RoboLyon Docs

Version 1.0

Robo'Lyon

25 août, 2025

Premiers pas avec WpiLib

CHAPITRE 1

Version : 25 août, 2025

Bienvenue dans ce tutoriel écrit par l'équipe 5553, Robo'Lyon, de France.

Son but est de permettre aux rookies de s'initier à la programmation d'un robot FRC en C++ avec la librairie WpiLib.

> **Attention :** Si vous n'êtes pas encore à l'aise avec le C++ et/ou la programmation orientée objet, ne commencez pas ce tutoriel. Retournez plutôt vers un cours de C++ comme celui d'"openclassroom.

Toute idée d'amélioration est la bienvenue. Pour en proposer, créez une issue dans le projet github ou contactez l'équipe.

Si vous voulez contribuer à ce cours, vous pouvez soumettre une pull request sur le projet github du cours. Les sources sont dans le dossier `source/docs` et sont écrites en reStructuredText. C'est un langage de balisage très facile à prendre en main, voici un guide pour apprendre sa syntaxe.

CHAPITRE 2

Sommaire

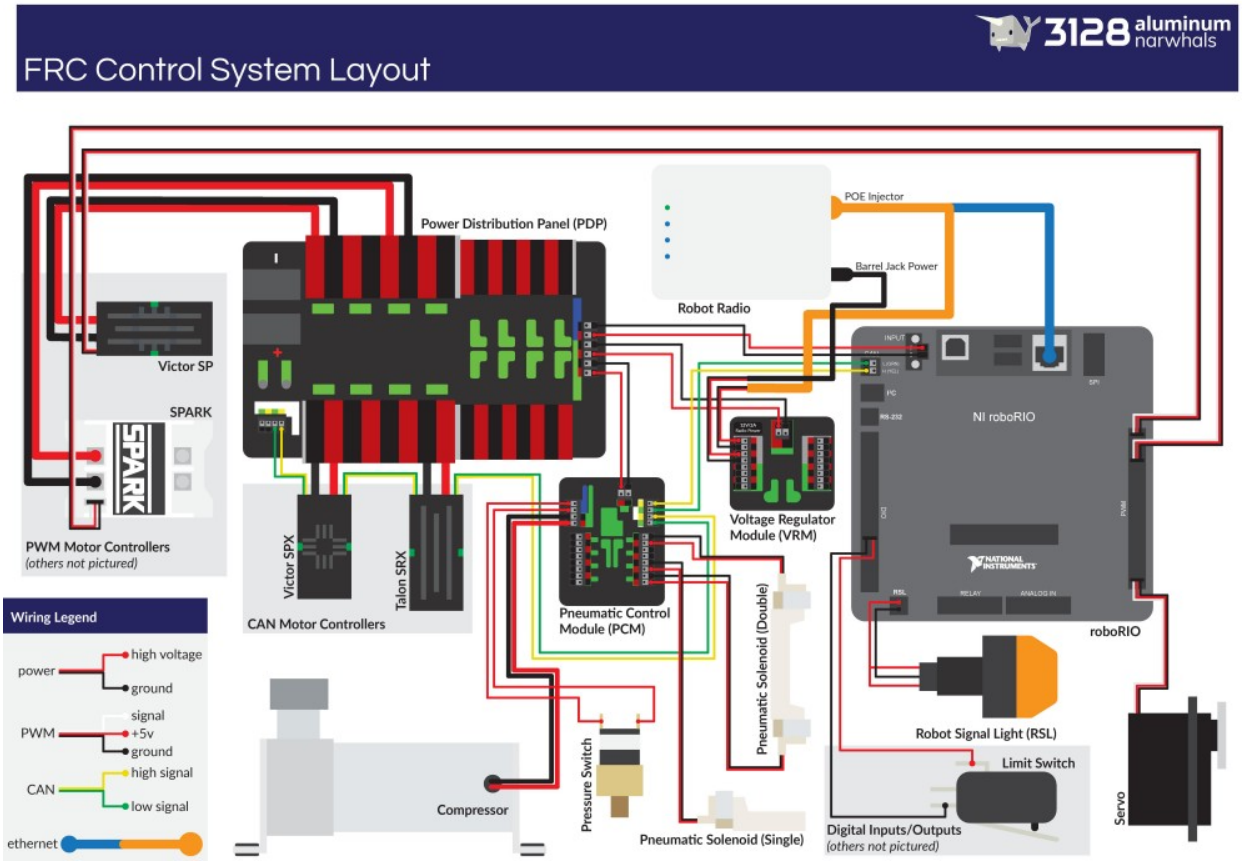
2.1 WpiLib, c'est quoi ?

2.1.1 Comment contrôle-t-on un robot ?

Le robot est contrôlé par le RoboRio, le cerveau du robot. C'est lui que nous allons programmer. Il communique avec de nombreux composants du robot pour, par exemple, envoyer des ordres aux moteurs et aux vérins. Il reçoit aussi les informations venant des capteurs du robot (encodeurs, ultrasons, limit switch).

Le RoboRio est connecté par ethernet à la borne wifi du robot. Il peut ainsi communiquer par wifi avec l'ordinateur du pilote.

Voici un schéma des branchements du robot pour mieux comprendre :



Sur l'ordinateur du pilote, c'est le programme appelé `FRC Driver Station` qui permet de communiquer avec le robot. Celui-ci permet au pilote de contrôler le robot avec son joystick.

2.1.2 Et WpiLib dans tout ça ?

Présentation

WpiLib (Worcester Polytechnic Institute Robotics Library) est une librairie qui permet la programmation du RoboRio. C'est un ensemble de classes qui permettent l'interface entre le logiciel et le hardware.

Cette librairie nous fournit des classes pour gérer les moteurs, la pneumatique, les capteurs et à peu près tout ce dont nous avons besoin pour contrôler un robot. C'est une librairie de haut-niveau, c'est à dire qu'elle nous permet de

manipuler facilement des objets complexes.

WpiLib est disponible en C++ et en Java. Il existe aussi des versions non-officiellement supportées en Python ou en Kotlin.

Installation

Pour programmer le robot en C++ et en Java, WpiLib propose d'utiliser Visual Studio Code qui est l'IDE officiellement supporté. Pour installer l'environnement de programmation, un installer Windows est disponible. Les instructions sont à retrouver ici.

VS Code permet l'utilisation des classes de WpiLib mais aussi de déployer le programme sur le robot.

Pour contrôler le robot, il faut aussi avoir la Driver Station. Voici les instructions pour l'installer l'installer.

2.1.3 Documentation

Pour obtenir des informations sur la librairie, savoir à quoi sert quelle classe, quelles fonctions sont disponibles, ect..., une documentation est disponible. Chaque classe y est répertoriée avec la description de chacune des méthodes qu'elle propose.

2.2 Notre premier programme

2.2.1 Créer un nouveau projet sur VS Code

Pour créer un nouveau projet, cliquez sur l'icône WPILib puis sur `WPILib: Create a new project`. Une page s'ouvre, choisissez `Template`, `c++` et `TimedRobot Skeleton (Advanced)`. Remplissez les autres champs et appuyez sur `Generate Project` : c'est fait !

2.2.2 A quoi ressemble un programme de robot ?

Le programme du robot créé est séparé en 2 fichiers `src/main/include/Robot.h` et `src/main/cpp/Robot.cpp`. Voici à quoi le fichier `Robot.h` ressemble :

```
#include <frc/TimedRobot.h>

class Robot : public frc::TimedRobot
{
public:
  void RobotInit() override;

  void AutonomousInit() override;
  void AutonomousPeriodic() override;

  void TeleopInit() override;
  void TeleopPeriodic() override;

  void TestInit() override;
  void TestPeriodic() override;
};
```

Mais où est le main() ?

En effet, le programme du robot est différent de ce que l'on a l'habitude de voir. Au lieu d'avoir un `int main()` et beaucoup de fonctions, le programme est une classe. Bien sûr, il existe quelque par un main ; mais il ne change jamais et il est déjà écrit pour nous.

Le programme du robot va donc lire la classe que nous allons coder.

Lisons le programme ligne par ligne

`#include <frc/TimedRobot.h>` : Cette ligne inclut le fichier « frc/TimedRobot.h »

`class Robot : public frc::TimedRobot` : Notre classe s'appelle `Robot` et qu'elle hérite d'une autre classe appelée `TimedRobot`. TimedRobot a été inclus précédemment, cette classe fait partie de la librairie WpiLib.

Ensuite, on voit que notre classe `Robot` possède plusieurs méthodes : `RobotInit()`, `AutonomousInit()`, `AutonomousPeriodic()`, ect... Ce sont ces méthodes que nous allons coder pour programmer le robot.

Toutes ces déclarations de méthode sont suivies du mot-clé `override`. Petite explication : ces méthodes sont en fait héritées de la classe mère `TimedRobot`. Cela permet à n'importe quelle programme d'appeler ces méthodes en étant sûr qu'elles existent. Par défaut, ces méthodes sont vides et ne font rien. C'est pourquoi on peut les `override` : ce sera ainsi notre version de la méthode qui sera appelée au lieu de la version vide du `TimedRobot`.

2.2.3 Les différentes méthodes

Le `TimedRobot` nous propose donc une structure pour coder le robot qui gère les transitions entre les états du robot et les boucles dans ces états. Pour chaque état (autonomous, teleop, disabled, test), deux méthodes sont appelées :

- Init : cette méthode est appelée chaque fois que l'on entre dans l'état correspondant (par exemple si l'on passe de Disable à Teleop, la méthode `TeleopInit()` sera appelée une fois)
- Periodic : cette méthode est appelée toutes les 20ms quand on est dans l'état correspondant (par exemple si l'on est en Teleop, la méthode `TeleopPeriodic()` sera appelée un fois toutes les 20ms).

Pour comprendre le fonctionnement du robot, essayez d'afficher un message pour chaque méthode. Les sorties `cout` sont redirigées vers le réseau et on peut les lire grâce à Riolog ou sur la Driver Station.

> **Attention :** Comme ces méthodes sont appelées très fréquemment, il ne faut pas y écrire du code trop long à s'executer. Sinon, cela bloque le programme et pose des problèmes. Les boucles `while`, `do .. while` et `for` sont donc en général à éviter. On utilisera à la place de celles-ci des `if` qui seront appelés régulièrement.

2.2.4 Utiliser WpiLib

Pour l'instant, on utilise une seule classe fournie par WpiLib, la classe `TimedRobot`. Elle est incluse grâce à la ligne :

```
#include <frc/TimedRobot.h>
```

Mais pour programmer le robot nous allons avoir besoin des autres classes WpiLib pour controller les moteurs, les capteurs, ... Une façon très rapide d'inclure toutes ces classes à la fois est d'écrire cette ligne au début de notre programme :

```
#include <frc/WPILib.h>
```

Vous pouvez jeter un coup d'oeil à l'intérieur de ce ficher. Il inclut en fait à son tour toutes les classes de WpiLib (dont TimedRobot).

```
#include "frc/ADXL345_I2C.h"
#include "frc/ADXL345_SPI.h"
#include "frc/ADXL362.h"
#include "frc/ADXRS450_Gyro.h"
#include "frc/AnalogAccelerometer.h"
#include "frc/AnalogGyro.h"
#include "frc/AnalogInput.h"
#include "frc/AnalogOutput.h"
#include "frc/AnalogPotentiometer.h"
#include "frc/AnalogTrigger.h"
#include "frc/AnalogTriggerOutput.h"
#include "frc/BuiltInAccelerometer.h"
#include "frc/Compressor.h"
#include "frc/ControllerPower.h"
#include "frc/Counter.h"
#include "frc/DMC60.h"
#include "frc/DigitalInput.h"
#include "frc/DigitalOutput.h"
........
......
```

2.3 Contrôler un moteur

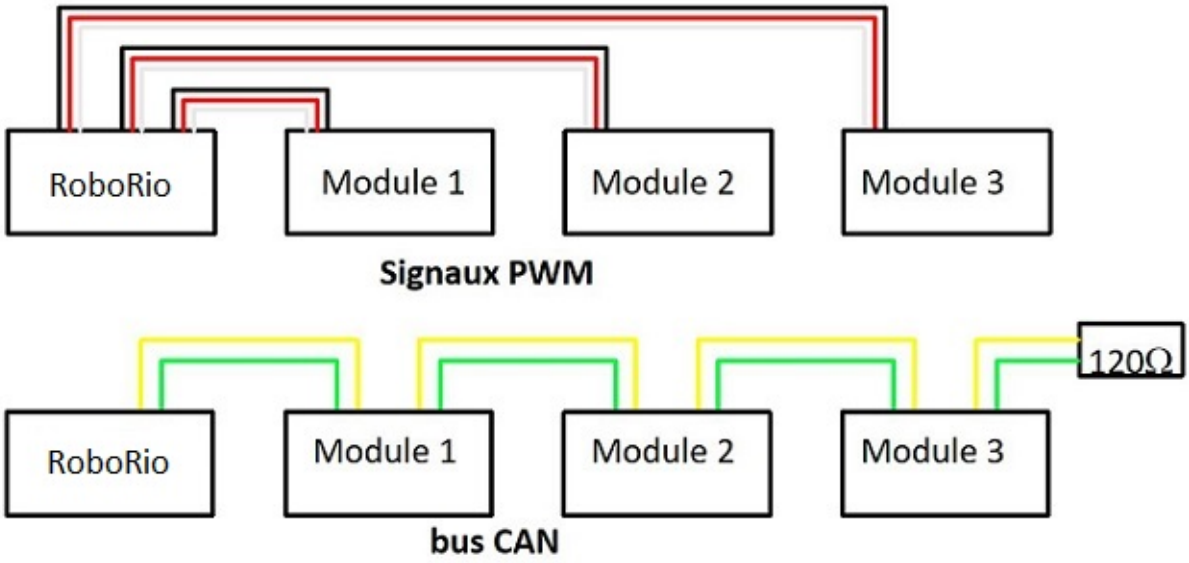
2.3.1 Les contrôleurs moteur

Pour contrôler les moteurs présents sur le robot, nous avons besoin de contrôleurs moteurs. En un mot, ceux-ci reçoivent un signal de faible intensité de la part du RoboRio et envoient au moteur un signal de plus forte intensité. Voici quelques exemples de contrôleurs moteur que nous utilisons : VictorSP, Spark et SparkMax.

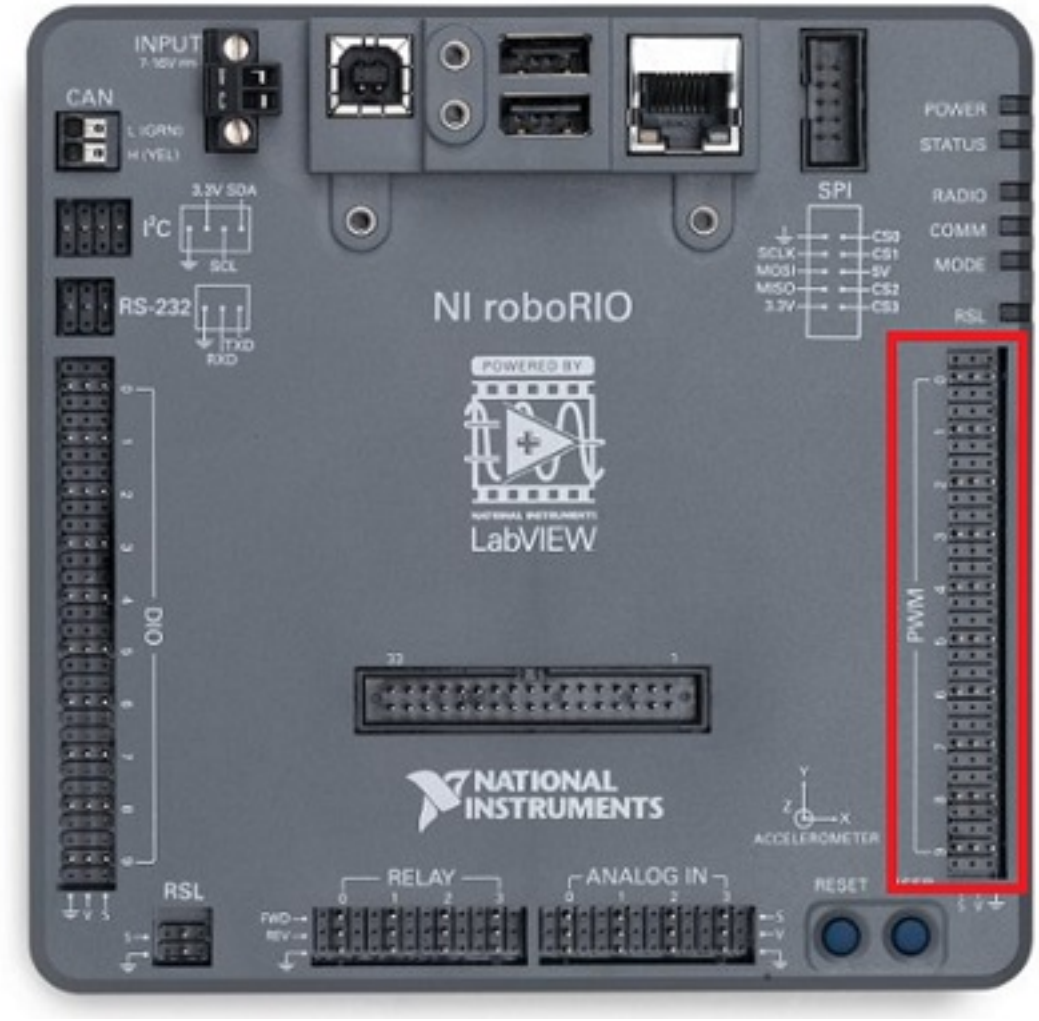
2.3.2 PWM et CAN

Il y a 2 manières de contrôler ceux-ci : en PWM ou en CAN. Tous les contrôleurs supportent le PWM mais seulement une partie d'entre eux supporte le CAN.

La principale différence entre les deux modes de transmission est que le PWM ne peut commander que la vitesse du moteur tandis que le CAN permet des contrôles plus avancés et la communication d'informations entre le contrôleur et le RoboRio.



La méthode la plus simple est le PWM. Elle nous permet de contrôler rapidement le moteur voulu en branchant le cable du contrôleur sur le bon port PWM du RoboRio (encadrés en rouge).



2.3.3 Dans le Code

Du côté du RoboRio, il nous suffit de créer un objet correspondant au contrôleur pour pouvoir asservir le moteur. WpiLib propose une classe pour chaque contrôleur. En fait, avec le PWM, ces classes sont toutes presque identiques car elles dérivent toutes de la même classe PWMSpeedController. Cependant, elles ont été dérivées sous plusieurs noms :

```
#include <frc/VictorSP.h>
#include <frc/Spark.h>
#include <frc/PWMVictorSPX.h>
```

Remarquez que `VictorSPX` est précédé de `PWM`, c'est parce qu'il peut être contrôlé via le CAN. Pour différencier les 2 classes (qui sont totalement différentes, celui-ci se nomme donc `PWMVictorSPX`.

Quand on crée un objet qui représente un contrôleur PWM, on doit spécifier dans le constructeur le port sur lequel il est branché. Par exemple, pour un VictorSP branché sur le port n°0 :

```
frc::VictorSP mon_moteur(0);
```

On a ensuite accès à tout un tas de méthodes qui nous permettent de contrôler le moteur.

Pour « donner » une puissance voulue à un moteur, on utilise la méthode : `void Set(double value)` qui attend en argument un double entre -1 (pleine puissance vers l'arrière) et 1 (pleine puissance vers l'avant). Si je veux faire tourner mon moteur à la moitié de sa vitesse maximum :

```
mon_moteur.Set(0.5);
```

On a aussi `void StopMotor()` qui remplace `Set(0)` et `void SetInverted(bool isInverted)` qui inverse la direction du moteur si on lui donne comme argument `true`.

```
mon_moteur.SetInverted(true);

mon_moteur.StopMotor();
```

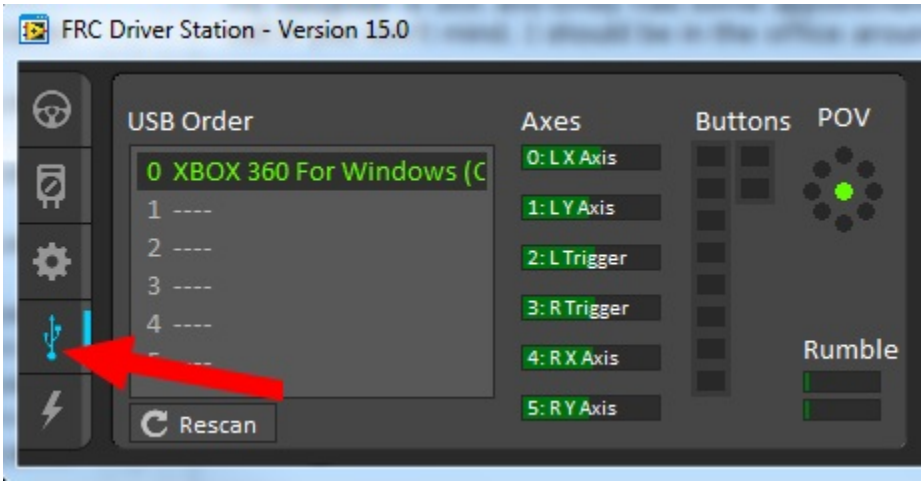
2.4 Lire les entrées d'un joystick

2.4.1 Le Joystick

Pour permettre au pilote de contrôler le robot, celui-ci utilise un joystick. Nous pouvons aussi utiliser une manette de Xbox.

Le joystick que nous utilisons est le Logitech 3D Pro. Il possède 3 axes (avant/arrière, gauche/droite et twist), 12 boutons, un POV (bouton avec 9 positions possibles) et un throttle (manette type avion).

On peut connaître l'état de chaque bouton/axe du joystick en temps réel sur la Driver Station :



2.4.2 Dans le Code

WpiLib fournit une classe Joystick qui nous permet de récupérer les infos de celui-ci. Son constructeur attend en argument le numéro du joystick. Les numéros sont attribués selon l'ordre de branchement : le 1er joystick sera le 0, le 2ème le 1, ect … Si on a un seul joystick, il aura donc pour numéro 0 :

```
#include <frc/Joystick.h>

frc::Joystick mon_joystick(0);
```

Pour récupérer l'état d'un bouton (appuyé/relâché, on peut utiliser la méthode `bool GetRawButton(int button)` qui attend en argument le numéro du bouton et qui renvoi `true` si le bouton est appuyé et `false` si il est relâché.

```
// Récupère l'état de la gâchette (trigger)
bool gachetteAppuyee = mon_joystick.GetRawButton(1);
```

Pour récupérer la position d'un axe entre -1 et 1, on peut utiliser les méthodes `double GetX()`, `double GetY()`, `double GetZ()` (ou `GetTwist()`) et `double GetThrottle()`.

```
// Récupère l'état de chaque axe
double x = mon_joystick.GetX();
double y = mon_joystick.GetY();
double twist = mon_joystick.GetTwist();
double throttle = mon_joystick.GetThrottle();
```

Note : Pour adoucir les valeurs du joystick, il est possible d'ajouter une deadband : une zone dans laquelle les valeur sont rejetées car trop petites. Pour en apprendre plus sur les deadbands, voici un site trés intéressant.

On a aussi la méthode `int GetPOV()` qui renvoi l'angle formé par le POV ou -1 si il est situé au centre.

```
int pov = mon_joystick.GetPOV();

switch (pov)
{
    case -1:
        // Centre
        break;

    case 0:
        // Haut
        break;

    case 180:
        // Bas
        break;

    case 90:
        // Droite
        break;

    case 270:
        // Gauche
```

(suite sur la page suivante)

(suite de la page précédente)

```
        break;
}
```

2.5 1er Défi : Contrôler un moteur grâce au joystick

Coder la méthode TeleopPeriodic d'un TimedRobot pour répondre à ces objectifs :

- Quand le joystick est proche de zéro (entre -0.2 et 0.2), le moteur ne tourne pas.
- Sinon, le moteur tourne à une vitesse proportionnelle à la position du joystick
- Quand la gâchette (bouton 1) du joystick est appuyée, le moteur tourne pas

Normalement, votre programme sera séparé en 2 fichiers différents : Robot.h et Robot.cpp. Ici, le programme est dans un seul fichier pour plus de simplicité :

```
#include <frc/TimedRobot.h>
#include <frc/VictorSP.h>
#include <frc/Joystick.h>

class Robot : public frc::TimedRobot
{
public:
    void TeleopPeriodic() override
    {
        if(m_Joystick.GetButton(1))
        {
            m_Moteur.Set(0);
        }
        else
        {
            double y = m_Joystick.GetY();

            if(y < 0.2 && y > -0.2)
            {
                y = 0;
            }

            m_Moteur.Set(y);
        }
    }

private:
    frc::Joystick m_Joystick(0);
    frc::VictorSP m_Moteur(0);
};
```

2.6 Utiliser les capteurs

Maintenant que l'on sait faire bouger notre robot, faisons le bouger intelligemment. Mais cela n'est pas possible si nous n'avons pas d'indications sur l'état du robot. C'est pourquoi nous utilisons des capteurs.

2.6.1 Limit switch

Description

L'un des types de capteur les plus simple à utiliser est le limit switch. C'est un interrupteur qui revient à sa place quand il n'est pas pressé (un bouton poussoir).

Ce capteur nous permet de savoir si un mécanisme a atteint un certain point. Par exemple, un limit switch situé au bout d'un bras pivotant peut nous dire si le bras touche le bord du robot.

Un limit switch se branche aux ports DIO du Roborio grâce à deux cables. Un va sur le ground () et l'autre sur le signal (S). Pourtant, le switch possède souvent 3 connecteurs : Normally Open (NO), Normally Closed (NC), et ground (C ou COM). Normally Open signifie que le switch est normalement non-pressé. Quand il sera pressé, un message sera envoyé au Roborio. Normally Closed signifie le contraire. Connectez un des cables au connecteur NO ou NC, et l'autre au ground.

Dans le Code

Pour programmer un limit switch, créez une instance de la classe DigitalInput, son constructeur attend comme argument le port DIO sur lequel le switch est branché :

```
#include <frc/DigitalInput.h>
frc::DigitalInput mon_switch(0);
```

La méthode `bool Get()` renvoie *true* ou *false* suivant la position du switch (appuyé/relâché) et ses branchements (NO/NC) :

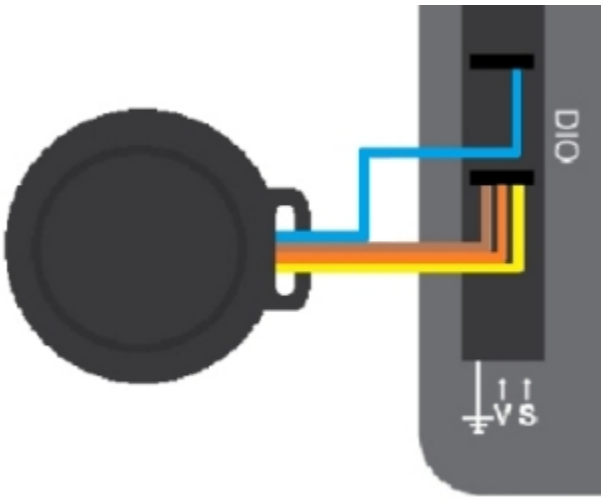
```
bool switchPresse = mon_switch.Get();
```

2.6.2 Encodeurs

Description

Les encodeurs permettent de connaître la position précise d'un mécanisme. Ils se fixent sur des axes et comptent le nombre de tours que ceux-ci font. Ce sont des sortes de compteurs : quand l'axe tourne dans un sens la « position » augmente, quand il tourne dans le sens inverse la « position » diminue. Les encodeurs peuvent être optiques ou bien magnétiques.

Voici la façon dont il se branche :



Dans le Code

Pour programmer un encodeur, créez une instance de la classe Encoder, son constructeur attend comme argument les port DIO sur lesquels l'encodeur est branché :

```
#include <frc/Encoder.h>
frc::Encoder mon_encodeur(0, 1);
```

La méthode `int Get()` renvoie la distance angulaire mesurée par l'encodeur en ticks. Selon le modèle, un tour équivaut à 360 ticks, 22 ticks, ... :

```
double distance = mon_encodeur.Get();
```

La méthode `void Reset()` remet le compteur à zéro :

```
mon_encodeur.Reset();
```

Les méthodes `void SetDistancePerPulse(double distancePerPulse)` et `double GetDistance()` permettent de convertir automatiquement les tick en une autre unité :

```
// 1 tour équivaut à 360 ticks
mon_encodeur.SetDistancePerPulse(1.0/360);
double nombre_de_tours = mon_encodeur.GetDistance();
```

La méthode `void GetRate()` renvoie la vitesse actuelle convertie en distance selon le facteur de conversion (1 par défaut) :

```
double vitesse = mon_encodeur.GetRate();
```

2.6.3 Gyroscopes

Description

Les gyroscopes permettent de connaître la vitesse et le sens de rotation du robot. Ils permettent aussi de connaître l'angle du robot sur le terrain. Ils peuvent se brancher sur le port SPI ou les ports Analog In 0 et 1 du Roborio.

Dans le Code

Pour programmer un gyroscope, créez une instance de la classe ADXRS450_Gyro (SPI) ou AnalogGyro (Analog In) en fonction du gyroscope. Le constructeur d'*AnalogGyro* attend comme argument le port Analog In (0 ou 1) sur lequel le gyroscope est branché :

```
#include <frc/ADXRS450_Gyro.h>
frc::ADXRS450_Gyro mon_gyro();
```

```
#include <frc/AnalogGyro.h>
frc::AnalogGyro mon_gyro(0);
```

La méthode `double GetAngle()` renvoie l'angle du robot en degrés dans le sens des aiguilles d'une montre :

```
double angle = mon_gyro.GetAngle();
```

La méthode `double GetRate()` renvoie la vitesse de rotation du robot en degrés par secondes dans le sens des aiguilles d'une montre :

```
double vitesse_rotation = mon_gyro.GetRate();
```

La méthode `void Calibrate()` calibre le gyroscope en définissant son centre. La méthode `void Reset()` remet le gyroscope à zéro :

```
// Initialisation du gyro
mon_gyro.Calibrate();
mon_gyro.Reset();
```

2.7 Le Contrôleur PID

2.7.1 Introduction

Quand on veut que le robot atteigne une position précise, suive une trajectoire, ou maintienne une vitesse constante, il ne suffit pas de lui dire « avance » ou « tourne ». Il faut contrôler *comment* il s'en approche, et avec quelle précision. C'est là qu'intervient le **PID**, un algorithme de régulation très utilisé dans l'industrie, en aéronautique… et bien sûr, en robotique.

2.7.2 Qu'est-ce qu'un PID ?

Le terme **PID** est l'acronyme de **Proportionnel – Intégral – Dérivé**. C'est un type de contrôleur (régulateur), c'est-à-dire un système qui ajuste une commande (ex. vitesse moteur) pour atteindre une consigne (ex. position cible). Il compare en permanence la position actuelle du système avec la cible, et corrige l'écart intelligemment.

Ce type de régulateur a été formalisé dans les années 1920 par Nicolas Minorsky, à l'origine pour contrôler des systèmes industriels (vannes, moteurs, etc.) et la navigation maritime. Sa simplicité, sa robustesse et son efficacité en font encore aujourd'hui une solution de référence dans de très nombreux domaines techniques et industriels.

2.7.3 À quoi sert un PID ?

Un PID sert à **réduire une erreur**, c'est-à-dire l'écart entre une mesure (position réelle, vitesse, etc.) et une consigne (autrement dit un objectif tel qu'une position désirée, une vitesse cible, etc.) comme ceci :

$$\text{erreur} = \text{consigne} - \text{mesure}$$

Le PID agit sur une commande (par exemple, la puissance envoyée à un moteur) pour diminuer cet écart de façon stable, performante et réactive.

Exemple courant :

Tu veux que ton élévateur atteigne la position 1.2 m. Sans PID, tu pourrais simplement « mettre de la puissance », mais :

- S'il y a trop de puissance, il dépasse la cible.
- Pas assez ? Il n'y arrive jamais.
- Un changement de poids ou de friction ? Il se dérègle.

Un PID ajuste la commande à chaque instant pour que l'élévateur atteigne exactement 1.2 m, sans oscillations ni erreur finale.

2.7.4 Le terme Proportionnel (P)

Introduction

Le **terme proportionnel** est la partie la plus simple du PID. Il applique une correction proportionnelle à l'erreur actuelle : plus l'erreur est grande, plus la correction est forte et rapide. C'est comme si tu disais « plus je suis loin de la cible, plus j'accélère ». Et si tu es proche, la correction est faible.

Formule

$$\text{Commande}_P = k_P \times \text{erreur}$$

— kP est un **coefficient fixe** que tu choisis.
— L'**erreur** est la différence entre la consigne et la mesure.

Exemple

Si ton élévateur est à 0.8 m, et que tu veux aller à 1.2 m :
— erreur = 1.2 - 0.8 = 0.4 m
— si kP = 5.0, la correction sera : 5.0 × 0.4 = 2.0 (par exemple, 2.0 V envoyés au moteur)

Code minimaliste

C++

```
double setpoint = 1.2;     // Position cible en mètres
double current = getPosition(); // Position mesurée
double error = setpoint - current;

double kP = 5.0;
double output = kP * error;

setMotorVoltage(output);
```

Java

```
 double setpoint = 1.2;     // Position cible en mètres
 double current = getPosition(); // Position mesurée
 double error = setpoint - current;

double kP = 5.0;
double output = kP * error;

setMotorVoltage(output);
```

Note : Ce code ne gère que le P. Il est simple, mais peut suffire pour des systèmes lents ou très stables.

2.7.5 Le terme Intégral (I)

Introduction

Le **terme intégral** prend en compte **l'historique de l'erreur**. Contrairement au terme proportionnel (P) qui réagit à l'instantané, le terme I observe le **cumul des erreurs dans le temps**.

Autrement dit, il « se souvient » si ton robot est resté longtemps en retard ou en avance par rapport à la consigne. Cela permet de **corriger les erreurs persistantes**, qu'on appelle aussi *erreurs statiques*.

Pour cela on utilise une **intégrale** de l'erreur sur le temps. C'est un concept mathématique qui signifie tout simplement **additionner** l'erreur à chaque appel à la valeur précédente.

Formule

$$\text{Commande}_I = k_I \cdot \int_0^t \text{erreur}(t)\, dt$$

— Le symbole $\int$ représente une **intégrale**, c'est-à-dire une **somme continue**.
— `dt` représente **le pas de temps** entre chaque mesure (typiquement 0.02s en FTC/FRC).
— En robotique, on **approxime** l'intégrale en additionnant l'erreur à chaque cycle :

$$\text{intégrale} \approx \sum \text{erreur} \times \Delta t$$

Exemple

Imagine que ton élévateur s'arrête à 1.15 m au lieu de 1.20 m. Le terme proportionnel devient petit (car l'erreur est petite), donc la correction s'arrête trop tôt. Le terme intégral va **accumuler** cette erreur de 5 cm à chaque cycle, et ajouter petit à petit une correction supplémentaire. Cela permet au bras de rattraper lentement ce dernier écart.

Code minimaliste

C++

```
double error = setpoint - current;
errorSum += error * dt; // dt = période du loop, typiquement 0.02s en FTC/FRC

double kI = 0.05;
double output_I = kI * errorSum;
```

Java

```
double error = setpoint - current;
errorSum += error * dt; // dt = période du loop, typiquement 0.02s en FTC/FRC

double kI = 0.05;
double output_I = kI * errorSum;
```

Avertissement : Le I peut devenir **trop fort** si l'erreur s'accumule trop longtemps (ex : robot bloqué). C'est ce qu'on appelle un *effet intégral excessif*. Pour cela il faut éviter que le `kI` ne soit trop grand. Aussi il est préférable de mettre un **limiteur** sur l'erreur cumulée ou d'utiliser un **anti-windup**.

2.7.6 Le terme Dérivé (D)

Introduction

Le **terme dérivé** mesure la **vitesse de variation de l'erreur**. Autrement dit, il regarde **à quelle vitesse l'erreur change**, pour anticiper ce qui va se passer.

On dit qu'il fait un **effet prédictif** : si l'erreur diminue rapidement, le D freine la commande pour **éviter un dépassement**. C'est un peu comme des amortisseurs sur une voiture : ils absorbent les variations brusques pour stabiliser le mouvement.

Formule

$$\text{Commande}_D = k_D \cdot \frac{d(\text{erreur})}{dt}$$

— Le symbole $\frac{d}{dt}$ est une dérivée, c'est-à-dire une mesure du **taux de changement**.
— En pratique, on approxime la dérivée par :

$$\frac{\Delta \text{erreur}}{\Delta t} = \frac{\text{erreur}_{\text{actuelle}} - \text{erreur}_{\text{précédente}}}{dt}$$

Exemple

Si ton élévateur va vite vers la cible, le D verra que l'erreur diminue très vite. Il va donc **freiner la commande** pour éviter qu'il ne dépasse la position voulue.

Inversement, si ton élévateur est presque immobile, le D ne fait rien (car le taux de variation est faible).

Code minimaliste

C++

```
double dError = (error - lastError) / dt;
lastError = error;

double kD = 0.4;
double output_D = kD * dError;
```

Java

```java
double dError = (error - lastError) / dt;
lastError = error;

double kD = 0.4;
double output_D = kD * dError;
```

Astuce : Le D est très utile pour éviter les oscillations (ex : bras qui rebondit autour de la cible). Mais il est **sensible au bruit** des capteurs. Si la position mesurée varie brutalement (bruit), la dérivée devient instable. On peut parfois lisser la mesure ou filtrer le D pour améliorer la stabilité.

2.7.7 Combiner les trois termes

Formule complète

$$\text{Commande} = k_P \cdot e + k_I \cdot \int e\,dt + k_D \cdot \frac{de}{dt}$$

Chaque terme a un rôle distinct :

Terme	Rôle principal
P	Corriger l'erreur actuelle
I	Corriger les erreurs passées
D	Anticiper les erreurs futures

Code complet

C++

```cpp
double error = setpoint - current;
errorSum += error * dt;
double dError = (error - lastError) / dt;
lastError = error;

double output = kP * error + kI * errorSum + kD * dError;

setMotorVoltage(output);
```

Java

```
double error = setpoint - current;
errorSum += error * dt;
double dError = (error - lastError) / dt;
lastError = error;

double output = kP * error + kI * errorSum + kD * dError;

setMotorVoltage(output);
```

Note : Les coefficients `kP`, `kI` et `kD` doivent être **ajustés (« tunés »)** pour chaque système afin d'obtenir un comportement optimal. Pour savoir comment régler ces paramètres, consulte la page *Le Contrôleur PID*.

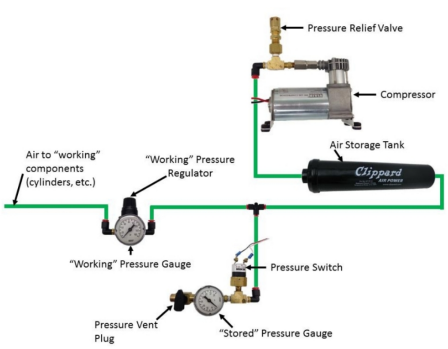
Note : Ce contrôleur est générique et réutilisable mais pas optimisé. Il est préférable d'utiliser un contrôleur PID déjà implémenté comme `PidRBL`.

2.8 Le Contrôleur PID

coucou

2.9 Utiliser la Pneumatique

Nous avons appris comment contrôler les moteurs et les capteurs du robot. Mais ces éléments ne sont pas les seuls présents sur le robot : il peut aussi y avoir de la pneumatique.



Le circuit pneumatique est composé de tous les éléments utilisant l'air compressé au sein du robot : un **compresseur**, de **réservoirs d'air** (air tanks), de jauges de pression, de **solénoïdes** et de **vérins**.

Les vérins sont les seuls actionneurs utilisant la pneumatique, ils sont utilisés pour des mouvements rapides ne nécessitant que 2 positions (rentré/sortir).

Les solénoïdes (ou solenoid valves) contrôlent le flux d'air. Ils permettent de diriger ou non l'air dans un vérin. Il existe 2 types de solénoïdes : les solénoïdes simple et doubles. Ils sont contrôlés par le PCM (Pneumatic Control Module) et se branchent donc sur les différents ports que celui-ci possède.

2.9.1 Solénoïdes simples

Les solénoïdes simples peuvent seulement appliquer ou bloquer la pression de leur unique sortie. Ils ne peuvent ainsi appliquer une pression qu'à une seule entrée d'un vérin. Ils sont utiles lorsqu'une force extérieur (gravité, ...) permet de rentrer/sortir le vérin ou bien lorsque le vérin n'est utilisé qu'une seule fois (Sorti mais jamais rentré).

Dans le code

Pour programmer un solénoïde simple, il faut créer une instance de la classe Solenoid, son constructeur attend comme argument le port sur lequel le solenoid est branché :

```
#include <frc/Solenoid.h>
frc::Solenoid mon_solenoide(0);
```

La méthode `void Set(bool on)` permet d'ouvrir ou de fermer le solénoïde :

```
// Je sors le vérin
mon_solenoide.Set(true);
```

Note : Quand on déclare une instance d'un solénoïde (simple ou double), le compresseur est automatiquement mis en route.

2.9.2 Solénoïdes doubles

Les solénoïdes possèdent une entrée et deux sorties. La pression peut donc être appliquée à une sortie, à l'autre ou bien à aucune des deux. Quand un solénoïde double est branché à un vérin, il peut ainsi le sortir, le rentrer ou bien le laisser « libre ».

Dans le code

La classe DoubleSolenoid est similaire à la classe Solenoid. Son constructeur attend comme argument les 2 ports (Forward et Reverse) sur lequel le solenoid est branché :

```
#include <frc/DoubleSolenoid.h>
frc::DoubleSolenoid mon_solenoide(0, 1);
```

La méthode `void Set(Value value)` permet de contrôler l'état du solénoïde. Il attend en argument une des valeur de l'enum `Value` : `Off`, `Forward` or `Reverse` :

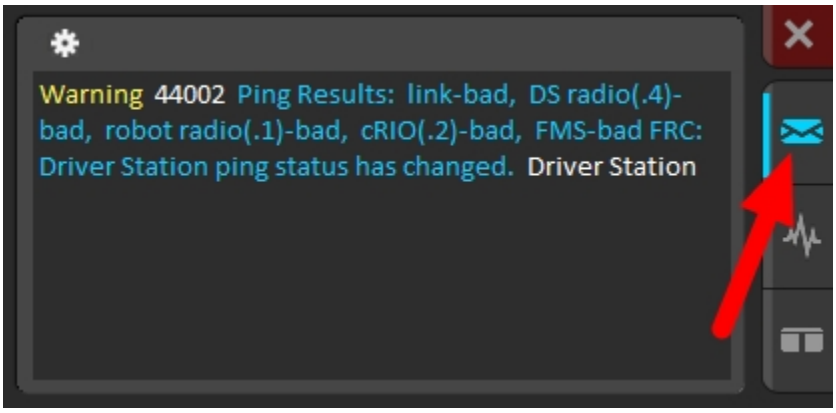
```
// Je sors le vérin
mon_solenoide.Set(frc::DoubleSolenoid::Value::kForward);

// Je rentre le vérin
mon_solenoide.Set(frc::DoubleSolenoid::Value::kReverse);

// Je laisse le vérin libre
mon_solenoide.Set(frc::DoubleSolenoid::Value::kOff);
```

2.10 Communiquer avec le robot

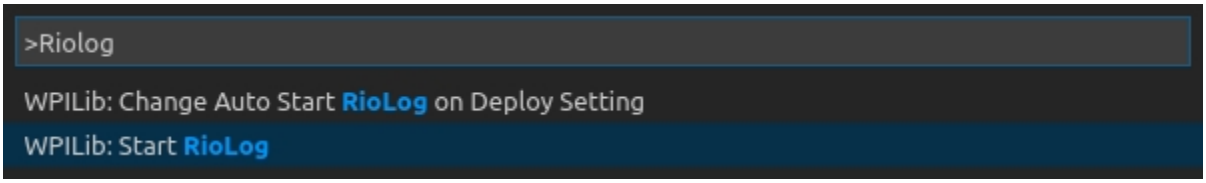
Une des notions que nous n'avons pas encore abordée et la communication entre le robot et l'ordinateur du pilote. Afficher des messages ou des valeurs numériques peut en effet être très utile tant pour faire des tests qu'en compétition.

2.10.1 Cout et Printf()

La première méthode à notre disposition est d'utiliser l'affichage console classique du C++ (`cout` et `printf()`). Le flux de sortie du robot est en effet redirigé sur le réseau. On peut le lire directement sur la Driver Station.



On peut aussi lire le flux de sortie du robot avec le RioLog. On peut lancer le RioLog dans VS Code en entrant `riolog` dans la palette de commandes (Ctrl + Shift + P) puis en sélectionnant `WpiLib: Start RioLog`.



2.10.2 SmartDashboard

Afficher des informations avec `cout` pose un problème : si on affiche régulièrement plusieurs messages, il peut devenir très compliqué de suivre le défilement de ceux-ci. Pour cela, une alternative existe : c'est le SmartDashboard. Pour l'ouvrir dans VS Code : Ctrl + Shift + P puis `Start Tool` et sélectionner `SmartDashboard`.

Pour utiliser le SmartDashboard, il n'y a pas besoin de déclarer une instance de la classe SmartDashboard. On peut appeler les méthodes de celle-ci avec l'opérateur de résolution de portée `::`. Pour afficher des données sur le SmartDashboard, il faut fournir 2 choses : une **key** qui identifie la donnée et sa **valeur**. Il existe plusieurs fonctions selon le type de donnée à afficher (texte, nombre ou booléen) :

```
#include <frc/smartdashboard/SmartDashboard.h>

frc::SmartDashboard::PutString("2020 Game","WaterGame");
frc::SmartDashboard::PutNumber("Best Team", 5553);
frc::SmartDashboard::PutBoolean("148+254=5553", false);
```

La fonction `PutNumber` attend comme argument un `double`. Ici, le int `5553` sera converti en un double.

Du côté du pilote, on peut lire les valeurs sur le SmartDashboard et on peut aussi les modifier. On peut alors récupérer ces valeurs avec les fonctions GetString, GetNumber et GetBoolean. Il faut alors donner en argument la **key** de la donnée et une valeur qui sera renvoyée si la donnée n'existe pas. Le pilote peut aussi modifier le widget qui affiche une donnée et ses propriétés.

2.10.3 NetworkTables

Pour assurer la communication entre le robot et l'ordinateur, le SmartDashboard utilise en fait un protocole de communication créé par WpiLib : les NetworkTables. Pour le voir, il suffit d'ouvrir l'OutlineViewer : Ctrl + Shift + P puis `Start Tool` puis `OutlineViewer`.

L'OutlineViewer permet de lire toutes les données transférées via les NetworkTables. On voit ainsi apparaître un « dossier » (une Table) nommé `/SmartDashboard` dans lequel on retrouve toutes les données (key + valeur) créées. Chaque Table est identifiable par une chaîne de caractères. Elle peut posséder plusieurs données (key + valeur) qui peuvent être, comme pour le SmartDashboard, des `string`, des `double` ou des `bool`.

Pour accéder à ces données, on peut passer par la classe NetworkTable. Premièrement, il faut créer et récupérer une l'instance de la classe NetworkTable grâce au nom de la Table :

```
#include <networktables/NetworkTable.h>
auto table = NetworkTable::GetTable("datatable");
```

Note : Le mot clé `auto` remplace la déclaration du type d'une variable ou d'un objet. Le type est automatiquement déduit. Par exemple, ici, la variable `x` sera automatiquement un `int` :

```
int i = 5553;
auto x = i;
```

Ensuite, on peut utiliser la NetworkTable (son pointeur en fait) comme avec le SmartDashboard :

```
table->PutNumber("PositionPivot", encodeur.GetDistance());
table->PutBoolean("Pince ouverte", isPinceOuverte);
table->PutString("Alliance", "Rouge");
```

Les Tables sont automatiquement synchronisées à intervalles réguliers. Mais pour plus de performances, il existe une fonction qui synchronise immédiatement tous les changements effectués sur les NetworkTables quand on l'appelle :

```
nt::NetworkTable::Flush();
```

2.10.4 ShuffleBoard

Le Shuffleboard est une version améliorée du SmartDashboard et il fonctionne de la même manière que ce dernier. Il possède une interface et des widgets plus plaisant et peut avoir plusieurs fenêtres (ou tabs).

Sa principale utilité vis-à-vis du SmartDashboard est que l'on peut configurer dans le code du robot la disposition des widgets et par example changer de fenêtre avec le Joystick. Pour découvrir toutes ses fonctionnalités : voici la documentation.

2.11 Streamer une video

Partager un flux vidéo peut s'avérer utile en match. En effet, voir sur son PC la vue d'une caméra montée sur le robot peut aider le pilote à se positionner correctement sur le terrain.

2.11.1 Quelle camera ?

Le RoboRio possède deux ports USB. C'est pourquoi il est possible d'utiliser des caméras USB avec celui-ci. Ces cameras sont pratiques car disponibles partout dans le commerce, bon marché, faciles à remplacer et à brancher.

Une des caméras USB les plus utilisées en FRC est la **Microsoft LifeCam HD-3000** :

Il existe aussi des caméras IP qui permettent directement de streamer un flux vidéo sur le réseau. Cependant, celles-ci sont de moins en moins utilisées.

2.11.2 Dans le code

Bien évidement, WPILib a créé une classe afin de streamer le flux vidéo d'une caméra USB. Il s'agit de la classe CameraServer.

Le constructeur de cette classe est privé, on ne peut donc pas créer d'instances de cette classe. Au lieu de cela, il est possible de récupérer l'unique instance de la classe grâce à la méthode `static CameraServer* GetInstance ()`.

Note : Les variables et les fonctions membres `static` appartiennent à la classe mais pas aux objets instanciés à partir de la classe. Une fonction membre déclarée `static` a ainsi la particularité de pouvoir être appelée sans devoir instancier la classe.

Après avoir récupérer un pointeur sur l'instance de la classe, on peut utiliser ses méthodes :

La fonction cs::UsbCamera StartAutomaticCapture() crée un flux vidéo à partir de la caméra USB n°0. Elle renvoie aussi une instance de la classe UsbCamera créée :

```
#include <CameraServer.h>
CameraServer::GetInstance()->StartAutomaticCapture();
```

Il existe aussi d'autres méthodes qui permettent de récupérer les images du flux pour, par exemple, les analyser.

2.11.3 Les alternatives

Utiliser le RoboRio pour streamer un flux vidéo n'est pas toujours la meilleure solution. En effet, le processeur du RoboRio n'est pas adapté à la compression vidéo et le flux vidéo utilise ainsi une grande partie de la bande passante.

Pour pouvoir utiliser moins de bande passante, il peut alors être intéressant d'avoir recours à un coprocesseur comme un Raspberry Pi. Celui-ci pourra alors utiliser un codec vidéo plus avantageux comme le H.264.

2.12 TEEEEESSST

hello :-)

2.13 Organiser son Programme

2.13.1 Diviser le programme en Classes

Pour l'instant, tout le code que nous écrivions était uniquement dans la classe principale `Robot`. Cependant, pour des projets plus importants, la quantité de code commence à être trop grande pour se située dans un unique fichier.

Pour organiser le programme du Robot, il est donc nécessaire de le structurer en plusieurs fichiers. Chacun gérant une fonctionnalité différente du code.

Une première méthode pour structurer le code peut être de créer une classe pour chacun des mécanismes du robot (ou subsystems) :

```
subsystems/
├── BaseRoulante/
├── Grimpeur/
```

(suite sur la page suivante)

(suite de la page précédente)

```
├── Pince/
└── Pivot/
```

Chacune de ses classes contient ainsi les méthodes nécessaires au fonctionnement du subsytem. Par exemple, voici à quoi pourrait ressembler le fichier BaseRoulante.h :

```
class BaseRoulante
{
public:
  BaseRoulante();

  void Drive(double x, double y);
  void Stop();

  void ActiverVitesse1();
  void ActiverVitesse2();
  void ChangerVitesse();

  double GetDistanceDroite();
  double GetDistanceGauche();
  double GetAngle();
  void ResetCapteurs();

private:
  // Variables, objets et méthodes
  // Privés pour gérer en interne le subsystem

};
```

2.13.2 Cablage.h ou Constants.h

En séparant les subsystems en plusieurs classes/fichiers, on sépare aussi les objets qu'ils contiennent (contrôleurs moteur, capteurs, ...). Il peut ainsi, par exemple, être compliqué de savoir si le port X du RoboRio est déjà utilisé. Sur quels ports sont branchés les encodeurs de la base ?

Pour simplifier cela, on crée un fichier nommé `Cablage.h` ou `Constants.h` qui contient des constantes utilisées dans les autres fichiers :

```
// PWM MOTORS
constexpr int PWM_ROUES_PINCE = 0;
constexpr int PWM_BASE_DROITE_1 = 1;
constexpr int PWM_BASE_DROITE_2 = 2;
constexpr int PWM_BASE_GAUCHE_1 = 3;
constexpr int PWM_BASE_GAUCHE_2 = 4;

// CAN MOTORS
constexpr int CAN_PIVOT = 1;

// DIO ENCODEURS
constexpr int DIO_ENCODEUR_PIVOT_A = 0;
constexpr int DIO_ENCODEUR_PIVOT_B = 1;
```

(suite sur la page suivante)

(suite de la page précédente)

```
// PCM PNEUMATICS
constexpr int PCM_PINCE_A = 0;
constexpr int PCM_PINCE_B = 1;
```

Grâce à la présence de ca fichier, il est maintenant facile de savoir où chacun des contrôleur moteur doit être branché, quels sont les port PWM libres, ect …

2.13.3 Le Programme Principal

Maintenant que les classes permettant de contrôler les subsystems existent, il faut les intégrer dans notre classe principale `Robot`. Pour cela, on a juste à créer une instance de chacune des classes dans `Robot`. Pour la partie Téléopérée, le but du programme principal est d'utiliser des `if` qui, en fonction des entrée du joystick, appellent certaines fonctions.

```
#include <frc/TimedRobot.h>
#include <frc/Joystick.h>
#include "BaseRoulante.h"
#include "Pince.h"

class Robot : public frc::TimedRobot
{
public:
  void TeleopPeriodic() override
  {
    if(m_Joystick.GetRawButton(1))
    {
      m_Pince.Attraper();
    }
    else if(m_Joystick.GetRawButton(2))
    {
      m_Pince.Ejecter();
    }
    else
    {
      m_Pince.Stop();
    }

    m_BaseRoulante.Drive(m_Joystick.GetX(), m_Joystick.GetY());
  }

private:
  frc::Joystick m_Joystick(0);
  BaseRoulante m_BaseRoulante;
  Pince m_Pince;
};
```

Attention : Encore une fois, les méthodes appelées par le programme principal ne doivent pas durer dans le temps au risque de rester bloqué dans une des fonctions. Les boucles `while`, `do while` et `for` sont donc généralement à éviter partout dans le code.

2.14 Lien utiles et sources

2.14.1 Tutos, Cours

— Frc-docs : documentation officielle
— Frc Programming Done Right
— Frc Electrical Bible

2.14.2 Autres librairies

— Cross The Road Electronics : pour contrôler les Victor SPX, Talon SRX, … via le bus CAN
— Rev : pour contrôler le Spark MAX, seul contrôleur moteur disponible pour le NEO
— OpenCV : pour faire de la reconnaissance visuelle

2.14.3 Documentations

— WpiLib
— Cross The Road Electronics
— Rev
— OpenCV

2.14.4 PID et Motion profiling

— Intro to Control Theory
— PID without a Phd
— Conférence de 254 sur le sujet et leurs slides

2.14.5 Visual Processing

— Conférence de 254 sur le sujet
— LyonVision-pi-gen : image de Raspberry custom pour faire la reconnaissance visuelle
— LyonVision-Template : template (= exemple) de projet sur lequel se baser pour faire la reconnaissance visuelle
— LyonVision-Calibration : programme pour calibrer une caméra sur RaspberryPi (basé sur LyonVision-Template)

2.14.6 Git et Github

— Cours OpenClassroom
— Cheat Sheet
— Github de Robo'Lyon

2.14.7 Forums et Communauté

— Chief Delphi : The place to be
— Reddit
— Frc Discord
— Frc Discord Quebec
— Groupe Facebook First Quebec
— The Compass Alliance