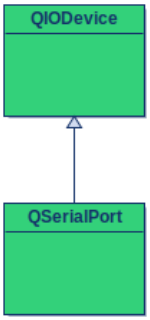
5. La communication série avec `QSerialPort`



`QSerialPort` est une classe de la librairie Qt couramment utilisée pour la transmission de données en série entre un ordinateur et des périphériques tels que des capteurs, des modems ou des microcontrôleurs.

Cette classe est dérivée de **`QIODevice`**, ce qui signifie qu'elle hérite de l'interface d'entrée/sortie (I/O) standard de Qt et peut être utilisée de manière similaire à d'autres classes pour la lecture et l'écriture de données, comme **`QFile`** pour les fichiers ou **`QTcpSocket`** pour la communication réseau.

Pour la gestion des ports série avec la librairie Qt, il est nécessaire d'ajouter dans le fichier .pro : QT += serialport

5.1. Gestion des ports physiques

`QSerialPort` permet de se connecter à un port série sur le système :

- **Sous linux :** */dev/ttyS0…*, /dev/AMA0… ou encore pour un périphérique USB exposé comme un port série */dev/USB0…*
- **Sous Windows :** com1, com2…

Pour découvrir la liste des ports série disponible sur le système, il est possible d'utiliser la classe `QSerialPortInfo` avec en particulier sa méthode statique `availablePorts()` qui retourne les caractéristiques des ports présents :

Lecture des informations concernant les ports série du système :

```
#include <QSerialPortInfo>

foreach (const QSerialPortInfo &info, QSerialPortInfo::availablePorts()) {
        qDebug() << "Nom : " << info.portName();
        qDebug() << "Fabricant : " << info.manufacturer();
}
```

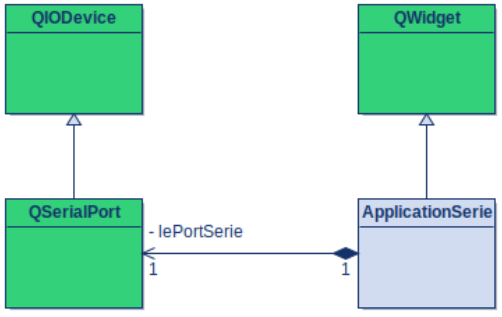
5.2. Déclaration et configuration du port

Il est dans un premier temps nécessaire d'instancier `QSerialPort` soit de manière automatique, soit de manière dynamique. Le constructeur se rencontre soit avec deux paramètres, le nom du port obtenu par exemple avec `QSerialPortInfo` sous la forme d'une `QString` et le pointeur sur la classe parent, soit uniquement avec le pointeur vers la classe parent. Si c'est le cas, il est nécessaire de préciser le nom du port par la suite avec la méthode `setPortName()`.

Instanciation du port

```
#include <QSerialPort>

QSerialPort lePortSerie("ttyS0");
//ou
QSerialPort *lePortSerie = new QSerialPort("ttyS0");
// ou
QSerialPort lePortSerie;
lePortSerie.setPortName("ttyS0");
```



La classe dispose d'un certain nombre de propriétés qui permettent de configurer le port, les principales sont :

baudRate : qint32	Pour fixer la vitesse
dataBits : DataBits	Pour déterminer le nombre de bits de données

parity : Parity	Pour fixer le type de parité
stopBits : StopBits	Pour fixer le nombre de bits de stop
flowControl : FlowControl	Pour fixer le contrôle de flux

À ces différents attributs correspond une valeur obtenue dans une des énumérations présentées par la suite.

5.2.1. Vitesse de transmission

Cette énumération décrit le débit en **bauds** avec lequel le périphérique de communication fonctionne. Les valeurs listées correspondent aux vitesses standard couramment utilisées pour la liaison série. Le baud est une mesure de la vitesse de transmission, exprimée en changements de signal par seconde. Généralement, dans la communication série simple, 1 baud = 1 bit/s. Dans d'autres types de communication, comme celles utilisant des modems, plusieurs bits peuvent être codés par changement d'état, et dans ce cas, les bauds et les bits par seconde peuvent différer.

enum QSerialPort::BaudRate

Constante	Valeur	Description
`QSerialPort::Baud1200`	1200	1200 bauds.
`QSerialPort::Baud2400`	2400	2400 bauds.
`QSerialPort::Baud4800`	4800	4800 bauds.
`QSerialPort::Baud9600`	9600	9600 bauds.
`QSerialPort::Baud19200`	19200	19200 bauds.
`QSerialPort::Baud38400`	38400	38400 bauds.
`QSerialPort::Baud57600`	57600	57600 bauds.
`QSerialPort::Baud115200`	115200	115200 bauds.

5.2.2. Nombre de bits de données

Cette énumération décrit le nombre de bits de données utilisées. Seules les dernières valeurs sont plus utilisées. **enum QSerialPort::DataBits**

Constante	Valeur	Description
`QSerialPort::Data5`	5	Le nombre de bits de données par caractère est de 5. Utilisé pour le code Baudot, généralement avec de vieux équipements comme les téléscripteurs.
`QSerialPort::Data6`	6	Le nombre de bits de données par caractère est de 6. Peu utilisé.
`QSerialPort::Data7`	7	Le nombre de bits de données par caractère est de 7. Utilisé pour le véritable code ASCII, souvent avec de vieux équipements comme les téléscripteurs.
`QSerialPort::Data8`	8	Le nombre de bits de données par caractère est de 8. Utilisé pour la plupart des types de données, car il correspond à la taille d'un octet. Universellement utilisé dans les applications modernes.

5.2.3. Contrôle de parité

La **parité** est un mécanisme utilisé dans les communications série pour détecter les erreurs de transmission de données. Elle consiste à ajouter un bit supplémentaire aux données envoyées, appelé **bit de parité**, qui sert à vérifier si le nombre total de bits 1 dans les données est pair ou impair. Il existe plusieurs types de contrôle de parité, et leur choix dépend du protocole de communication utilisé ainsi que des exigences de fiabilité.

Cette énumération décrit le schéma utilisé pour le contrôle de parité. **enum QSerialPort::Parity**

Constante	Valeur	Description
`QSerialPort::NoParity`	0	Pas de bit de parité, **aucun contrôle d'erreur**
`QSerialPort::EvenParity`	2	Le nombre de bits 1 doit être **pair**, détection d'erreur simple
`QSerialPort::OddParity`	3	Le nombre de bits 1 doit être **impair**, détection d'erreur simple
`QSerialPort::SpaceParity`	4	Pas de contrôle d'erreur, le bit de parité est **toujours 0**
`QSerialPort::MarkParity`	5	Pas de contrôle d'erreur, le bit de parité est **toujours 1**

Seul pour le mode de parité « pair » ou « impair » détecte un nombre impair ou pair d'erreurs coté réception, mais ne peut pas les corriger. La parité toujours à 1 ou toujours à 0 ajoute un bit supplémentaire, mais ne permet pas de détection d'erreur et est de ce fait que très peu utilisé dans les communications actuelles. Le premier cas n'ajoute pas de bit de parité, la transmission se fait simplement sans contrôle d'erreur possible.

5.2.4. Nombre de bits de stop

Dans une communication série, le **bit de start** et le ou les **bits de stop** sont des éléments essentiels du protocole de transmission de données. Ils encadrent les bits de données pour assurer la synchronisation et la validité des informations transmises. Le **bit de start** signale le début de la transmission, tandis que les **bits de stop** indiquent la fin du caractère. Le choix du nombre de bits de stop permet d'adapter la vitesse de transmission aux besoins du récepteur, avec 1 bit pour des systèmes rapides et 2 bits pour plus de fiabilité dans des systèmes plus lents ou anciens.

L'énumération suivante décrit le nombre de bits de stop utilisés. Il y a toujours qu'un seul bit de start.

enum QSerialPort::StopBits

Constante	Valeur	Description
`QSerialPort::OneStop`	1	**1 bit de stop**. Option la plus courante. Elle permet une transmission plus rapide, car moins de temps est consacré à la signalisation de la fin du caractère.
`QSerialPort::OneAndHalfStop`	3	**1.5 bits de stop**. Rarement utilisé et uniquement sur les plateformes Windows, mais cela peut être nécessaire dans certains systèmes où le récepteur a besoin de plus de temps pour traiter les caractères avant de recevoir le suivant.
`QSerialPort::TwoStop`	2	**2 bits de stop**. Utilisé lorsque le système de réception est plus lent ou que la synchronisation est critique. Cela donne plus de temps au récepteur pour traiter chaque caractère, au prix d'une réduction de la vitesse de transmission effective.

5.2.5. Contrôle de flux

Le **contrôle de flux** est un mécanisme essentiel en communication série pour gérer le débit des données entre un émetteur et un récepteur. Il assure que l'émetteur n'envoie pas plus de données que ce que le récepteur peut traiter à un moment donné, évitant ainsi les **surcharges** et la **perte de données**.

Il existe deux principaux types de contrôle de flux utilisés dans les communications série :

- Contrôle de flux matériel (basé sur des signaux physiques)
- Contrôle de flux logiciel (basé sur des caractères de contrôle)

Contrôle de flux matériel (Hardware Flow Control) RTS/CTS :

Le contrôle de flux matériel utilise des **lignes supplémentaires** dans la connexion série pour signaler quand le récepteur est prêt ou non à recevoir des données. Ces lignes de contrôle physiques permettent à l'émetteur et au récepteur de se coordonner sans dépendre des données elles-mêmes.

- **RTS (Request to Send)** : C'est un signal émis par l'émetteur pour indiquer qu'il est prêt à envoyer des données.
- **CTS (Clear to Send)** : C'est la réponse du récepteur qui signale qu'il est prêt à recevoir les données.

Fonctionnement :	**Avantages** :
L'émetteur vérifie si le signal **CTS** est actif (haut). Si c'est le cas, il commence à envoyer des données. Le récepteur, de son côté, active ou désactive le signal **CTS** selon sa capacité à recevoir plus de données. Si son tampon est plein, il désactive le signal, forçant l'émetteur à attendre.	Hautement fiable, car il repose sur des signaux physiques. Utilisé dans les systèmes où le débit de données est élevé et où la perte de données doit être évitée.

Contrôle de flux logiciel (Software Flow Control) XON/XOFF :

Le contrôle de flux logiciel utilise des caractères spéciaux insérés dans le flux de données pour gérer la communication. Ces caractères sont interprétés par l'émetteur et le récepteur pour démarrer ou arrêter la transmission de données.

- **XON (Transmission On, ASCII 0x11) :** Ce caractère est envoyé par le récepteur pour indiquer à l'émetteur qu'il peut reprendre l'envoi des données.
- **XOFF (Transmission Off, ASCII 0x13) :** Ce caractère est envoyé pour indiquer à l'émetteur qu'il doit suspendre l'envoi de données, car le récepteur n'est pas en mesure de traiter davantage de données pour le moment.

Fonctionnement
Le récepteur surveille son tampon de réception. Si son tampon devient plein (ou presque), il envoie un **XOFF** à l'émetteur pour suspendre temporairement l'envoi des données. Lorsque le récepteur a traité suffisamment de données et que son tampon est à nouveau disponible, il envoie un **XON** pour indiquer à l'émetteur qu'il peut reprendre la transmission.

Avantages	Inconvénients
Ne nécessite pas de câblage supplémentaire ou de signaux matériels. Peut être utilisé dans les systèmes où les lignes de contrôle physiques ne sont pas disponibles.	Ajoute des caractères de contrôle dans le flux de données, ce qui peut interférer avec certaines transmissions si des données ressemblant à **XON/XOFF** sont envoyées accidentellement. Impose une transmission de caractères ASCII uniquement. Moins rapide et moins fiable que le contrôle de flux matériel, car il repose sur l'interprétation des caractères.

Ce type de contrôle est généralement moins performant en termes de débit que le contrôle matériel, bien qu'il soit souvent utilisé dans des configurations où le câblage RTS/CTS n'est pas possible.

Pas de contrôle de flux (No Flow Control)

Dans certaines situations, le contrôle de flux n'est pas nécessaire. Cela peut se produire lorsque :

- Le débit de données est suffisamment faible pour que le récepteur puisse tout traiter sans risque de surcharge.
- Un autre mécanisme de gestion des données, comme des tampons de grande taille ou des temporisations, est utilisé pour réguler le flux de données.

Dans ce cas, aucun mécanisme de contrôle de flux n'est implémenté, et les données sont envoyées continuellement.

L'énumération suivante décrit le nombre de bits de stop utilisés. Il y a toujours qu'un seul bit de start.

enum QSerialPort::FlowControl

Constante	Valeur	Description
`QSerialPort::NoFlowControl`	0	Pas de contrôle de flux
`QSerialPort::HardwareControl`	1	Contrôle de flux matériel RTS/CTS
`QSerialPort::SoftwareControl`	2	Contrôle de flux logiciel XOn/XOFF

Remarque

La configuration du port série doit être effectuée avant que celui-ci soit ouvert.

5.3. Ouverture du port

Ce paragraphe introduit la notion de mode de transmission que l'on peut rencontrer avec une liaison série. Le **mode full duplex** et le **mode half duplex** sont les deux types de modes de communication utilisés. Ces modes déterminent comment les données peuvent être envoyées et reçues entre deux dispositifs, en fonction de leur capacité à transmettre dans une ou deux directions simultanément. :

5.3.1. Mode Half Duplex

Le mode **half duplex** (ou **semi-duplex**) permet aux deux parties (émetteur et récepteur) de **communiquer dans les deux directions**, mais **pas simultanément**. Cela signifie qu'un dispositif peut soit émettre, soit recevoir des données à un instant donné, mais pas les deux en même temps. Pour certains périphériques, ne faisant qu'émettre des données, cela peut-être suffisant.

5.3.2. Mode Full Duplex

Le mode **full duplex** permet aux deux parties de **communiquer simultanément** dans les deux directions. Autrement dit, l'émetteur et le récepteur peuvent envoyer et recevoir des données en même temps, sur des canaux distincts ou en utilisant des méthodes qui permettent la transmission simultanée.

5.3.3. Possibilité d'ouverture du port sous Qt

L'ouverture du port comme pour les fichiers ou les autres types de communication est basée sur les méthodes de la classe `QIODevice`. La méthode `open()` est surchargée dans la classe `QSerialPort`. Ce qui permet de choisir le mode d'ouverture du port série. Elle retourne VRAI si l'ouverture s'est effectuée avec succès et FAUX sinon. Un code d'erreur peut alors être obtenu avec la méthode `error()`. La méthode renvoie également FAUX, si l'ouverture a réussi, mais que les paramètres de configuration ne sont pas corrects. Dans ce cas, le port est automatiquement refermé pour ne pas laisser un port avec des paramètres incorrects.

Les valeurs possibles pour le port série sont :

Constantes	valeur	Description
`QIODeviceBase::ReadOnly`	`0x0001`	Le port est ouvert en lecture. Il ne pourra que recevoir des données.
`QIODeviceBase::WriteOnly`	`0x0002`	Le port est ouvert en écriture. Il ne pourra que transmettre des données.
`QIODeviceBase::ReadWrite`	`ReadOnly | WriteOnly`	Le port est ouvert en écriture et en lecture. Il peut donc émettre et recevoir simultanément des données.

Exemple de programme pour la configuration et l'ouverture d'un port série.

Configuration et ouverture du port

```
#include <QSerialPort>
QSerialPort serialPort;      // Création de l'objet QSerialPort
serialPort.setPortName("/dev/ttyS0"); // ou /dev/ttyACM0 selon votre périphérique

// Configurer les paramètres de communication série
serialPort.setBaudRate(QSerialPort::Baud9600);
serialPort.setDataBits(QSerialPort::Data8);
serialPort.setParity(QSerialPort::NoParity);
serialPort.setStopBits(QSerialPort::OneStop);
serialPort.setFlowControl(QSerialPort::NoFlowControl);

// Ouverture du port en mode lecture/écriture
if (serialPort.open(QIODevice::ReadWrite)) {
    qDebug() << "Port série ouvert avec succès.";
} else {
    qDebug() << "Erreur lors de l'ouverture du port : " << serialPort.errorString();
}
```

5.4. Ecriture synchrone

Pour écrire des données sur le port série, plusieurs surcharges de la méthode write() sont disponibles :

Méthodes pour l'écriture sur le port série

```
qint64 QIODevice::write(const char *data)
qint64 QIODevice::write(const char *data, qint64 maxSize)
qint64 QIODevice::write(const QByteArray &data)
```

Chacune de ces méthodes retourne le nombre d'octets qui a réellement été écrit sur le port ou -1 si une erreur est survenue.

La première écrit la chaîne de caractères passée en paramètre, elle se termine obligatoirement par '\0'.

La seconde réalise la même opération, mais uniquement pour les **maxSize** premiers caractères

La dernière permet d'envoyer chaque octet un par un du tableau passé en paramètre.

5.5. Lecture synchrone

Plusieurs méthodes sont également disponibles pour la lecture des données en provenance du port série :

Méthodes pour la lecture sur le port série

```
qint64 QIODevice::read(char *data, qint64 maxSize)
QByteArray QIODevice::read(qint64 maxSize)
QByteArray QIODevice::readAll()
```

La première lit au maximum **maxSize** octets sur le port et renvoie le nombre d'octets lus. En cas d'erreur la méthode retourne -1 et 0 si il n'y a plus d'octet disponible pour la lecture. Le résultat est stocké à l'adresse fournie par le paramètre **data**.

La seconde lit au plus maxSize octets sur le port et retourne un QByteArray avec les valeurs lues.

La dernière lit toutes les données disponibles sur le port et les stocke en retour dans un QbyteArray.

5.6. Lecture et écriture asynchrones

Par défaut, la lecture et l'écriture sont des opérations bloquantes. Par exemple si l'on demande la lecture d'un certain nombre d'octets, le système attend que les octets arrivent sur le port série, le programme reste bloqué. Pour éviter ce phénomène, le signal **void QIODevice::readyRead()** indique que des données sont disponibles à la lecture sur le port. Grâce au mécanisme de signaux et de slot de Qt, un slot connecté au signal **readyRead** sera en mesure de lire les données reçues le moment venu.

Lecture asynchrone

```
connect(&serialPort, &QSerialPort::readyRead, this, &MyClass::onQSerialPort_readyRead);

void MyClass::onQSerialPort_readyRead() {
    QByteArray data = serialPort.readAll();
    // Traitement des données reçues
}
```

La méthode **bytesAvaillable()** retourne le nombre d'octets disponibles en lecture dans le tampon du port série. Si ce nombre est supérieur à zéro, cela signifie qu'il y a des données à lire.

Si le flux de données est important, **bytesAvailable()** peut être utilisé pour lire les données par morceaux afin d'éviter de surcharger le buffer ou de lire des données incomplètes.

De même, le signal **void QIODevice::bytesWritten(qint64 bytes)** est émis chaque fois que l'écriture demandée a été réalisée, le nombre d'octets écrit est fourni en paramètre.

Lecture asynchrone

```
#include <QCoreApplication>
#include <QSerialPort>
#include <QDebug>

class SerialWriter : public QObject {
    Q_OBJECT

public:
    SerialWriter() {
        serialPort.setPortName("/dev/ttyS0");
        serialPort.setBaudRate(QSerialPort::Baud9600);
        serialPort.setDataBits(QSerialPort::Data8);
        serialPort.setParity(QSerialPort::NoParity);
        serialPort.setStopBits(QSerialPort::OneStop);
        serialPort.setFlowControl(QSerialPort::NoFlowControl);
       connect(&serialPort, &QSerialPort::bytesWritten, this,
&SerialWriter::onBytesWritten);
       if (serialPort.open(QIODevice::WriteOnly)) {
            qDebug() << "Port série ouvert en écriture.";
            QByteArray data = "Hello, Serial Port!";
            serialPort.write(data);
        } else {
            qDebug() << "Erreur à l'ouverture du port :" <<
serialPort.errorString();
        }
    }

private slots:
    void onBytesWritten(qint64 bytes) {
        qDebug() << bytes << " octets écrits sur le port série.";
      // Si d'autres données doivent être envoyées, cette méthode peut les traiter ici.
    }

private:
    QSerialPort serialPort;
};

int main(int argc, char *argv[]) {
    QCoreApplication app(argc, argv);

    SerialWriter writer;

    return app.exec();
}
```

5.7. Gestion des erreurs

QSerialPort a plusieurs mécanismes pour gérer les erreurs via **error()** et les événements spécifiques de la liaison série :

- **Signal errorOccurred(QSerialPort::SerialPortError error)** : émis lorsqu'une erreur se produit.. Le paramètre renvoie une des valeurs présentes dans l'énumération ci-dessous.

Cette énumération décrit les différentes erreurs que peut générer QSerialPort

Constante	Valeur	Description
QSerialPort::NoError	0	Pas d'erreur
QSerialPort::DeviceNotFoundError	1	Une erreur s'est produite lors de la tentative d'ouverture d'un périphérique inexistant.
QSerialPort::PermissionError	2	Une erreur s'est produite lors de la tentative d'ouverture d'un appareil déjà ouvert par un autre processus ou par un utilisateur ne disposant pas de suffisamment d'autorisations et d'informations d'identification pour l'ouvrir.
QSerialPort::OpenError	3	Une erreur s'est produite lors de la tentative d'ouverture d'un périphérique déjà ouvert dans cet objet.
QSerialPort::NotOpenError	10	Cette erreur se produit lors de l'exécution d'une opération qui ne peut être effectuée avec succès que si le périphérique est ouvert.
QSerialPort::WriteError	4	Une erreur d'E/S s'est produite lors de l'écriture des données.
QSerialPort::ReadError	5	Une erreur d'E/S s'est produite lors de la lecture des données.
QSerialPort::ResourceError	6	Une erreur d'E/S s'est produite lorsqu'une ressource devient indisponible, par ex. lorsque le périphérique est supprimé de manière inattendue du système.
QSerialPort::UnsupportedOperationError	7	Le fonctionnement de l'appareil demandé n'est pas pris en charge ou interdit par le système d'exploitation en cours d'exécution.
QSerialPort::TimeoutError	9	Une erreur de délai d'attente s'est produite.
QSerialPort::UnknownError	8	Une erreur non identifiée s'est produite.

La Méthode **QString QIODevice::errorString()** const retourne la chaîne de caractère en correspondance avec le numéro de la dernière erreur qui c'est produit.