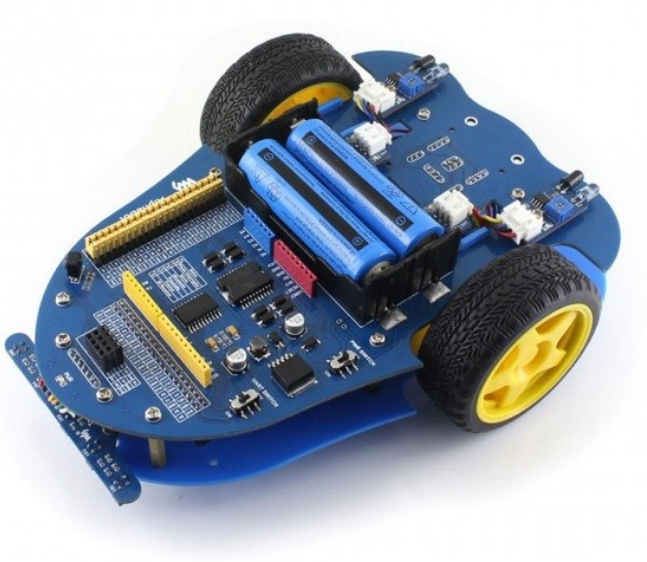
Cette étude porte sur la mise en pratique de développement sur la gestion des entrées-sorties associées au **GPIO** sur le Robot **AlphaBot** à l'aide du Raspberry. Nous utiliserons la bibliothèque **wiringPi**.



AlphaBot est un simple châssis en kit rudimentaire, sans soudure, comportant le nécessaire pour la réalisation d'un projet robotique sur base d'une carte compatible Arduino Uno ou d'une carte Raspberry Pi.

Utilisation de la bibliothèque WiringPI

WiringPi est une des bibliothèques importante pour le Raspberry Pi qui possède des cartes d'extension comme c'est le cas avec notre robot. Elle est écrite en C. Elle contient des routines permettant un accès facile et surtout direct aux périphériques intégrés dans la Raspberry.

Cette bibliothèque WiringPi nous permet d'agir directement sur les GPIO (avec un temps de réponse beaucoup plus réactif) et surtout de pouvoir activer les sorties en mode PWM afin de proposer des tensions moyennes de sorties réglables suivant le rapport cyclique.

Installation de WiringPi sur la raspberry

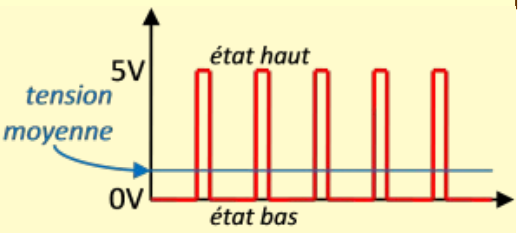
```
http://wiringpi.com/download-and-install/ (lien utile avec les commandes à suivre)
wget https://lion.drogon.net/wiringpi-2.50-1.deb (Récupération du paquet sur la Raspberry)
sudo dpkg -i wiringpi-2.50-1.deb (installation "-i" du paquet sur la Raspberry)

dpkg -L wiringPi > wiringpi.txt (Pour connaître l'ensemble des fichiers installés)
nano wiringpi.txt (supprimer de la liste tous les répertoires sans fichier, sinon tout est intégré, résultat ci-dessous)

/usr/lib/libwiringPiDev.so.2.50
/usr/lib/libwiringPi.so.2.50
/usr/include/mcp23016reg.h
/usr/include/piGlow.h
/usr/include/mcp4802.h
/usr/include/drcNet.h
...
/usr/lib/libwiringPiDev.so
/usr/lib/libwiringPi.so

cat wiringpi.txt (vérifiez que votre travail est correct)
cat wiringpi.txt | xargs tar cvJf wiringpi-pc.tar.xz (création de l'archive pour le PC de développement)
```

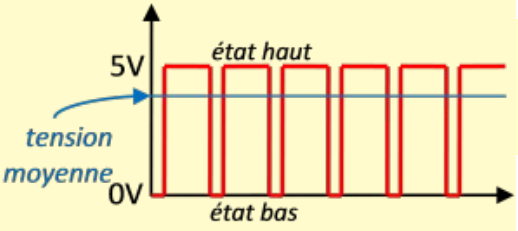
Installation de WiringPi sur le PC de développement (compilation croisée)

```
scp wiringpi-pc.tar.gz manu@172.16.20.31:/home/manu/Public (déploiement de l'archive vers le PC de développement)
sudo tar xJvf wiringpi-pc.tar.xz -C / (installations de tous les fichiers sur le PC de développement)
```

Conversion Numérique/Analogique – PWM

En principe, un port numérique ne peut délivrer qu'une information « tout ou rien » : allumé/éteint. La tension sur la sortie numérique ne peut être que **0V** ou **5V**, jamais entre les deux. Mais, si nous la faisons varier très rapidement (**0V-5V-0V-5V-…**) la valeur moyenne est différente. En jouant sur la durée des états hauts (5V) et états bas (**0V**), nous modifions la valeur de cette tension moyenne, à volonté. Cette technique s'appelle la Modulation de Largeur d'Impulsion (**MLI**) – ou **PWM** (Pulse Width Modulation) en anglais.





Un signal PMW est caractérisé par :

** sa fréquence (environ 500Hz par défaut, mais elle est modifiable)*

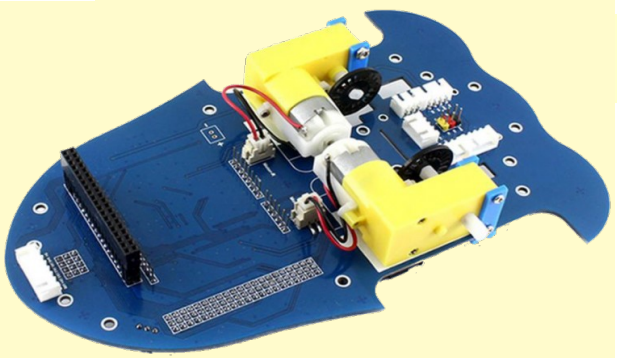
** son amplitude (5V sur un port numérique de Raspberry*

** son rapport cyclique (0% = 0V continu → 100% = 5V continus)*

Les moteurs de l'AlphaBot

La propulsion de **l'AlphaBot** est de type différentielle : deux moteurs indépendants propulsent et dirigent le robot.

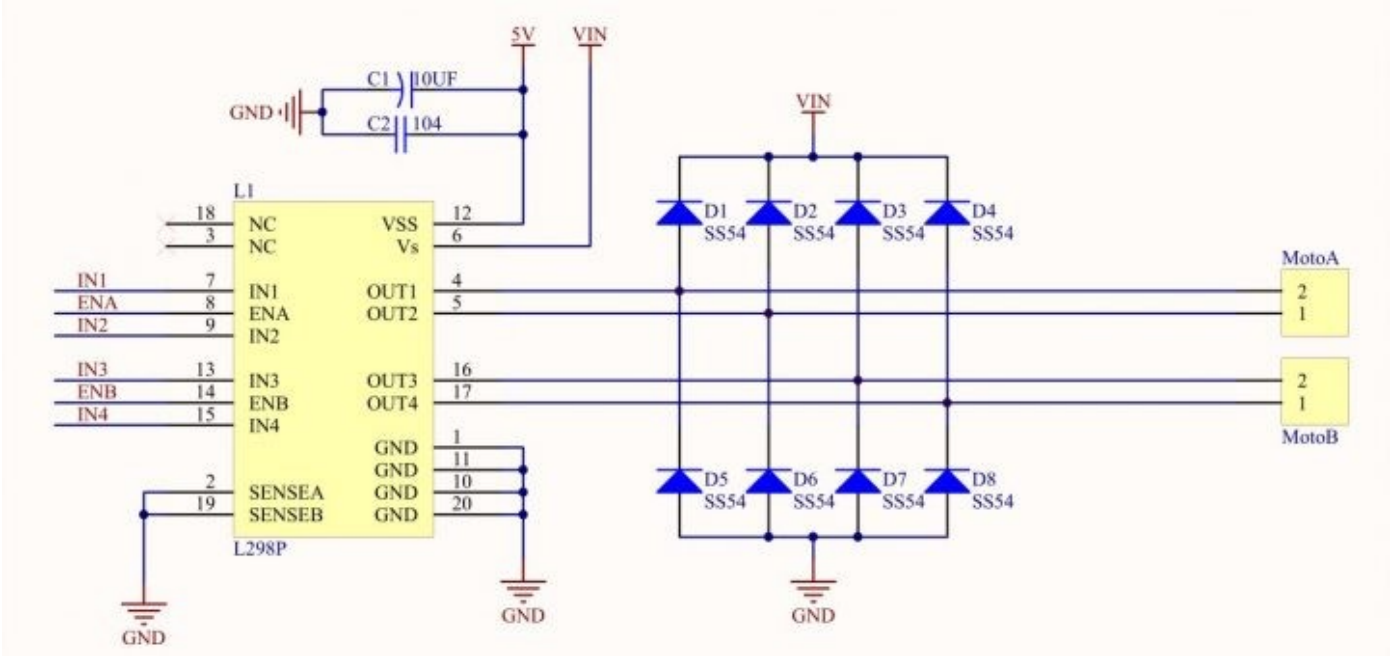
Chaque moteur est piloté par un hacheur de tension qui fait varier la tension moyenne (et donc la vitesse de rotation du moteur) grâce aux signaux PWM vus précédemment.



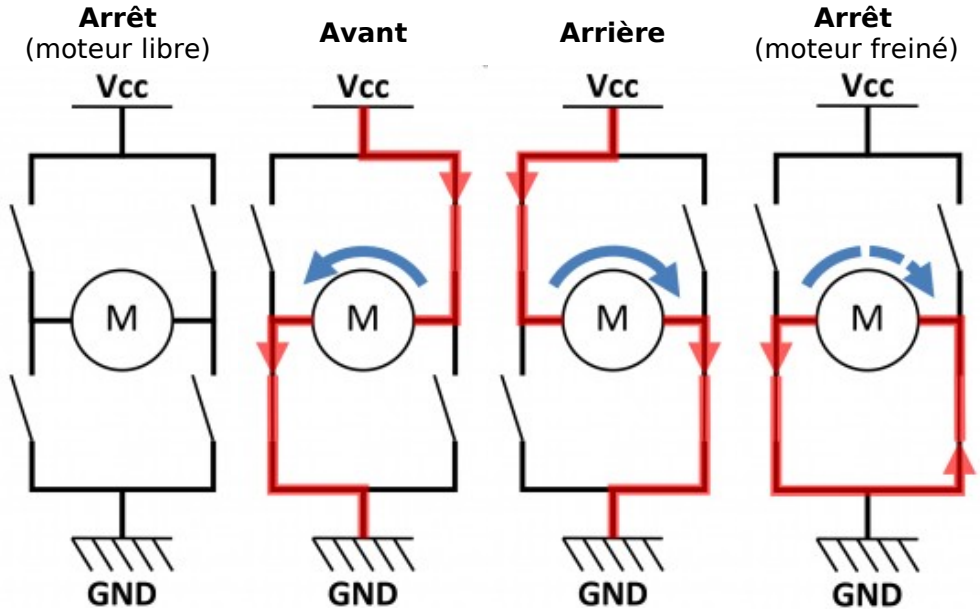
Voici les correspondances entre les ports de communications du Raspberry et les interfaces de commande du pilote moteur :

	Moteur Gauche	Moteur Droit
Nom de l'interface	**ENA**	**ENB**
Port sur la Raspberry	**GPIO6**	**GPIO26**

Il existe également un pont en H (L298P) pour modifier le sens de rotation :



Voici les correspondances entre les ports de communications du Raspberry et les états qu'ils doivent prendre (**LOW** ou **HIGH**) pour obtenir le comportement moteur attendu :



	Moteur Gauche	Moteur Droite	Arrêt (moteur libre)	Avant	Arrière	Arrêt (moteur freiné)
Nom de l'interface	**IN1**	**IN3**				
Port Raspberry	**GPIO12**	**GPIO20**	**LOW**	**HIGH**	**LOW**	pas possible !
Nom de l'interface	**IN2**	**IN4**				
Port Raspberry	**GPIO13**	**GPIO21**	**LOW**	**LOW**	**HIGH**	pas possible !

Les codeurs incrémentaux - description

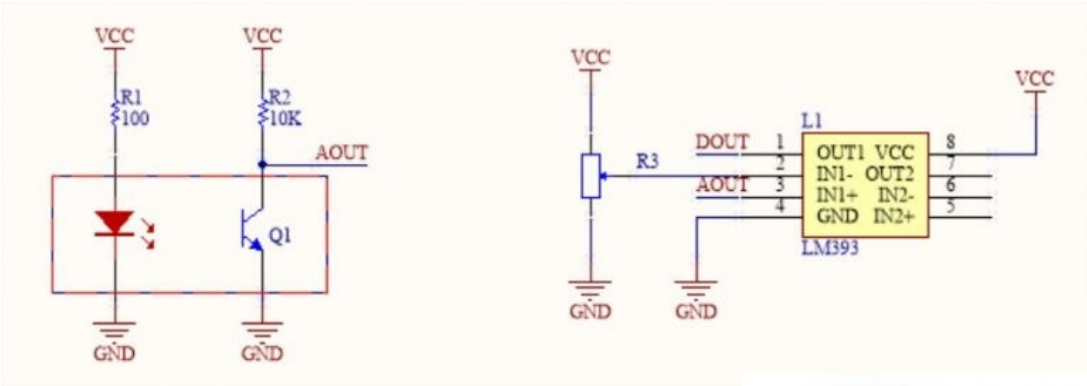
Chaque roue étant équipée d'un codeur incrémental, il est donc possible de commander le mouvement en distance plutôt qu'en durée.

> ***Attention : les contrôleurs des moteurs n'autorisent pas le freinage ! Il y a donc un fort risque de dépassement des positions visées !***
>
> ***Les disques des codeurs sont sur le même axe que les roues, et comportent 20 fentes :***

	Codeur Gauche	Codeur Droit
Nom de l'interface	**CE0**	**CE1**
Port sur la Raspberry	**GPIO8**	**GPIO7**

Détecteur de proximité à infrarouge

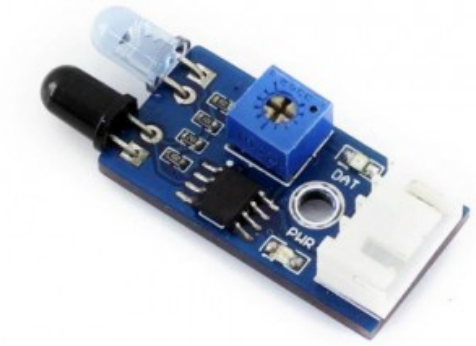
Les détecteurs utilisent une diode IR, un photo-transistor et un un montage comparateur associé à un potentiomètre pour détecter un obstacle.



Le potentiomètre permet de régler le seuil de détection (distance).

Sur l'AlphaBot, les modules détecteur IR sont branchés sur les ports GPIO 16 et 19. Le comparateur LM393 donnant une information logique, nous pouvons connaître l'état des capteurs en lisant directement l'état des broches 16 et 19 . Il faut activer la résistance de Pull-Up sur la broche DOUT du comparateur LM393.

La détection d'un obstacle est un niveau bas.



	Détecteur Gauche	Détecteur Droit
Nom de l'interface	**IRL**	**IRR**
Port sur la Raspberry	**GPIO19**	**GPIO16**

Programme expérimental sur le Raspberry de l'AlphaBot

Je vous propose un petit programme qui permet de tester ces différents systèmes, les moteurs, les codeurs incrémentaux et les détecteurs de proximité afin de bien maîtriser les fonctions utiles de la bibliothèque **wiringPi**.

alphabot.pro

```
TEMPLATE = app
CONFIG += console c++11
CONFIG -= app_bundle qt
SOURCES += main.cpp

LIBS += -lwiringPi -lpthread   (Intégration de la bibliothèque WiringPI – ne pas oublier)
target.path = /home/pi/c++
INSTALLS += target
```

main.cpp

```
#include <iostream>
#include <wiringPi.h>
#include <softPwm.h>
using namespace std;

// Ports de contrôle du moteur gauche
const int MG=6; // Activation du moteur avec rapport cyclique variable
const int MGAvant=12; // Déplacement du moteur gauche vers l'avant
const int MGArriere=13; // Déplacement du moteur gauche vers l'arrière

// Ports de contrôle du moteur droit
const int MD=26; // Activation du moteur avec rapport cyclique variable
const int MDArriere=20; // Déplacement du moteur droit vers l'arrière
const int MDAvant=21; // Déplacement du moteur droit vers l'avant

// Codeurs incrémentaux
const int CDG=8; // Codeur incrémental Gauche (CE0)
const int CDD=7; // Codeur incrémental Droit (CE1)

// Détecteurs de proximité Infra-rouge
// Il faut activer la résistance de tirage sur la broche DOUT du comparateur LM393.
// La détection d'un obstacle est un niveau bas :
const int IRD=16;
const int IRG=19;
```

```
void deplacementRoueGauche()
{
  static int nbImpuls=0;
  cout << "Gauche : " << ++nbImpuls << endl;
}

void deplacementRoueDroite()
{
  static int nbImpuls=0;
  cout << "Droite : " << ++nbImpuls << endl;
}

int main()
{
  // Initialisation de wiringPi
  wiringPiSetupGpio();

  // Prise en compte des différentes broches du moteur
  softPwmCreate(MG, 0, 100);
  softPwmCreate(MD, 0, 100);
  pinMode(MGAvant, OUTPUT);
  pinMode(MGArriere, OUTPUT);
  pinMode(MDAvant, OUTPUT);
  pinMode(MDArriere, OUTPUT);

  // Prise en compte des capteurs incrémentaux
  pinMode(CDG, INPUT);
  pinMode(CDD, INPUT);

  // Lancement de fonctions à chaque front montant des capteurs incrémentaux
  // Montant->INT_EDGE_RISING, Descendant->INT_EDGE_FALLING, Les deux->INT_EDGE_BOTH
  wiringPiISR(CDG, INT_EDGE_RISING, &deplacementRoueGauche);
  wiringPiISR(CDD, INT_EDGE_RISING, &deplacementRoueDroite);

  // Prise en compte des détecteurs de présence
  pinMode(IRG, INPUT);
  pinMode(IRD, INPUT);

  // Prise en compte des résistances de tirage montée sur l'alimentation
  pullUpDnControl(IRG, PUD_UP);
  pullUpDnControl(IRD, PUD_UP);

  // démarrer le robot vers l'avant
  digitalWrite(MGAvant, HIGH);
  digitalWrite(MGArriere, LOW);
  digitalWrite(MDAvant, HIGH);
  digitalWrite(MDArriere, LOW);
  softPwmWrite(MG, 40);
  softPwmWrite(MD, 40);

  for (int i=0; i<10; i++) {
    cout << "IRG : " << digitalRead(IRG) << ' ';
    cout << "IRD : " << digitalRead(IRD) << endl;
    delay(300);
  }

  digitalWrite(MGAvant, LOW);
  digitalWrite(MDAvant, LOW);
  softPwmStop(MG);
  softPwmStop(MD);

  return 0;
}
```

Dans votre projet, pensez à inclure votre librairie. Ensuite, sur votre source qui doit exploiter les compétences de cette librairie, vous devez inclure les en-têtes « wiringPi.h » (pour le mode tout ou rien) et « softPwm.h » (pour le mode PWM).

Avant de manipuler les broches du GPIO, avec cette librairie, il est nécessaire qu'elles soient activées, grâce à la fonction wiringPiSetupGpio().

Pour utiliser les broches du GPIO en mode tout ou rien, vous devez au préalable fixer la direction (entrée ou sortie) de chaque broche, là aussi au moyen d'une fonction spécifique qui s'appelle pinMode() qui prend deux arguments : le numéro de la broche et la direction avec des constantes toutes faites INPUT, OUTPUT.

Une fois que le mode est fixé pour chacune des broches, vous pouvez lire l'état de la broche concernée au moyen de la fonction digitalRead() avec en argument le numéro correspondant. Cette fonction renvoie les valeur HIGH (1) ou LOW (0) suivant l'état de la broche.

Lorsqu'une broche du GPIO est en sortie, si vous désirez modifier son état vous passez par la fonction digitalWrite() en spécifiant la broche concernée et la valeur de l'état souhaité (HIGH ou LOW).

Si vous souhaitez plutôt utiliser le mode PWM sur certaines sorties du GPIO, vous devez prendre d'autres fonctions qui reprennent le même principe que précédemment, en fixant le mode choisi en préalable sur la broche concernée et en proposant le bon rapport cyclique par la suite.

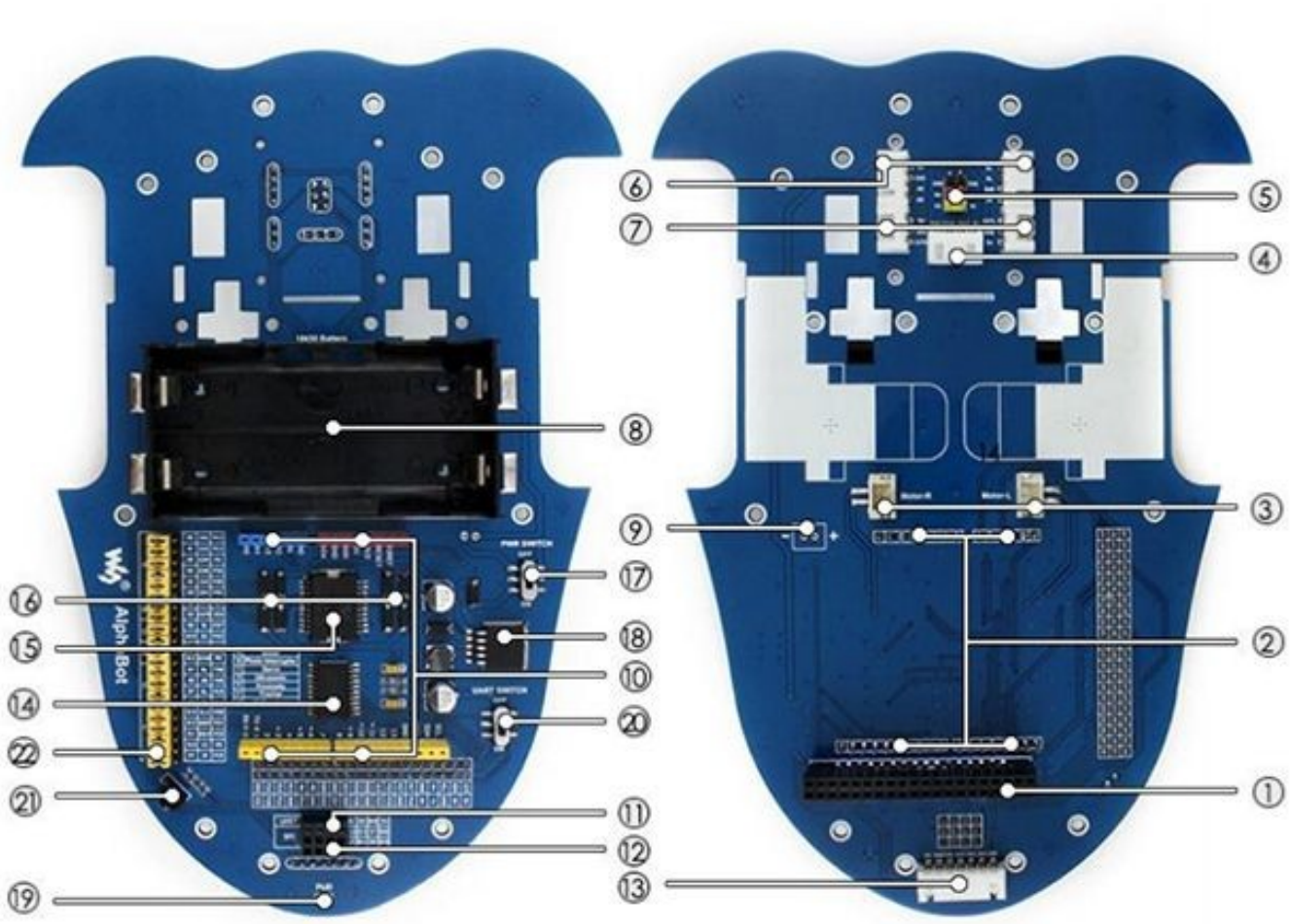
Pour activer le mode PWM à une broche particulière, vous devez utiliser la fonction softPwmCreate() en spécifiant le numéro de la broche, la valeur de départ et la valeur maxi du rapport cyclique.

Par la suite, sur cette même broche, vous pouvez modifier la valeur du rapport cyclique en utilisant la fonction softPwmWrite() en spécifiant le numéro et la nouvelle valeur du rapport cyclique.

Enfin, si vous n'avez plus besoin d'utiliser la fonctionnalité du mode PWM, il est souhaitable de le désactiver (consomme un peu plus d'énergie) grâce à la fonction softPwmStop() en spécifiant le numéro de la broche.

Il faut activer la résistance de tirage sur la broche DOUT du comparateur LM393 grâce à la méthode pullUpDnControl() en spécifiant le numéro de la broche ainsi que le type de connexion de la résistance de tirage (+Vcc → PUL_UP, 0v → PUL_DOWN).

Pour la prise en compte des fronts de montée ou des fronts de descente (ou les deux), la fonction s'appelle wiringPiISR(). Cette fonction est très particulière puisqu'elle active une autre fonction de rappel automatiquement à chaque front du capteur (mécanisme d'interuption). Elle prend trois paramètres, le numéro de la broche, le type de front (Montant->INT_EDGE_RISING, Descendant->INT_EDGE_FALLING, Les deux→INT_EDGE_BOTH) et enfin l'adresse de la fonction à appeler à chaque événement.



1 - Interface Raspberry Pi
2 - Interface Arduino
3 - Interfaces pour les deux moteurs
4 - Interface pour le module US
5 - Interfce pour servos
6 - Inteface pour leds IR de détection d'obstacle
7 - Interface pour les encodeurs
8 - Support pour 2 accus 18650
9 - Pastille à souder pour le raccordement d'une autre source d'alimentation
10 - Connecteur pour Shield compatible Arduino
11 - Interface UART pour module Bluetooth
12 - Interface SPI pour module sans fil NRF24L01
13 - Interface pour module suiveur de ligne
14 - Circuit d'acquisition permettant une carte Raspberry Pi d'utiliser des capteurs analogiques
15 - L298P, double pont en H jusqu'à 2 A par moteur
16 - Diode contre les inversions de polarités
17 - Interrupteur marche-arrêt
18 - LM2596, régulateur 5 Vcc
19 - Led d'indication d'alimentation
20 - Interrupteur UART: permet d'établir la communication série entre l'Arduino et la carte Raspberry Pi
21 - Récepteur IR
22 - Sélection du µC de contrôle (Arduino ou Pi)

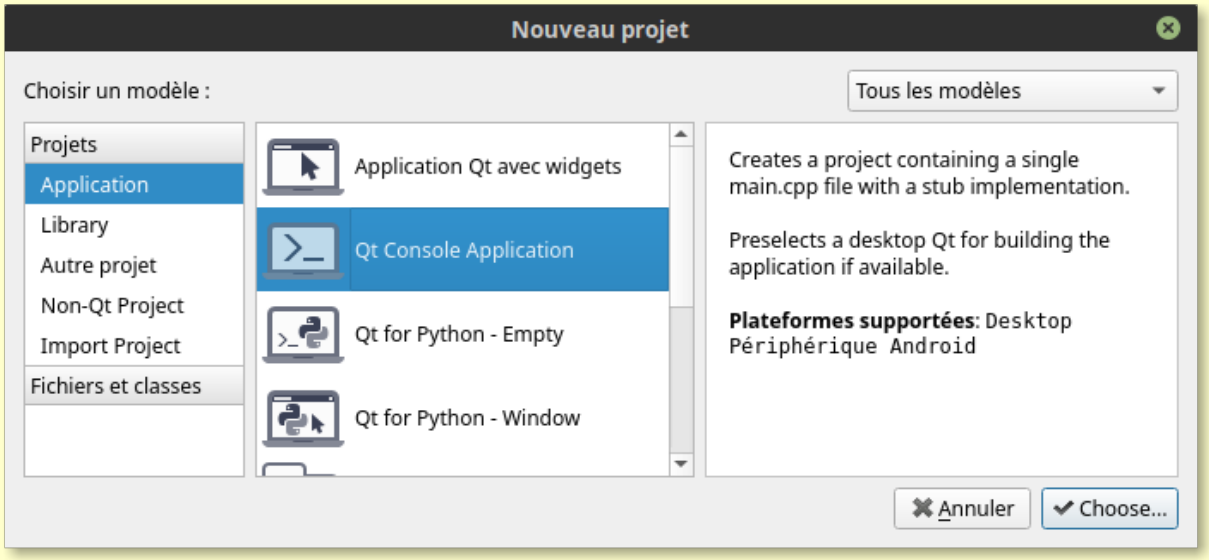
Piloter le robot à distance depuis un smartphone

Après ce petit programme de test, je vous propose maintenant de piloter notre robot **AlphaBot** à distance à l'aide d'un smartphone ou d'une application cliente sur PC. Pour cela, vous devez au préalable générer un service sur le robot en utilisant le protocole « ***websocket*** ».

Dans un premier temps, nous n'utiliserons pas toutes les possibilités du robot, notamment les capteurs incrémentaux. Par contre, nous tiendrons compte des capteurs de proximité afin que le robot s'arrête automatiquement devant les obstacles éventuels.

Pour réaliser ce service, nous prévoyons plusieurs modules afin de bien séparer les différentes tâches et de bien maîtriser les fonctionnalités des différents systèmes intégrés au robot. Ces modules seront représentés par des classes spécifiques avec des noms logiques suivant les dispositifs évoqués : Moteur, Détecteur, AlphaBot et Service.

Enfin, pour prendre en compte le protocole « websocket », nous devons choisir un projet de type « Qt Console ».



alphabot.pro

```
QT += websockets
QT -= gui

CONFIG += c++11 console
CONFIG -= app_bundle

DEFINES += QT_DEPRECATED_WARNINGS

SOURCES += main.cpp moteur.cpp alphabot.cpp service.cpp
HEADERS += moteur.h detecteur.h alphabot.h service.h
LIBS += -lwiringPi -lpthread

target.path = /home/pi/c++
INSTALLS += target
```

detecteur.h

```
#ifndef DETECTEUR_H
#define DETECTEUR_H

#include <wiringPi.h>

// Détecteurs de proximité Infra-rouge
// Il faut activer la résistance de tirage sur la broche DOUT du comparateur LM393.
// La détection d'un obstacle est un niveau bas :

const int IRD=16;
const int IRG=19;

class Detecteur
{
  int capteur;
public:
  Detecteur(int capteur) : capteur(capteur)
  {
    pullUpDnControl(capteur, PUD_UP);
    pinMode(capteur, INPUT);
  }
  bool obstacle() { return digitalRead(capteur)==LOW; }
};

#endif // DETECTEUR_H
```

Cette classe permet de représenter les deux capteurs de proximité de l'AlphaBot. Deux objets seront donc créés par la classe AlphaBot identifiant chacun de ces capteurs. Le constructeur attend le numéro de broche correspondant au détecteur. La méthode obstacle(), comme son nom l'indique, nous renseigne sur la présence d'un obstacle ou non (détection au niveau bas).

moteur.h

```
#ifndef MOTEUR_H
#define MOTEUR_H

#include <wiringPi.h>
#include <softPwm.h>

// Ports de contrôle du moteur gauche
const int MG=6; // Activation du moteur avec rapport cyclique variable
const int MGAvant=12; // Déplacement du moteur gauche vers l'avant
const int MGArriere=13; // Déplacement du moteur gauche vers l'arrière
```

```
// Ports de contrôle du moteur droit
const int MD=26; // Activation du moteur avec rapport cyclique variable
const int MDArriere=20; // Déplacement du moteur droit vers l'arrière
const int MDAvant=21; // Déplacement du moteur droit vers l'avant

class Moteur
{
  int moteur, versAvant, versArriere;
public:
  Moteur(int moteur, int versAvant, int versArriere);
  ~Moteur();
  void avant(int vitesse);
  void arriere(int vitesse);
  void arret();
};

#endif // MOTEUR_H
```

La classe Moteur représente l'un des deux moteurs du robot. Pour le bon fonctionnement, je rappelle que chaque moteur doit connaître la broche qui permet de régler le rapport cyclique afin de maîtriser la vitesse de rotation et les deux broches de commutation qui permettent d'indiquer le sens de rotation. Toutes ces indications doivent être précisées lors de la construction.

Vous remarquez la présence d'un destructeur qui permet d'arrêter définitivement la gestion du rapport cyclique sur la broche concernée.

Nous avons ensuite deux méthodes, avant() et arriere(), qui contrôlent les deux sens de rotation en spécifiant la vitesse désirée et une méthode, arret(), qui permet de stopper le moteur.

moteur.cpp

```
#include "moteur.h"

Moteur::Moteur(int moteur, int versAvant, int versArriere) : moteur(moteur), versAvant(versAvant), versArriere(versArriere)
{
  softPwmCreate(moteur, 0, 100);
  pinMode(versAvant, OUTPUT);
  pinMode(versArriere, OUTPUT);
}

Moteur::~Moteur()
{
  arret();
  softPwmStop(moteur);
}

void Moteur::avant(int vitesse)
{
  digitalWrite(versAvant, HIGH);
  digitalWrite(versArriere, LOW);
  softPwmWrite(moteur, vitesse);
}

void Moteur::arriere(int vitesse)
{
  digitalWrite(versArriere, HIGH);
  digitalWrite(versAvant, LOW);
  softPwmWrite(moteur, vitesse);
}

void Moteur::arret()
{
  digitalWrite(versArriere, LOW);
  digitalWrite(versAvant, LOW);
  softPwmWrite(moteur, 0);
}
```

La structure de cette classe Moteur est vraiment très simple et surtout intuitive. Nous remarquons que chacune de ces méthodes utilisent pleinement les compétences de la librairie « wiringPi » qui n'est plus accessible côté utilisateur (notion d'encapsulation). Ce que nous avons besoin pour piloter un moteur c'est de pouvoir le faire tourner dans un sens ou dans l'autre à une certaine vitesse ou de l'arrêter. C'est là l'intérêt de la programmation objet.

La classe suivante AlphaBot utilise ces deux classes que nous venons de décrire, Detecteur et Moteur. Elle crée tous les objets nécessaires à la gestion complète du robot, avec les deux détecteur de proximité et les deux moteurs. Nous prévoyons un fonctionnement spécifique du robot. Nous choisissons ainsi qu'il peut se déplacer vers l'avant, il peut reculer, il peut tourner à droite ou à gauche en ne faisant fonctionner qu'un seul moteur. Il peut également avancer à droite ou à gauche avec pour cela des vitesses différentes sur les deux moteurs.

Tous ces déplacements peuvent se faire en choisissant les vitesses de déplacement. La vitesse est un attribut à part entière. Les différentes déplacements tiendrons compte de la vitesse présélectionnée. Il est bien sûr possible de changer de vitesse en cours de déplacement, la méthode changeVitesse() s'occupe de cela. Il faut connaître à tout moment qu'elle type d'action est en cours, c'est pour cela que nous avons construit une énumération qui spécifie l'action de mouvement (ou pas).

Vous remarquez que la classe AlphaBot hérite de la classe QObject. Cela est nécessaire afin de prendre en compte la gestion événementielle associée au timer interne. Tout les 1/10s nous vérifions l'état des détecteurs afin d'arrêter le robot dans le cas de la présence d'un obstacle.

alphabot.h

```
#ifndef ALPHABOT_H
#define ALPHABOT_H

#include <QObject>
#include <moteur.h>
```

```
#include <detecteur.h>

class AlphaBot : public QObject
{
  Q_OBJECT
  enum {Avant, AvantDroite, Droite, AvantGauche, Gauche, Arriere, Arret} action = Arret;
public:
  explicit AlphaBot(QObject *parent = nullptr);
  void avant();
  void avantDroite();
  void droite();
  void avantGauche();
  void gauche();
  void arriere();
  void arret();
  void changeVitesse(int nouvelle);
protected:
  void timerEvent(QTimerEvent*);
private:
  int vitesse=40;
  Moteur moteurGauche, moteurDroit;
  Detecteur detecteurGauche, detecteurDroit;
};

#endif // ALPHABOT_H
```

alphabot.cpp

```
#include "alphabot.h"

AlphaBot::AlphaBot(QObject *parent) : QObject(parent),
  moteurGauche(MG, MGAvant, MGArriere), moteurDroit(MD, MDAvant, MDArriere), detecteurGauche(IRG), detecteurDroit(IRD)
{
  startTimer(100);
}

void AlphaBot::avant()
{
  action = Avant;
  moteurGauche.avant(vitesse);
  moteurDroit.avant(vitesse);
}

void AlphaBot::avantDroite()
{
  action = AvantDroite;
  moteurGauche.avant(vitesse);
  moteurDroit.avant(vitesse/2);
}

void AlphaBot::droite()
{
  action = Droite;
  moteurGauche.avant(vitesse);
  moteurDroit.avant(0);
}

void AlphaBot::avantGauche()
{
  action = AvantGauche;
  moteurGauche.avant(vitesse/2);
  moteurDroit.avant(vitesse);
}

void AlphaBot::gauche()
{
  action = Gauche;
  moteurGauche.avant(0);
  moteurDroit.avant(vitesse);
}

void AlphaBot::arriere()
{
  action = Arriere;
  moteurGauche.arriere(vitesse/2);
  moteurDroit.arriere(vitesse/2);
}

void AlphaBot::arret()
{
  action = Arret;
  moteurGauche.arret();
  moteurDroit.arret();
}

void AlphaBot::changeVitesse(int nouvelle)
{
  if (nouvelle<30 || nouvelle>100) return;
  vitesse=nouvelle;
  switch (action) {
    case Avant : avant(); break;
    case AvantDroite : avantDroite(); break;
```

```
    case Droite: droite(); break;
    case AvantGauche: avantGauche(); break;
    case Gauche: gauche(); break;
    case Arriere: arriere(); break;
  }
}

void AlphaBot::timerEvent(QTimerEvent *)
{
  if (action==Arriere) return;
  if (detecteurDroit.obstacle() || detecteurGauche.obstacle()) arret();
}
```

La classe Service s'occuper de la gestion du réseau en mettant un place un service WebSocket avec le protocole associé. C'est elle qui crée l'objet représentant le robot AlphaBot. Une fois la connexion établie, l'utilisateur proposera les différentes commandes associées au déplacement du robot, grâce à la méthode commande(). Il s'agit d'un « slot » automatiquement activé dès qu'une information sous forme de chaîne de caractères est envoyée par le réseau. Il s'agit de commandes très simples :

- *av : pour le déplacement vers l'avant,*
- *ar : pour reculer,*
- *dr : pour tourner à droite,*
- *ga : pour tourner à gauche,*
- *avd : pour tourner légèrement à droite,*
- *avg : pour tourner légèrement à gauche,*
- *stop : pour arrêter le déplacement du robot,*
- *quit : pour stopper définitivement le service à distance.*

service.h

```
#ifndef SERVICE_H
#define SERVICE_H

#include <QObject>
#include <QWebSocketServer>
#include <QWebSocket>
#include <alphabot.h>

class Service : public QObject
{
  Q_OBJECT
public:
  explicit Service(QObject *parent = nullptr);
signals:
  void quit();
private slots:
  void connexion();
  void deconnexion();
  void commande(const QString& demande);
private:
  AlphaBot robot;
  QWebSocketServer service;
};

#endif // SERVICE_H
```

service.cpp

```
#include "service.h"

Service::Service(QObject *parent) : QObject(parent), service("gpio", QWebSocketServer::NonSecureMode, this)
{
  if (service.listen(QHostAddress::Any, 8080))
    connect(&service, SIGNAL(newConnection()), this, SLOT(connexion()));
}

void Service::connexion()
{
  QWebSocket *client = service.nextPendingConnection();
  connect(client, SIGNAL(textMessageReceived(QString)), this, SLOT(commande(QString)));
  connect(client, SIGNAL(disconnected()), this, SLOT(deconnexion()));
  client->sendTextMessage("Connexion avec l'AlphaBot !");
}

void Service::deconnexion()
{
  QWebSocket *client = (QWebSocket *) sender();
  client->deleteLater();
  disconnect(client, SIGNAL(textMessageReceived(QString)), this, SLOT(commande(QString)));
  disconnect(client, SIGNAL(disconnected()), this, SLOT(deconnexion()));
  robot.arret();
}

void Service::commande(const QString &demande)
{
  if (demande=="quit") quit();
  else if (demande=="av")   robot.avant();
```

```
  else if (demande=="avg")  robot.avantGauche();
  else if (demande=="ga")   robot.gauche();
  else if (demande=="avd")  robot.avantDroite();
  else if (demande=="dr")   robot.droite();
  else if (demande=="ar")   robot.arriere();
  else if (demande=="stop") robot.arret();
  else robot.changeVitesse(demande.toInt());
}
```

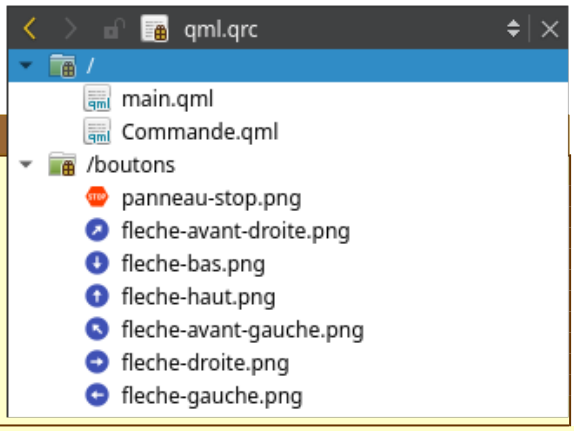
Attention, avant de pouvoir utiliser toutes ces classes qui encapsulent toutes les compétences de « wiringPi », vous devez initialiser correctement cette bibliothèque, pour que les différents bus du Raspberry soient opérationnels, grâce à l'appel de la fonction wiringPiSetupGpio() que vous lancez dans la fonction principale de cette application (avant la création du service qui crée lui même l'objet représentant le robot).

main.cpp

```
#include <QCoreApplication>
#include <service.h>
#include <wiringPi.h>

int main(int argc, char *argv[])
{
  QCoreApplication a(argc, argv);
  wiringPiSetupGpio();
  Service alphabot;
  alphabot.connect(&alphabot, SIGNAL(quit()), &a, SLOT(quit()));
  return a.exec();
}
```

Application cliente qui gère les déplacement à distance qui utilise ce service intégré dans le robot

Nous pouvons maintenant prévoir un nouveau projet multi-plateforme utilisable soit au travers d'une application cliente sur un PC classique, soit au travers de votre smartphone. Cela ne peut se faire qu'à partir d'un projet de type **Quick**.



alphabot-client.pro

```
QT += quick websockets

CONFIG += c++11
DEFINES += QT_DEPRECATED_WARNINGS

SOURCES += main.cpp
RESOURCES += qml.qrc

# Default rules for deployment.
qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target
```

Mis à part le fichier « main.cpp », tout se passe dans les ressources QML, même pour la gestion du réseau au travers de la classe WebSocket. La première ressource crée des boutons personnalisés correspondant aux différentes commandes du robot. La deuxième ressource gère l'ensemble de l'application cliente.

Commande.qml

```
import QtQuick 2.0
import QtQuick.Controls 2.12

ToolButton {
  id: bouton
  property alias icone: bouton.icon.source

  width: 96
  height: 96
  background: Rectangle { color: "transparent" }

  icon {
    width: parent.width
    height: parent.height
    color: "transparent"
  }
}
```

main.qml

```
import QtQuick 2.12
import QtQuick.Window 2.12
import QtQuick.Controls 2.5
import QtGraphicalEffects 1.0
import QtWebSockets 1.13

Window {
  visible: true
  width: 320
  height: 480
  title: qsTr("AlphaBot")

  RadialGradient {
```

```
    anchors.fill: parent
    gradient: Gradient {
      GradientStop { position: 0.0; color: "lightblue" }
      GradientStop { position: 1.0; color: "blue" }
    }
  }

  WebSocket {
    id: alphabot
    onTextMessageReceived: adresse.text = message
    onStatusChanged: if (status==WebSocket.Error) {
      adresse.text = "AlphaBot inaccessible !"
      connexion.text = "Réinitialiser"
    }
  }

  TextField {
    id: adresse
    anchors {
      top: parent.top
      topMargin: 8
      left: parent.left
      leftMargin: 8
      right: connexion.left
      rightMargin: 8
    }
    placeholderText: qsTr("@IP")
    color: "darkblue"
    font.pixelSize: 18
    font.bold: true
    inputMethodHints: Qt.ImhDigitsOnly
  }

  ToolButton {
    id: connexion
    anchors {
      top: parent.top
      topMargin: 8
      right: parent.right
      rightMargin: 8
    }
    text: qsTr("Connexion")
    font.bold: true
    checkable: true
    onCheckedChanged:
      if (checked) {
        alphabot.url = "ws://"+adresse.text+":8080/"
        alphabot.active = true
        text = "Déconnexion"
      }
      else {
        alphabot.active = false
        text = "Connexion"
        adresse.text = ""
      }
  }

  Commande {
    id: stop
    anchors.centerIn: parent;
    icone: "boutons/panneau-stop.png"
    onClicked: alphabot.sendTextMessage("stop")
  }

  Commande {
    id: avant
    anchors {
      horizontalCenter: parent.horizontalCenter
      bottom: stop.top
      bottomMargin: 50
    }
    icone: "boutons/fleche-haut.png"
    onClicked: alphabot.sendTextMessage("av")
  }

  Commande {
    anchors {
      horizontalCenter: parent.horizontalCenter
      top: stop.bottom
      topMargin: 35
    }
    icone: "boutons/fleche-bas.png"
    onClicked: alphabot.sendTextMessage("ar")
  }
  Commande {
    anchors {
      left: avant.right
      bottom: stop.top
    }
    icone: "boutons/fleche-avant-droite.png"
    onClicked: alphabot.sendTextMessage("avd")
  }
```

```
  Commande {
    anchors {
      right: avant.left
      bottom: stop.top
    }
    icone: "boutons/fleche-avant-gauche.png"
    onClicked: alphabot.sendTextMessage("avg")
  }
  Commande {
    anchors {
      verticalCenter: parent.verticalCenter
      left: stop.right
      leftMargin: 35
    }
    icone: "boutons/fleche-droite.png"
    onClicked: alphabot.sendTextMessage("dr")
  }
  Commande {
    anchors {
      verticalCenter: parent.verticalCenter
      right: stop.left
      rightMargin: 35
    }
    icone: "boutons/fleche-gauche.png"
    onClicked: alphabot.sendTextMessage("ga")
  }

  SpinBox {
    anchors {
      bottom: parent.bottom
      left: parent.left
      right: parent.right
      margins: 80
      bottomMargin: 40
    }
    font.bold: true
    font.pixelSize: 40
    value: 40
    stepSize: 5
    from: 30
    to: 100
    onValueChanged: alphabot.sendTextMessage(value)
  }
}
```