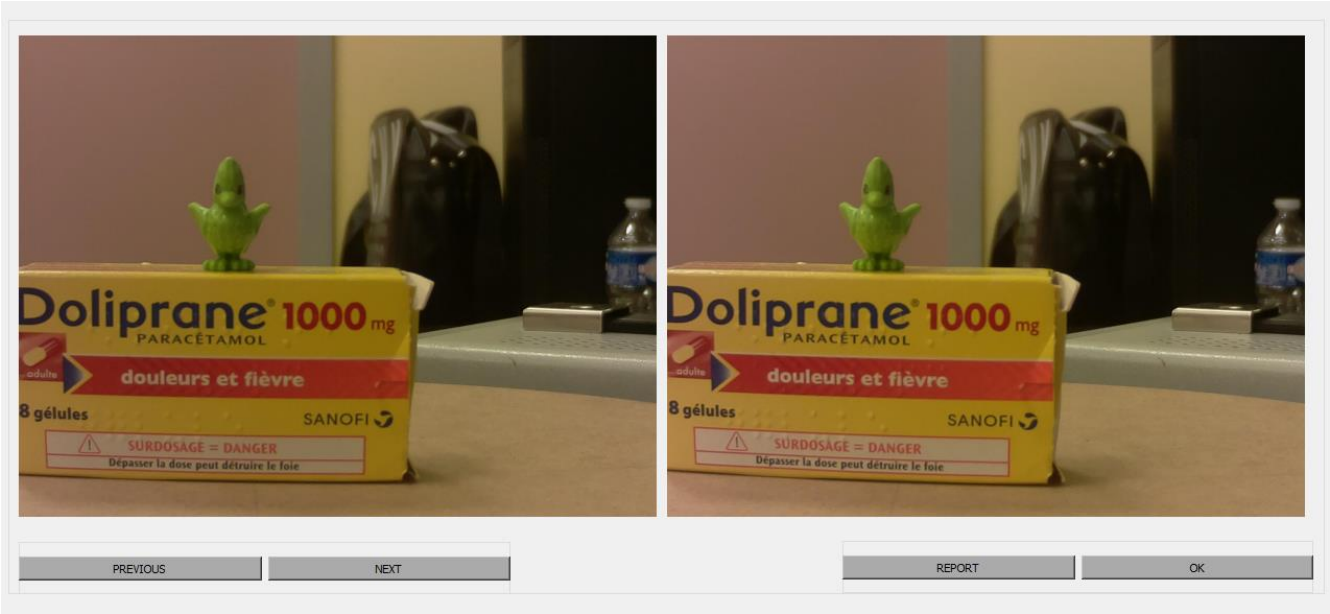
L'objectif principal du système est de pouvoir comparer les images qui viennent d'être prises (à la date courante) avec les images prises à la date précédente.

On affichera ainsi les images deux par deux : L'image d'une des faces du produit qui vient d'être prise et l'image de la même face du même produit prise à la date précédente.

En cas de plusieurs dates présentes pour un même produit, il faudra faire attention à récupérer la date la plus récente mais qui n'est pas c'elle du jour.

Si une différence est constatée, l'opérateur aura la possibilité de notifier cette différence en rédigeant celle-ci dans une zone de texte. La note pourra être ensuite enregistrée dans un fichier texte à l'emplacement des images enregistrées à la date courante.

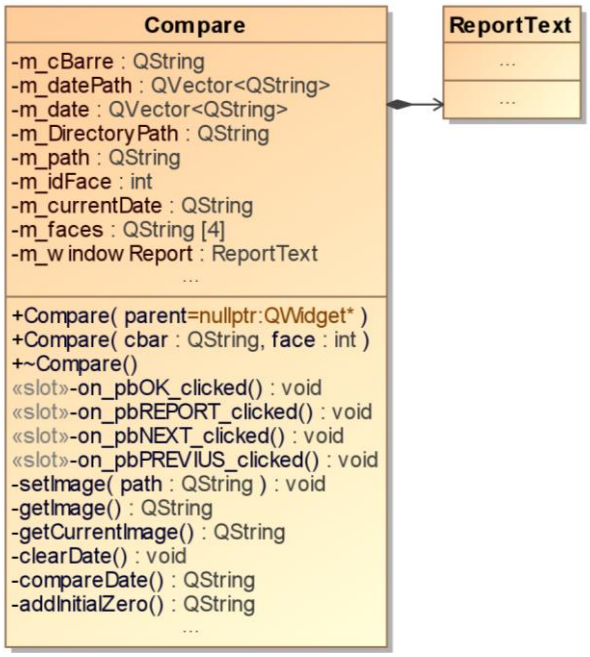
Nous avons donc créé une nouvelle classe avec une IHM :



Les photos ci-dessus ont été prises durant des tests.

L'image de droite est la photo prise à la date courante et c'elle de gauche est l'image prise à la date précédente.

Cette IHM fonctionne au sein d'une classe nommée **Compare**. Voici le diagramme de classe :



Fichier Compare.h :

```
#ifndef COMPARE_H
#define COMPARE_H

#include "reporttext.h"

#include <QDebug>
#include <QDirIterator>
#include <QMainWindow>
#include <QVector>

namespace Ui {
class Compare;
}

class Compare : public QMainWindow
{
    Q_OBJECT

public:
    explicit Compare(QWidget *parent = nullptr);
    Compare(QString, int);
    ~Compare();

private slots:
    void on_pbOK_clicked();

    void on_pbREPORT_clicked();

    void on_pbNEXT_clicked();

    void on_pbPREVIUS_clicked();
```

```
private:
    Ui::Compare *ui;
    QString m_cbar;
    QVector<QString> m_datePath;
    QVector<QString> m_date;
    QString m_DirectoryPath = "/home/pi/Desktop/stockPhotoShare/";
    QString m_path;

    void setImage(QString);

    QString getImage();
    QString getCurrentImage();
    void clearVoidElement();
    void clearCurrentDate();
    QString compareDate();
    QString addInitialZero();

    QString m_faces[4] = {"F", "R", "B", "L"};
    int m_idFace;
    QString m_curentDate;

    ReportText *m_windowReport;

};

#endif // COMPARE_H
```

Nous utilisons un constructeur surchargé prenant en argument le chemin des quatre images (chaque faces) à comparer, ainsi qu'un entier de 0 à 3 pour identifier la face à comparer.

Passer l'identifiant de la face en argument est utile lorsque l'on va choisir qu'elle image (face) on veut comparer en cliquant dessus depuis l'IHM principale, Nous reviendrons sur cette fonctionnalité dans la partie suivante : **Agrandissement des images**.

Voici l'implémentation du constructeur :

```
Compare::Compare(QString cbarre, int face) :
    ui(new Ui::Compare),
    m_idFace(face),
    m_cbarre(cbarre)
{
    ui->setupUi(this);
    m_curentDate = QDate::currentDate().toString(QString("dd-MM-yyyy"));
    this->setImage(this->getImage());
}
```

On utilise une liste d'initialisation pour affecter le code-barre et l'identifiant de là face à des variables membres (attributs) pour pouvoir les réutiliser ailleurs dans la classe.

On initialise ensuite l'attribut **m_currentDate** avec la date courante.

Enfin, on appelle la méthode **setImage()** pour afficher les images sur l'IHM. Cette méthode prend en argument le chemin de l'image à comparer. On appelle donc directement la méthode **getImage()** en argument.

Voici l'implémentation de la méthode **getImage()** :

```
QString Compare::getImage()
{
    QDirIterator it("/home/pi/Desktop/stockPhotoShare/" + m_cbar + "/", QDir::Dirs, QDirIte
rator::NoIteratorFlags);
    int val = 0;
    while (it.hasNext()) {
        m_datePath.push_back(it.next());
        m_date.push_back(m_datePath[val].section('/', 6, 6));
        if (m_date[val] == ".." || m_date[val] == ".") {
            m_date[val].clear();
        } //if
        qDebug() << "date origines" << m_date[val];
        val++;
    } //while
    this->clearDate();
    QString finalDate = this->compareDate();
    finalDate = this->addInitialZero(finalDate);

    m_date.clear();
    m_datePath.clear();

    return m_DirectoryPath + m_cbar + "/" + finalDate + "/" + m_cbar + "-" + finalDate + "-
" + m_faces[m_idFace] + ".jpg";
}
```

L'objectif ici est de trouver automatiquement la version la plus récente des images prises pour un même produit (ou plutôt des produits de même références).

Imaginons que l'entreprise ai reçue quatre livraisons de stock d'un produit en quatre semaines, les produits datant de la livraison du jours doivent être comparés avec les photos datant de la semaine dernière (la dernière livraison) et non celle d'il y a deux semaines ou plus.

La première étape va être de récupérer toutes les dates auxquelles il y a eu des prises de vues pour un même produit.

Pour rappel, la sauvegarde des photos se fait avec l'arborescence suivante :

code_barre/date/date-code_barre-face.jpg

Il nous faut donc récupérer tous les répertoires « **date** » dans le dossier d'un produit (**code_barre**).

Voici un exemple plus visuel :

La première étape est donc de récupérer l'ensemble des noms des répertoires (l'ensemble des dates). Pour cela on utilise la classe **QDirIterator**.

Cette classe permet de naviguer entre plusieurs répertoires. Ici ont s'en sert pour les lister :

```
QDirIterator it("/home/pi/Desktop/stockPhotoShare/" + m_cbar + "/", QDir::Dirs, QDirIterato
r::NoIteratorFlags);
    int val = 0;
    while (it.hasNext()) {
        m_datePath.push_back(it.next());
        m_date.push_back(m_datePath[val].section('/', 6, 6));
        if (m_date[val] == ".." || m_date[val] == ".") {
            m_date[val].clear();
        } //if
        val++;
    } //while
```

On commence par créer une instance de cette classe nommée **it**. Dans le constructeur, le premier argument que l'on donne est le chemin du répertoire parant des répertoires que l'on souhaite lister.

L'instance se comporte donc comme un tableau. On utilise une **boucle while** pour la parcourir. Le **chemin absolu** est ensuite **stocké** dans l'attribut **m_datePath** qui est un **QVector** permettant de créer des tableau a taille variable (on ne connait pas à l'avance la taille que fera le tableau, elle dépend du nombre de sous répertoire **date** qu'il y a), c'est pour cela que l'on utilise la méthode **push_back()** pour insérer les éléments dans le tableau.

L'attribut **m_pathDate** contient donc les chemins absolus de chaque répertoire. Or, notre objectif est de récupérer uniquement la dernière partie du chemin : la date.

/home/pi/Desktop/stockPhotoShare/2310/21-05-2021

Il nous faut donc couper la chaine de caractère pour ne récupérer que la date. Pour cela, la classe **QString** dispose d'une méthode **section()** permettant d'extraire une partie d'une chaine de caractère :

```
m_date.push_back(m_datePath[val].section('/', 6, 6));
```

Cette méthode prend en argument un **délimiteur** (un caractère qui identifie l'endroit ou on doit couper la chaine), ici ce délimiteur est « **/** ». Ensuite on indique à partir de quel **slash** il doit couper la chaine. Dans notre cas, on compte sur le chemin absolu que la date est après le 6ème slash. Le dernier argument indique jusqu'à quel autre slash on doit couper.

Suite à cela, il y a un problème : parmi les chemins listés par l'instance de **QDirIterator**, on a également les chemins suivants : « **..** » et « **.** » qui correspondent respectivement au répertoire parent et au répertoire courant.

L'étape suivante consiste donc à retirer ces deux chemins :

```
if (m_date[val] == ".." || m_date[val] == ".") {
            m_date[val].clear();
        } //if
```

On utilise la méthode **clear()** de la classe **QVector** qui a pour but de supprimer le contenu d'une case **mais ne supprime pas la case elle-même**.

Pour ne pas trop surcharger la méthode **getDate()** il a été décidé de morceler la logique en passant par des méthode intermédiaire. Cela permet d'avoir un code plus lisible.

L'étape suivante consiste donc à supprimer les éléments vide du tableau (dû à l'étape précédente où on a retiré les deux éléments indésirables).

On appelle donc la méthode **clearDate()** :

```
this->clearDate();
```

Voici son implémentation :

```
void Compare::clearDate()
{
    for (int i = 0; i < m_date.size(); ++i) {
        //Suppression de la date courante
        if (m_date[i] == m_curentDate) {
            m_date[i] = "";
        } //if
        //Supression des élements vides
        if (m_date[i] == "") {
            for (int j = i; j < m_date.size(); j++) {
                if(m_date[i+1] == "") {
                    if (m_date[i+2] == "") {
                        m_date[i] = m_date[i+3];
                    } else {
                        m_date[i] = m_date[i+2];
                    } //else
                } else {
                    m_date[i] = m_date[i+1];
                } //else
            } //for
        } //if
    } //for
}
```

Étant donné que l'on ne veut pas la date d'aujourd'hui comme expliqué précédemment, on commence par la retirer en parcourant toutes les dates.

Ensuite, pendant ce parcoure, si une date est **nulle** (vide) alors on regarde si la date suivante n'est pas **nulle** et on la copie à la place de la case actuelle du tableau qui est vide.

Dans le cas contraire, si la date suivante est nulle aussi, alors on regarde la date d'après et on fait ça 3 fois car on ne peut avoir maximum que trois dates nulles : Les deux chemins retirés précédemment et la date du jour.

Enlever les éléments vide est essentiel pour que la partie suivante consistant à comparer les dates entre elles fonctionne.

Revenons à la méthode **getImage()**, l'étape suivante consiste à comparer les dates entre elles pour récupérer la plus récente. On appelle donc la méthode **compareDate()** :

```
QString finalDate = this->compareDate();
```

Voici l'implémentation de cette méthode :

```
QString Compare::compareDate()
{
    int intDate[m_date.size()][3];

    for (int i = 0; i < m_date.size(); i++) {
        for (int j = 0; j < 3; j++) {
            intDate[i][j] = m_date[i].section('-
', j, j).toInt(); //Séparation de la date en J M A
        } //for
    } //for

    QString finalDate;
    //Variables tempons
    int lastYear(0);
    int lastMonth(0);
    int lastDay(0);

    //Algorithme de comparaison des dates
    for (int i = 0; i < m_date.size(); i++) {
        if (i < m_date.size() - 1) {

            //Comparaison de l'année
            if (intDate[i][0] != 0 && intDate[i][1] != 0 && intDate[i][2] != 0) {
                if ((intDate[i][2] < intDate[i+1][2])) {
                    if (lastYear < intDate[i+1][2]) {
                        lastYear = intDate[i+1][2];
                        finalDate = QString::number(intDate[i+1][0]) + "-
" + QString::number(intDate[i+1][1]) + "-" + QString::number(intDate[i+1][2]);
                    } //if
                } else if (intDate[i][2] > intDate[i+1][2]) {
                    if (lastYear < intDate[i][2]) {
                        lastYear = intDate[i][2];
                        finalDate = QString::number(intDate[i][0]) + "-
" + QString::number(intDate[i][1]) + "-" + QString::number(intDate[i][2]);
                    } //if

                    //Comparaison du mois si les années de deux dates sont identiques
                } else if ((intDate[i][2] == intDate[i+1][2]) && (intDate[i][2] >= lastYear
)) {
                    if (intDate[i][1] < intDate[i+1][1]) {
                        if (lastMonth < intDate[i+1][1]) {
                            lastMonth = intDate[i+1][1];
                            finalDate = QString::number(intDate[i+1][0]) + "-
" + QString::number(intDate[i+1][1]) + "-" + QString::number(intDate[i+1][2]);
                        } //if
                    } else if (intDate[i][1] > intDate[i+1][1]) {
                        if (lastMonth < intDate[i][1]) {
                            lastMonth = intDate[i][1];
                            finalDate = QString::number(intDate[i][0]) + "-
" + QString::number(intDate[i][1]) + "-" + QString::number(intDate[i][2]);
                        } //if

                        //Comparaison des jours si les mois de deux dates sont identiques
                    } else if ((intDate[i][1] == intDate[i+1][1]) && (intDate[i][1] >= last
Month)) {
                        if (intDate[i][0] < intDate[i+1][0]) {
                            if (lastDay < intDate[i+1][0]) {
                                lastDay = intDate[i+1][0];
                                finalDate = QString::number(intDate[i+1][0]) + "-
" + QString::number(intDate[i+1][1]) + "-" + QString::number(intDate[i+1][2]);
                            }

                        } else if (intDate[i][0] > intDate[i+1][0]) {
```

```
                            if (lastDay < intDate[i][0]) {
                                lastDay = intDate[i][0];
                                finalDate = QString::number(intDate[i][0]) + "-
" + QString::number(intDate[i][1]) + "-" + QString::number(intDate[i][2]);
                            } // if
                        } // else if
                    } // else if
                } //else if
            } //if
        } //if
    } //for
    return finalDate;
}
```

Comme on le voit le code de cette méthode est très lourd mais pas si compliqué. L'objectif est de comparer les dates par années, par mois, puis par jours.

On commence donc par créer un tableau multidimensionnel (deux dimensions). Les lignes représentent le nombre de dates et les colonnes la date décomposée. On a donc trois colonnes (jours, mois, années) pour chaque date :

```
int intDate[m_date.size()][3];
```

Pour définir la taille des lignes, on utilise simplement la méthode **size()** pour récupérer la taille du tableau **m_date** qui contient les dates.

Pour remplir ce nouveau tableau, on viens à nouveau utiliser la méthode **section()** de la classe **QString** pour séparer la chaine de caractère :

```
for (int i = 0; i < m_date.size(); i++) {
    for (int j = 0; j < 3; j++) {
      intDate[i][j] = m_date[i].section('-', j, j).toInt(); //Séparation de la date
    } //for
} //for
```

Vient ensuite l'algorithme de comparaison dans lequel on va venir comparer l'année de la case du tableau parcouru avec la méthode suivante :

```
if (intDate[i][2] < intDate[i+1][2]) {
   if (lastYear < intDate[i+1][2]) {
      lastYear = intDate[i+1][2];
      finalDate = QString::number(intDate[i+1][0]) + "-
" + QString::number(intDate[i+1][1]) + "-" + QString::number(intDate[i+1][2]);
   } //if
```

La variable **lastYear** va sauvegarder l'année la plus récente qu'elle rencontrera, comme ça, pour les dates suivantes, si elles ont des années inférieures à cette variable, on ne définit pas de nouvelle date.

Cela est nécessaire lorsque l'on a une grande liste de date car si la date que l'on cherche (la plus récente) se trouve en haut de la liste, elle sera oubliée lorsque la boucle avancera étant donné qu'une nouvelle date sera définit. Le test de la variable lastYear est donc indispensable.

Si le premier test est faux, alors on fait le teste inverse :

```
else if (intDate[i][2] > intDate[i+1][2]) {
                    if (lastYear < intDate[i][2]) {
                        lastYear = intDate[i][2];
                        finalDate = QString::number(intDate[i][0]) + "-
" + QString::number(intDate[i][1]) + "-" + QString::number(intDate[i][2]);
```

Si non, si les deux années sont identiques, alors on fait la même chose mais en testant les mois :

```
} else if ((intDate[i][2] == intDate[i+1][2]) && (intDate[i][2] >= lastYear)) {
          if (intDate[i][1] < intDate[i+1][1]) {
             if (lastMonth < intDate[i+1][1]) {
                 lastMonth = intDate[i+1][1];
                 finalDate = QString::number(intDate[i+1][0]) + "-
" + QString::number(intDate[i+1][1]) + "-" + QString::number(intDate[i+1][2]);
             } //if
```

Et ainsi de suite jusqu'à arriver au jour (voir le code complet plus haut).

Puis il suffit à la fin de retourner la date la plus récente.

Un nouveau problème est que lors de la conversion de la date en entier pour la comparaison, le programme enlève les zéros initiaux des mois et des jours.

Par exemple : on a **22-5-2021** au lieu de **22-05-2021**. Cela est problématique car les répertoires sont nommés avec des dates ayants des zéros initiaux.

C'est la qu'intervient la méthode **addInitialZero()**. On l'appelle à la suite, dans la méthode **getImage() :**

```
finalDate = this->addInitialZero(finalDate);
```

Ici, **finalDate** est la variable contenant le retour de la méthode de comparaison précédente.

Voici l'implémentation de la méthode **addInitialZero()** :

```
QString Compare::addInitialZero(QString finalDate)
{
    QString dateWithoutZero[3];
    for (int i = 0; i < 3; i++) {
        dateWithoutZero[i] = finalDate.section("-", i, i);
        if (dateWithoutZero[i].toInt() < 10) {
            dateWithoutZero[i] = "0" + dateWithoutZero[i];
        }
    }
    return finalDate = dateWithoutZero[0] + "-" + dateWithoutZero[1] + "-
" + dateWithoutZero[2];
}
```

On commence par redécouper la date dans le tableau **dateWithoutZero()** qui contient trois lignes pour le jour, le mois et l'année. Puis on vient simplement ajouter un zéro si le chiffre est inférieur à 10.

Enfin, on réassemble la date en chaine de caractère sous son format initial : **jour-mois-année**.

Pour finir avec la méthode **getImage()**, on réinitialise les attributs **m_date** et **m_datePath** :

```
    m_date.clear();
    m_datePath.clear();
```

Puis avec la date récupérée, on assemble le chemin final et on le retourne :

```
return m_DirectoryPath + m_cbar + "/" + finalDate + "/" + m_cbar + "-" + finalDate + "-
" + m_faces[m_idFace] + ".jpg";
```

L'attribut **m_faces** est un tableau contenant les lettres des différentes faces des produits (ceci est expliqué dans la partie **prise de vues** de l'étudiant 1). On sélectionne c'elle que l'on veut grâce à l'attribut **m_idFace** dont la valeur est passée en argument dans le constructeur de la classe.

Voir la page 105 pour le code complet de cette méthode.

Revenons ensuite à la méthode **setImage()** appelée dans le constructeur de la classe :

```
this->setImage(this->getImage());
```

Maintenant que l'on a vu le code la méthode **getImage()**, voyons l'implémentation de la méthode **setImage()** :

```
void Compare::setImage(QString path)
{
    ui->img2->setPixmap(QPixmap(path));
    ui->img2->setScaledContents(true);

    ui->img1->setPixmap(this->getCurrentImage());
    ui->img1->setScaledContents(true);
}
```

On vient simplement définir la propriété **Pixmap** des labels pour afficher les images.

La méthode **setScaledContents()** permet d'adapter la taille de l'image au label.

À noter : Les **QLabel** ont été dimensionnés avec même ratio que la résolution de la caméra pour ne pas déformer l'image.

Pour le deuxième label, on vient simplement récupérer la date qui a été prise à la date du jour. Pour cela, on passe par la méthode **getCurrentImage()** :

```
QString Compare::getCurrentImage()
{
    m_path = m_DirectoryPath + m_cbar + "/" + m_curentDate + "/";
    return m_path + m_cbar + "-" + m_curentDate + "-" + m_faces[m_idFace] + ".jpg";
}
```

Ici, on initialise l'attribut **m_path** à part car il sera ce chemin sera utile pour enregistrer les fichiers texte contenants les modifications constatées. Ainsi, ce chemin pourra être réutiliser ailleurs dans le code.

La prochaine étape est d'être capable de faire défiler les trois autres photos.

Pour cela, les boutons suivants sur l'IHM seront utilisés :

Voici l'implémentation des slots :

```
void Compare::on_pbNEXT_clicked()
{
    m_idFace++;
    if(m_idFace > 3) {
        m_idFace = 0;
    } //if
    this->setImage(this->getImage());
}

void Compare::on_pbPREVIUS_clicked()
{
    m_idFace--;
    if(m_idFace < 0) {
        m_idFace = 3;
    } //if
    this->setImage(this->getImage());
}
```

Le code est simple : on vient incrémenter ou décrémenter l'attribut **m_idFace** pour passer à la face suivante ou précédente.

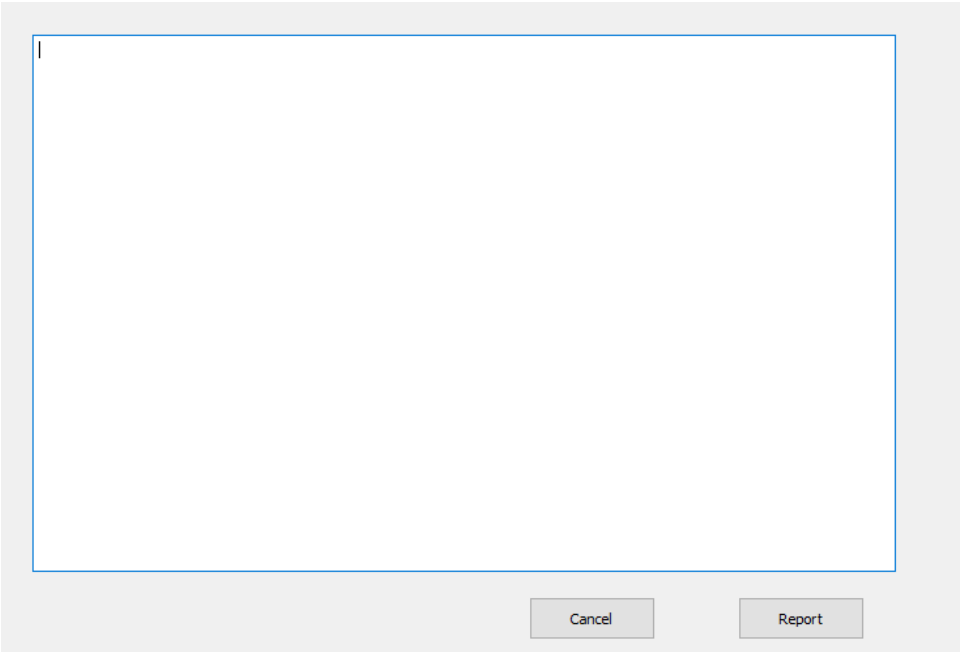
On contrôle aussi la valeur de l'attribut pour revenir à la première image lorsque l'on arrive à la dernière et inversement étant donné que l'on a que quatre images.

Enfin, on rappelle la méthode **setImage()**.

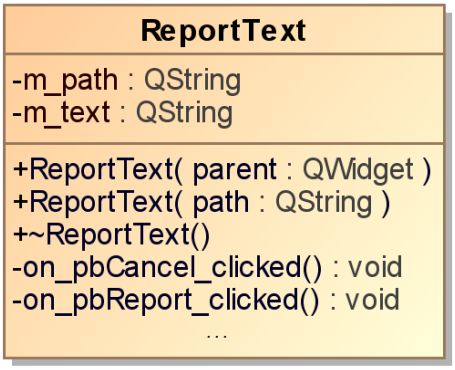
Il ne reste plus qu'à notifier les différences constatées. Lorsque l'opérateur clic sur le bouton permettant de notifier que l'on a trouvé une différence :

Nous souhaitons ouvrir une nouvelle fenêtre.

Pour cela, nous avons fait une nouvelle classe avec une nouvelle IHM :



Voici le diagramme de classe :

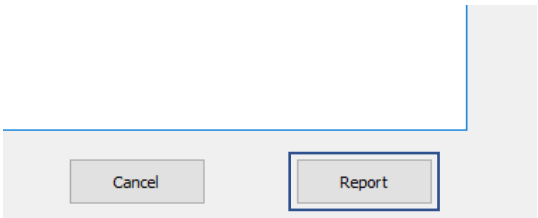


Fichier reporttext.h :

```
#ifndef REPORTTEXT_H
#define REPORTTEXT_H

#include <QMainWindow>

namespace Ui {
class ReportText;
}

class ReportText : public QMainWindow
{
    Q_OBJECT

public:
    explicit ReportText(QWidget *parent = nullptr);
    ReportText(QString);
    ~ReportText();
```

```
private:
    Ui::ReportText *ui;
    QString m_text;
    QString m_path;

private slots:
    void on_pbCancel_clicked();
    void on_pbReport_clicked();
};

#endif // REPORTTEXT_H
```

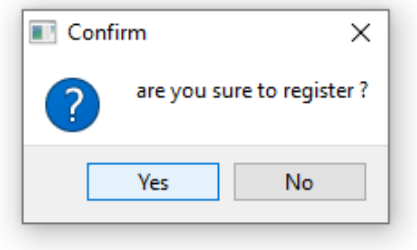
Il suffit de taper les modifications constatées dans la zone de texte. Lorsque le bouton permettant « **Report** » est cliqué :



Voici l'implémentation du bouton :

```
void ReportText::on_pbReport_clicked()
{
    if (ui->text->toPlainText() != "") {
        QMessageBox::StandardButton reply;
        reply = QMessageBox::question(this, "Confirm", "are you sure to register ?", QMessa
geBox::Yes|QMessageBox::No);

        if (reply == QMessageBox::Yes) {
            m_text = ui->text->toPlainText();
            QFile file(m_path + "report.txt");
            if (!file.open(QIODevice::WriteOnly | QIODevice::Text)) {
                return;
            }
            QTextStream out(&file);
            out << QDate::currentDate().toString(QString("dd-MM-
yyyy")) << " : " << endl << m_text << endl;
            this->close();
            delete this
          }
    }
}
```

Le bouton **Report** permet de demander confirmation, nous utilisons donc une **QMessageBox** qui est une classe permettant de générer des boites de dialogue :



Pour générer notre boite de dialogue, on utilise la méthode statique **question()** de la classe **QMessageBox** dont on stocke la valeur de retour (constante d'énumération représentant le bouton cliqué) dans une instance de l'énumération **StandardButton**.

Il suffit ensuite de tester la valeur de retour de la boite de dialogue :

Si le test est vrai, on récupère le texte de la **zone de texte** et on le stock dans l'attribut **m_text** : `m_text = ui->text->toPlainText();`

Pour générer et écrire un fichier, on utilise la classe **QFile.** Son constructeur prend le chemin du fichier que l'on souhaite créer / éditer :

```
QFile file(m_path + "report.txt");
```

Ensuite, on ouvre le fichier avec la méthode **open()** tout en testant le bon fonctionnement de l'opération :

```
if (!file.open(QIODevice::WriteOnly | QIODevice::Text)) {
   return;
} //if
```

Pour écrire dans le fichier, on utilise la classe **QTextStream** ce qui nous permet d'éditer le fichier en passant par des flux. Le constructeur de cette classe prend la référence d'un **objet QFile** : `QTextStream out(&file);`

Enfin, on utilise le l'opérateur de flux de sortie pour écrire dans le dossier :

```
out << QDate::currentDate().toString(QString("dd-MMyyyy")) << " : " << endl << m_text
<< endl;
```

On commence par écrire la fate courant et on écrit le texte entrer par l'opérateur à la ligne.

Lorsque l'opérateur confirme la note qu'il a entré (clic sur le bouton), on souhaite également fermer la fenêtre de comparaison.

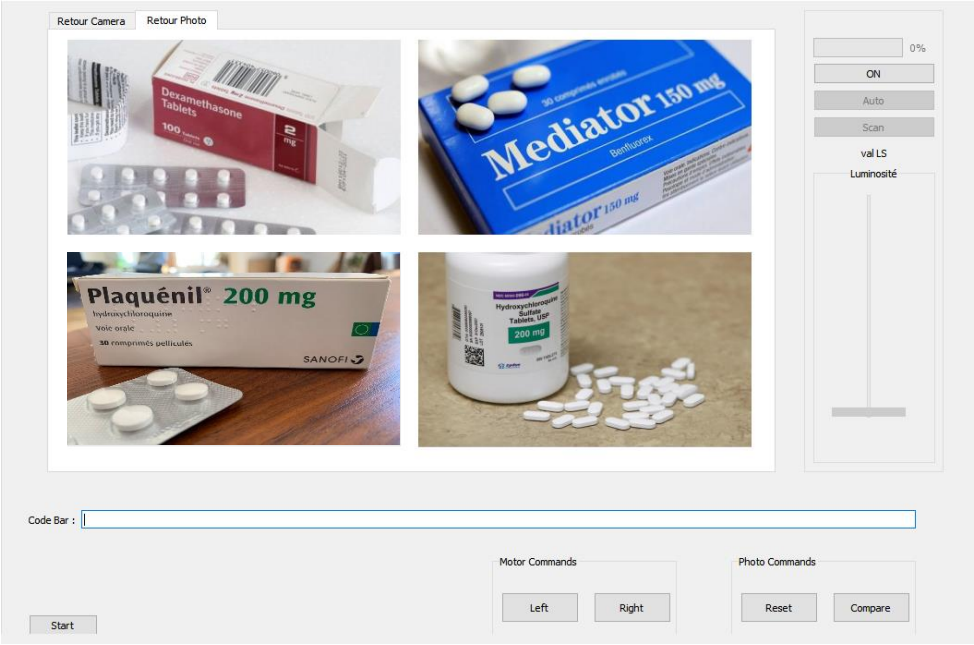
On vient donc appeler le slot **close()** propre à Qt, puis on supprime l'instance courante de la classe.

```
this->close();
delete this
```

En effet, fermer la fenêtre n'appelle pas le **destructeur** de la classe. Il faut donc forecer la suppression de l'instance pour libérer la mémoire étant donné que l'on peut instancier plusieurs fois la classe a chaque fois que l'on voudra notifier une différence.

Agrandissement des images :

Une fois le processus de prise de vues achevé, les images sont affichées sur l'IHM grâce à des **QLabel** :



Les images ci-dessus n'ont pas été prise par notre équipe de projet.

Cette partie a déjà été traitée dans la partie de « prises de vues » de l'étudiant 1.

Nous avons souhaité pouvoir agrandir les images pour avoir une meilleure vue de ces dernières lorsque l'on clique dessus.

La problématique est qu'il est impossible tel quel de pouvoir cliquer sur des **QLabel** étant donné que leur fonctionnalité première est d'afficher des informations.

Le travail ici consiste à résoudre ce problème :

Il est possible de plus ou moins réimplémenter la classe pour ajouter cette fonctionnalité.

Pour cela, il est nécessaire de créer une nouvelle classe qui **héritera** de **QLabel**.

Voici le diagramme de classe :

Fichier SPRIHM.h :

```
#ifndef CLICKABLELABEL_H
#define CLICKABLELABEL_H

#include <QLabel>
#include <QWidget>
#include <Qt>

/*
 * Classe ClickableLabel
 * Réimplemente la classe QLabel pour les rendre cliquables
 */

class ClickableLabel : public QLabel { //On hérite de QLabel

    Q_OBJECT

public:
    explicit ClickableLabel(QWidget* parent = Q_NULLPTR, Qt::WindowFlags f = Qt::WindowFlags());
    ~ClickableLabel();

signals:
    void clicked(); //creation d'un signal clicked()

protected:
    void mousePressEvent(QMouseEvent* event);

};

#endif // CLICKABLELABEL_H
```

On **hérite de QLabel** à la déclaration de la classe (voir les commentaires dans le code).

Cela permet de pouvoir accéder à toutes les méthodes de QLabel, et de pouvoir en rajouter ici.

Ce qui nous intéresse, c'est donc d'ajouter un signal **clicked()** qui sera émis lorsque l'on cliquera sur le **QLabel**.

Il faut maintenant faire en sorte de détecter le clic de la sourie. Pour cela, il existe une méthode dans Qt qu'il nous reste à réimplémenter. Il s'agit de la méthode suivante :

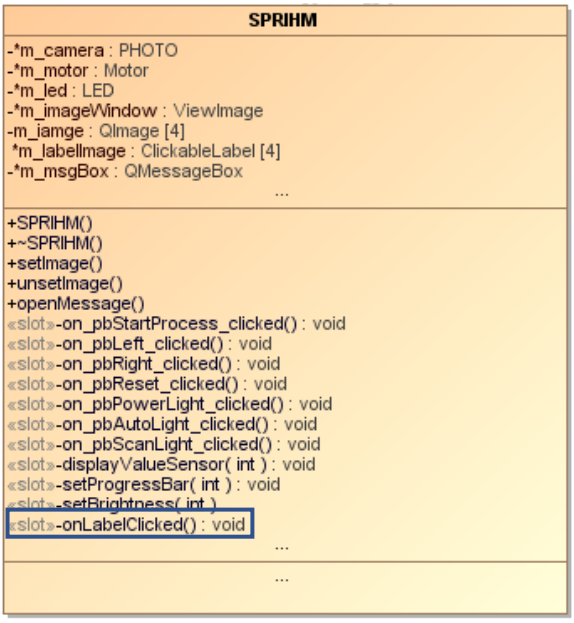
```
void mousePressEvent(QMouseEvent* event);
```

Voici son implémentation :

```
void ClickableLabel::mousePressEvent(QMouseEvent* event) {
    emit clicked();
}
```

On vient simplement **émettre** le signal **clicked()** que l'on a déclaré.

Il faut ensuite connecter manuellement ce signal à un slot de l'IHM, ou plutôt 4 étant donné que l'on a quatre labels différents :



Le slot onLabelClicked est en réalité présent 4 fois. Il n'apparait qu'une seule fois sur le diagramme pour simplifier la lecture.

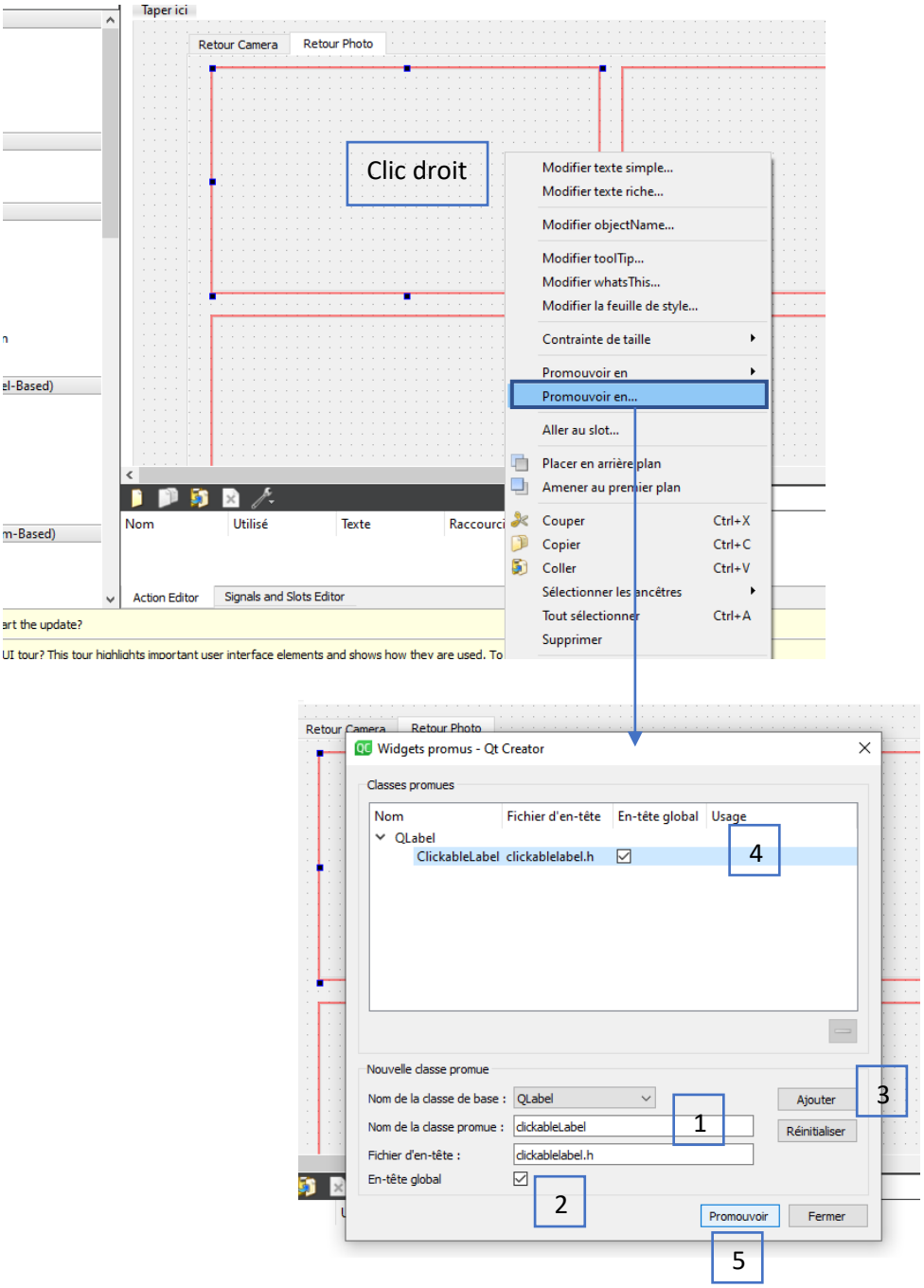
Voici la connexion du signal « **clicked** » de la classe **ClickableLabel**, au slot « **onLabelClicked** » de la classe **SPRIHM** :

```
    connect(m_labelImg[0], &ClickableLabel::clicked, this, &sprIHM::onLabel1Clicked);
    connect(m_labelImg[1], &ClickableLabel::clicked, this, &sprIHM::onLabel2Clicked);
    connect(m_labelImg[2], &ClickableLabel::clicked, this, &sprIHM::onLabel3Clicked);
    connect(m_labelImg[3], &ClickableLabel::clicked, this, &sprIHM::onLabel4Clicked);
```

Cette opération se fait depuis le **constructeur de la classe SPRIHM**. L'attribut **m_labelImg** est un tableau de type **ClickableLabel** de taille quatre pour contenir chacun des labels de l'interface graphique :

```
    m_labelImg[0] = ui->labPhoto_1;
    m_labelImg[1] = ui->labPhoto_2;
    m_labelImg[2] = ui->labPhoto_3;
    m_labelImg[3] = ui->labPhoto_4;
```

Pour que cela fonctionne, il faut changer le type des labels : **QLabel à ClickableLabel** depuis Qt-Desiner :



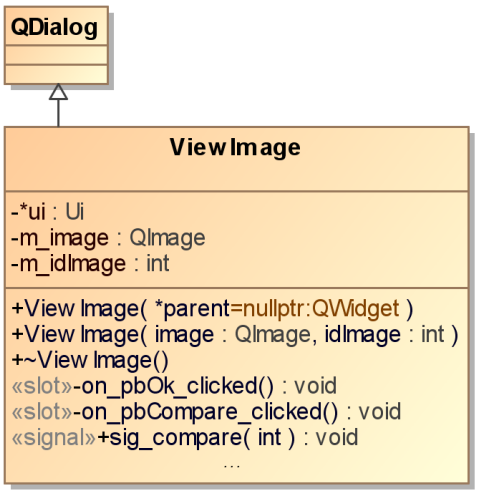
Suite à cela, les **QLabel** sont devenus des **ClickableLabel**.

La dernière étape est d'implémenter les slots **onLabelClicked()** que l'on a connecté aux signaux juste avant :

```
void sprIHM::onLabel1Clicked()
{
    qDebug() << m_image[0];
    m_imageWindow = new ViewImage(m_image[0], 0);
    connect(m_imageWindow, &ViewImage::sig_compare, this, &sprIHM::CompareOneImage);
    m_imageWindow->show();

}

void sprIHM::onLabel2Clicked()
{
    m_imageWindow = new ViewImage(m_image[1], 1);
    connect(m_imageWindow, &ViewImage::sig_compare, this, &sprIHM::CompareOneImage);
    m_imageWindow->show();
}

void sprIHM::onLabel3Clicked()
{
    m_imageWindow = new ViewImage(m_image[2], 3);
    connect(m_imageWindow, &ViewImage::sig_compare, this, &sprIHM::CompareOneImage);
    m_imageWindow->show();
}

void sprIHM::onLabel4Clicked()
{
    m_imageWindow = new ViewImage(m_image[3], 3);
    connect(m_imageWindow, &ViewImage::sig_compare, this, &sprIHM::CompareOneImage);
    m_imageWindow->show();
}
```

On instancie la classe **ViewImage** qui permet de commander une deuxième **IHM** (une nouvelle fenêtre) pour afficher les images en plus grandes. Il est nécessaire de faire l'instanciation ici pour avoir une nouvelle fenêtre lors de chaque clic. Le constructeur de la classe **SPRIHM** ne s'exécutant qu'une seule fois au démarrage du programme, on ne peut donc pas y instancier nos nouvelles fenêtres ici. Cela implique donc de gérer la libération de la mémoire, nous y reviendrons après.

Voici le diagramme de classe pour la deuxième IHM permettant d'afficher nos images en plus grand :



Fichier ViewImage.h :

```
#ifndef VIEWIMAGE_H
#define VIEWIMAGE_H

#include <QDialog>

/*
 * Classe ViewImage
 * Gère une deuxième fenêtre pour afficher les images
 */

namespace Ui {
class ViewImage;
}

class ViewImage : public QDialog
{
    Q_OBJECT

public:
    ViewImage(QWidget *parent = nullptr);
    ViewImage(QImage, int);
    ~ViewImage();

private slots:
    void on_pbOk_clicked();
    void on_pbCompare_clicked();

private:
    Ui::ViewImage *ui;
    QImage m_image;
    int m_idImage;

signals:
    void sig_compare(int);
};

#endif // VIEWIMAGE_H
```

Cette classe est plutôt simple. Il y a un constructeur surchargé pour pouvoir passer l'image à afficher dans la fenêtre ainsi que l'ID de l'image qui correspond à la face.

On stocke ensuite l'image dans l'attribut **m_image** et l'ID dans **m_idImage** :

```
ViewImage::ViewImage(QImage image, int idImage) :
    ui(new Ui::ViewImage),
    m_idImage(idImage),
    m_image(image)
{
    ui->setupUi(this);
    ui->label->setPixmap(QPixmap::fromImage(m_image));
    ui->label->setScaledContents(true);
}
```

Implémentation du constructeur surchargé

On utilise aussi le constructeur pour afficher l'image sur le **QLabel**.

La méthode `setScaledContents(true);` permet de **d'adapter la taille de l'image au QLabel**.

Voici le rendu sur l'IHM :

<u>À noter</u> : la dimension de la fenêtre est adaptée au ratio d'une photo prise par la caméra retenu pour le projet, cette dernière prenant de image avec la résolution : 2592x1944 px, le ratio est donc de **4:3**.

La résolution de la fenêtre étant de **820x615 px**, le ratio est respecté.

Le bouton **OK** de la fenêtre permet de fermer celle-ci, voici sont implémentation :

```
void ViewImage::on_pbOk_clicked()
{
    this->close();
    delete this;
}
```

En plus de fermer la fenêtre, on supprime l'instance courante de la classe (on supprime la fenêtre).

Cela est nécessaire car une utilisation prolongée de cette fonctionnalité (agrandissement des images) peut provoquer une perte de performance étant donné que l'on réinstancie la classe à chaque fois que l'on va cliquer sur les images. Il est donc important de libérer la mémoire.

Pour le bouton ce qui est du bouton **Compare**, on émet un signal :

```
void ViewImage::on_pbCompare_clicked()
{
    emit sig_compare(m_idImage);
}
```

Ce signal est ensuite connecter dans la classe principale **SPRIHM** au moment de l'instanciation de la classe **viewImage()**, lorsque l'on va cliquer sur les label :

```
void sprIHM::onLabel1Clicked()
{
    if(ui->labPhoto_1->pixmap() != 0x0) {
        m_imageWindow = new ViewImage(m_image[0], 0);
        connect(m_imageWindow, &ViewImage::sig_compare, this, &sprIHM::CompareOneImage);
        m_imageWindow->show();
    }
}
```

On appelle alors le slot **CompareOneImage()** permettant d'instancier la classe **Compare** en choisissant l'image que l'on veut comparer en donnant l'identifiant de sa face en argument :

```
void sprIHM::CompareOneImage(int id)
{
    m_idImage = id;
    m_compareWindow = new Compare(m_cbar, m_idImage);
    m_compareWindow->show();
}
```

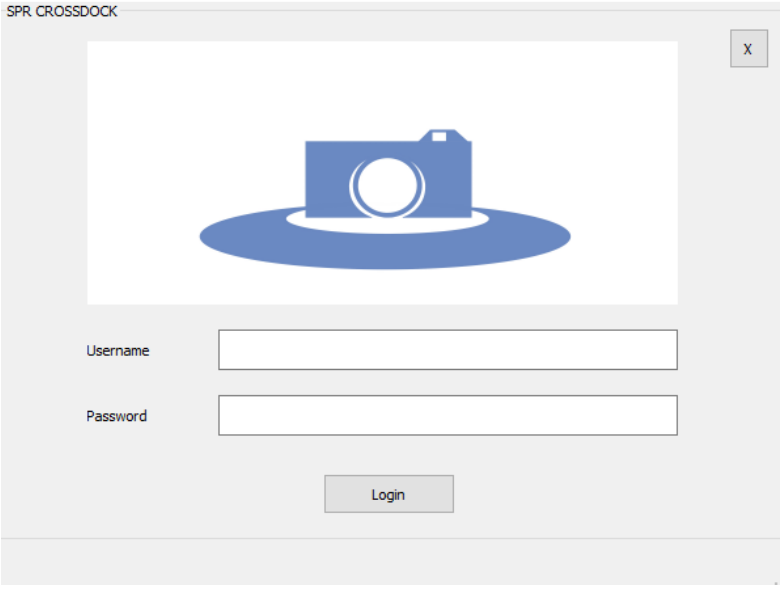
Système d'authentification :

Dans cette partie, nous allons voir la mise en place d'un système d'authentification au lancement de l'application.

L'objectif est de restreindre et de sécuriser l'accès au système. Avant chaque utilisation l'opérateur devra donc s'authentifier.

Le système d'authentification est composé d'une interface qui lui est propre ainsi qu'une classe qui lui est dédié.

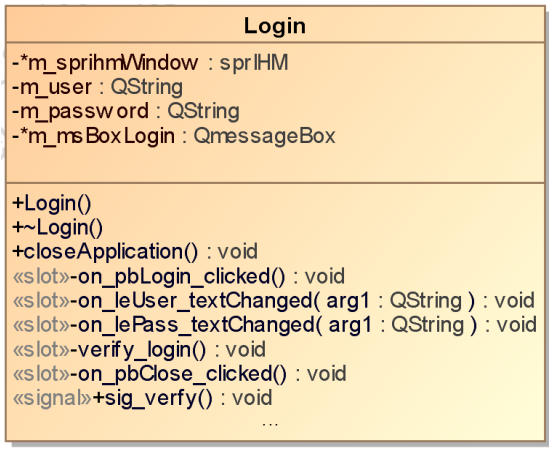
Voici l'**interface** de la classe **Login** :



Elle se compose de deux zones de texte « **Username** » et « **Password** », où l'utilisateur pourra rentrer son identifiant et son mot de passe. Celui-ci clique ensuite sur le bouton « Login » est l'interface principale de l'application se lance alors.

Classe Login.h :

Voici le diagramme de la classe Login :



Fichier Login.h :

```
#ifndef LOGIN_H
#define LOGIN_H

#include <QWidget>
#include <QMessageBox>
#include "sprihm.h"

namespace Ui {
class Login;
}

class Login : public QWidget
{
    Q_OBJECT

public:
    explicit Login(QWidget *parent = nullptr);
    ~Login();

    void closeApplication();

private slots:
    void on_pbLogin_clicked();
    void on_leUser_textChanged(const QString &arg1);
    void on_lePass_textChanged(const QString &arg1);
    void verify_login();
    void on_pbClose_clicked();

private:
    Ui::Login *ui;
    sprIHM *m_sprihmWindow;

    QString m_user;
    QString m_password;

signals:
    void sig_verify();
};

#endif // LOGIN_H
```

Cette classe permet donc d'entrer ses identifiants, de vérifier si les informations sont correctes et d'autoriser la connexion.

Par défaut, le bouton **Login (**permettant de vérifier ses identifiants de connexion) est grisé.

Pour le rendre cliquable, il faut entrer le nom d'utilisateur et le mot de passe correct.

On commence par détecter lorsque qu'un caractère est entré dans l'un des deux champs :

```
void Login::on_leUser_textChanged(const QString &arg1)
{
    this->verify_login();
}

void Login::on_lePass_textChanged(const QString &arg1)
{
    this->verify_login();
}
```

On utilise le slot les **textChanged()** appartenant à la classe **QLineEdit** (qui nous sert à faire les zones de texte).

Ainsi, lorsque l'on détecte qu'un caractère est entré, on appelle la méthode **verfy_login()** qui va vérifiée si que les identifiants correspondes à ceux attendus :

```
void Login::verify_login()
{
    m_user = ui->leUser->text();
    m_password = ui->lePass->text();

    if ((m_user == "user" && m_password == "user") || (m_user == "admin" && m_pass
word == "admin")) {
        ui->pbLogin->setEnabled(true);
    } else {
        ui->pbLogin->setEnabled(false);
    } //else
}
```

On stock le nom d'utilisateur et le mot de passe dans les attributs **m_user et m_password** que l'on va venir tester avec des valeurs en **dures** dans le code. Le bouton sera alors activé ou désactivé en fonction du test.

A noter : Cette fonctionnalité à pour but de mettre en pratique un système d'authentification avec comme nom d'utilisateur **user** et mot de passe **user** ou **admin** et **admin**. Le code de cette méthode est prévu pour évoluer car l'authentification n'est actuellement pas sécurisée.

Lorsque le bouton login est cliquable et qu'on l'active, on appelle slot suivant :

```
void Login::on_pbLogin_clicked()
{
    m_user = "";
    m_password = "";

    ui->leUser->setText("");
    ui->lePass->setText("");

    m_sprihmWindow = new sprIHM;
    m_sprihmWindow->show();
}
```

On réinitialise les attributs contenants les identifiants, on réinitialise les zones de textes et enfin, on instancie l'IHM principale et on l'affiche.

Enfin, une méthode permettant de quitter l'application a également été mise en place.

Fichier Login.cpp :

```
void Login::closeApplication()
{
    QMessageBox::StandardButton result;
    result = QMessageBox::question(this, "Confirm", "do you want to exit?", QMessa
geBox::Yes|QMessageBox::No);

    switch (result) {
        case QMessageBox::Yes :
            this->close();
            break;

        case QMessageBox::No :
            break;
    }  // switch
}
```

Suite à cela, il faudra également adapter le fichier **main.cpp** pour exécuter l'IHM d'authentification et la principale :

```
#include "login.h"

#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Login w;
    w.showMaximized();
    return a.exec();
}
```

Diagramme de classe complet :