ROB - UE 3.2 Découverte - Contrôle Bas Niveau d'un Robot Mobile

B. Zerr

2020-2021- S3

1 Objectif du TE/TD

L'objectif de ce TD/BE est d'être capable de contrôler le robot DART V2 afin de parcourir un circuit de façon entièrement autonome en utilisant uniquement les capteurs à ultrasons et les odomètres. La programmation se fera en Python3. La mise au point se fera sur le robot DART V2 virtuel et les tests se feront sur le robot réel. Il n'y a (en principe) pas de changements à faire dans le code Python pour passer du robot virtuel au robot réel. Ce travail s'effectue en binome. Le résultat de ce travail servira de base aux épreuves de qualification du DART Challenge 2020-2021. L'approche est progressive, en plusieurs étapes :

- étape 1 : fonctions de base du robot DART V2 : commande des moteurs et acquisition des informations issues des capteurs (sonars et odomètres),
- étape 2 : fonctions simples de déplacement du robot DART V2 : ligne droite, rotation sur place.
- étape 3 : fonctions de déplacement tenant compte des l'environnement : évitement des obstacles, suivi de mur.

Nous utiliserons le système d'exploitation Linux.

2 Installation du logiciel DART

Le logiciel DART V2 permet l'exécution de votre programme sur les robots virtuels et réels. Placez vous dans le dossier ou vous allez travailler (dans les commandes suivantes, il faudra remplacer my_working_path par le vrai nom de votre dossier). Récupérez sur le dossier /public/share/ue32rob le code de contrôle du robot et le simulateur V-REP, puis le décompresser dans votre dossier de travail.

```
cd my_working_path
wget https://www.ensta-bretagne.fr/zerr/filerepo/dartv2/dartv2-td1-20201412-UE32.tgz
tar xf dartv2-td1-20201412-UE32.tgz
cd dartv2/py
```

Ouvrez un second terminal pour lancer le simulateur V-REP :

```
cd my_working_path/dartv2/V-REP_PRO_EDU_V3_3_2_64_Linux
./vrep.sh
```

Chargez la scène "dartv2_ue31_2020_challenge.ttt" se trouvant dans le dossier "scenes" (remonter de 2 niveaux de dossiers pour accèder au dossier "scenes").

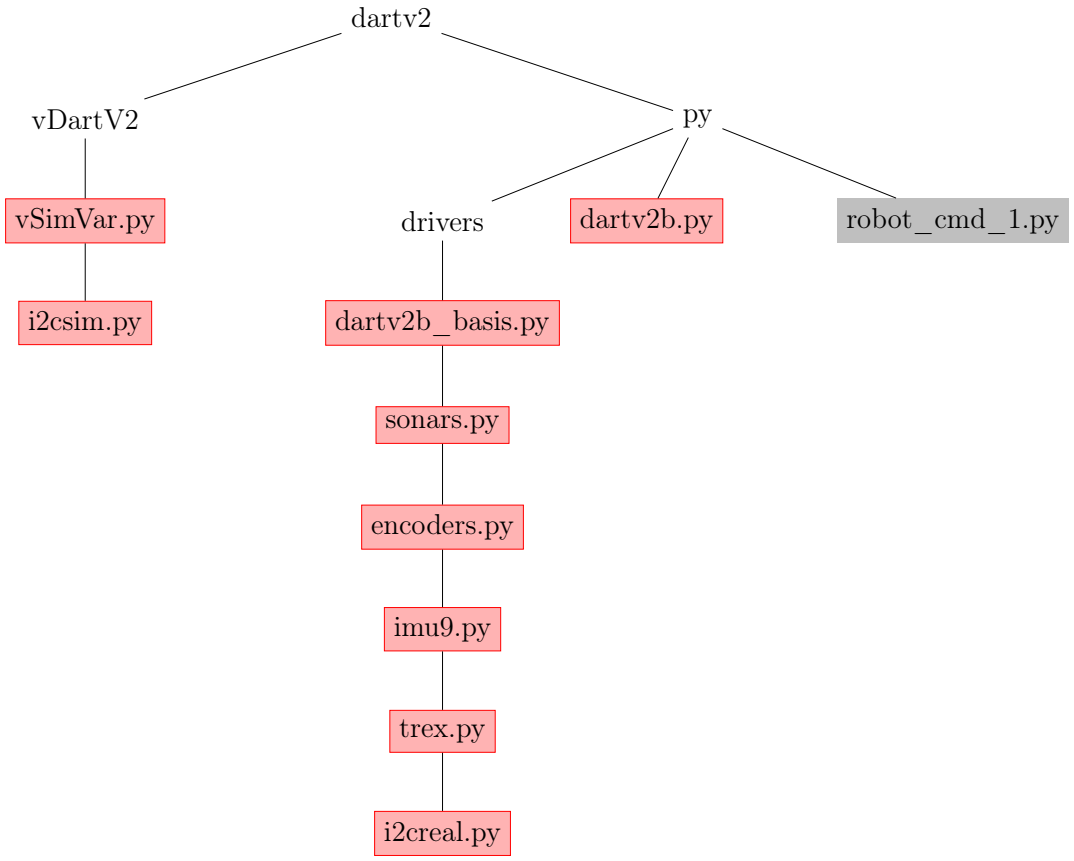


FIGURE 1 – Architecture du logiciel de contrôle du robot DART V2.

3 Fonctions de base du robot DART

Nous allons dans cette partie concevoir et implémenter les outils (fonctions Python) qui nous permettront d'accéder aux éléments matériels de base (capteurs et actionneurs) du robot DART.

3.1 Architecture logicielle du robot Dart V2

L'architecture logicielle du robot DART V2 est représentée à la figure 1. La classe DartV2 dans le fichier dartv2b.py donne accès aux robots DART V2 réels et virtuels. L'accès au capteurs se fait à travers de programmes que l'on appelle drivers. Par exemple, pour accèder aux sonars, nous aurons deux fichiers drivers à complèter : sonars.py (dans le dossier dartv2/py/drivers). Dans un premier temps, vous allez travailler uniquement sur le robot virtuel. Voici un exemple de programme de contrôle ("robot_cmd_1.py" dans le dossier py) :

```
import dartv2b

if __name__ == "__main__":
    mybot = dartv2b.DartV2()

    # place your work here

    mybot.end() # clean end of the robot mission
```

Ce programme définit l'objet **mybot** instance de la classe **DartV2** qui contient l'ensemble des fonctions de contrôle du robot DART V2 (virtuel ou réel).

L'objet **mybot** possède déjà quelques fonctions. La fontion **end()** permet de terminer proprement l'exécution. Il est aussi possible de terminer l'exécution avec **Ctrl-C**. Après avoir démarrer le simulateur V-REP, vous pouvez exécuter ce programme en exécutant simplement le script depuis la ligne de commande (si la commande python3 n'existe pas elle peut être remplacée par python3.5 , python3.6 ou +) :

```
cd py
python3 robot_cmd_1.py
```

Tout devrait bien se passer et le résultat devrait être le suivant :

```
Work with virtual DART on V-REP
init i2c device addr = 0x21 on bus 2
init i2c device addr = 0x70 on bus 2
init i2c device addr = 0x72 on bus 2
init i2c device addr = 0x14 on bus 2
init i2c device addr = 0x1e on bus 2
init i2c device addr = 0x6b on bus 2
vDart ready ...
... end
```

3.2 Commande des moteurs

Pour contrôler le déplacement du robot, la première chose à faire est de faire tourner les moteurs. Les moteurs sont commandés par une carte de puissance (T-REX) pilotée par un microcontrôleur. Cette carte est reliée à la carte principale du robot (PCDuino3) par le bus I2C. La documentation de la carte T-Rex de commande des moteurs disponible sous MOODLE ou aux liens suivants :

```
general description:
https://www.sparkfun.com/products/retired/12075

user manual:
https://cdn.sparkfun.com/datasheets/Robotics/
        T%27REX%20robot%20controller%20instruction%20manual1.pdf
```

La communication avec la carte se fait en envoyant un **"command data packet"** de 27 octets et en recevant un **"command status packet"** de 24 octets. Par rapport à la version originale, la commande de la carte T-Rex a été simplifiée et elle peut être commandée par seulement 6 octets. La commande se fait à l'aide d'un dictionnaire situé dans le driver (fichier trex.py). Pour ce premier driver, nous n'allons pas directement programmer les lectures et écritures sur le bus I2C mais utiliser des fonctions déjà codées dans la classe **DartV2** (objet **mybot**). La fonction **set_speed(self,speedLeft,speedRight)** commande la vitesse des moteurs gauches et droits. La valeur de commande doit être comprise entre -255 (vitesse maximum arrière) et 255 (vitesse maximum avant). Attention, le moteur demande un courant d'appel au démarrage et il faudra au minimum une commande de 70 pour faire tourner les moteurs sur le simulateur (sur le robot réel cette valeur est généralement comprise entre 60 et 80 et varie d'un robot à l'autre).

q.1 Modifier le programme "robot_cmd_1.py" en ajoutant une rotation du robot sur lui-même (sans translation) de durée 2 secondes avec une commande de vitesse de 100 en valeur absolue.

Si le programme est en boucle infinie, vous pouvez l'arrêter en tapant "controle C". Vous pouvez aussi en cas de problème arrêter et redémarrer le simulateur V-REP

3.3 Lecture de encodeurs (roues avant)

L'utilisation d'encodeurs sur les roues pour mesurer le déplacement d'un robot est une technique appelée odométrie et les encodeurs sont aussi appelés odomètres. La mesure des odomètres (encodeurs) avant se fait avec la fonction **get_front_encoders()** de la classe **DartV2** (objet **mybot**). Cette fonction retourne un tableau de deux valeurs : **[odoLeft,odoRight]**. Les encodeurs permettent de définir le nombre de tours effectués par les roues du robot DART en comptant un certain nombre d'impulsions (ticks) par tour de roue. Le robot Dart V2 produit, en principe, 300 ticks par tour de roue. La valeur renvoyée sur 16 bits est signée, elle varie de -32768 à +32767. Au départ, la valeur est 0. Attention, comme la mesure n'est que sur 16 bits, lorsque que l'on ajoute 1 tick à 32767 (valeur maximale), l'encodeur repasse à -32768 (valeur minimale).

q.2 Mesurer et afficher les valeurs des odomètres (avant gauche et droit) avant et après le mouvement angulaire du robot.

q.3 Ecrire dans la classe **DartV2** et tester avec le simulateur une nouvelle fonction (par exemple, delta_front_odometers()) calculant la différence odométrique entre deux mesures successives des encodeurs. Attention à bien prendre en compte les sauts dus au rebouclage de la mesure sur 16 bits.

q.4 Trouver empiriquement la durée de mouvement qui permet de réaliser un demi tour. Tester ce programme en faisant faire au robot un parcours aller-retour : avance pendant 5 secondes, demi-tour, avance pendant 5 secondes et demi-tour final.

q.5 En reprenant le programme précédent, afficher toutes les 500 ms les différences odométriques lors des deux lignes droites. Pour calculer la différence odométrique sans faire deux mesures, il peut être intéressant de mémoriser dans une variable de la classe **DartV2** les dernières mesures des odomètres.

3.4 Acquisition des odomètres arrières et de la tension batterie

La mesure des odomètres arrières est plus précise que celle des odomètres avants (vus au § 3.3) car elle mesure aussi le sens de rotation des roues. Un micro-contrôleur contrôle les capteurs et l'accès aux informations se fait par le bus I2C. Le micro-contrôleur donne accès aux encodeurs (odomètres) des roues arrières, au sens de rotation des moteurs et à la tension de la batterie. Le tableau suivant indique

La classe **i2c** possède trois fonctions d'accès au bus i2c :

— read(offs,nbytes) : lecture de nbytes octets à partir de l'adresse offs
— read_byte (offs) : lecture d'un octet à l'adresse offs
— write (offs, vData) : écriture des octets du tableau vData à partir de l'adresse offs

```
import smbus
import time
class i2c():
    def __init__(self,addr,bus_nb=2):
        self.__bus_nb = bus_nb
        self.__addr = addr
        self.__bus = smbus.SMBus(self.__bus_nb)
        print ("i2c device addr = 0x%x on bus %d"%(addr,bus_nb))

    def read(self,offs,nbytes):
        v = self.__bus.read_i2c_block_data(self.__addr,offs,nbytes)
        time.sleep(0.001)
        return v

    def read_byte(self,offs):
        v = self.__bus.read_byte_data(self.__addr,offs)
        time.sleep(0.001)
        return v

    def write(self,cmd,vData):
        self.__bus.write_i2c_block_data(self.__addr,cmd,vData)
        time.sleep(0.001)
```

adresse du registre	R/W	fonction
0x00	R	odomètre arrière gauche sur deux octets [poids faible, poids fort]
0x00	R	odomètre arrière gauche : octet de poids faible
0x01	R	odomètre arrière gauche : octet de poids fort
0x02	R	odomètre arrière droit sur deux octets [poids faible, poids fort]
0x02	R	odomètre arrière droit : octet de poids faible
0x03	R	odomètre arrière droit : octet de poids fort
0x04	R	sens de rotation roue gauche sur un octet
0x05	R	sens de rotation roue droite sur un octet
0x06	R	tension batterie sur deux octets [poids faible, poids fort]
0x06	R	tension batterie : octet de poids faible
0x07	R	tension batterie : octet de poids fort
0xC0	R	version du code sur un octet
0x01	W	écriture d'un octet de valeur 0 fait un reset de l'odomètre droit
0x02	W	écriture d'un octet de valeur 0 fait un reset de l'odomètre gauche
0x05	W	écriture d'un octet de valeur 0 fait un reset des deux odomètres

R et W indiquent respectivement que les valeurs sont lues ou écrites sur le bus I2C.

Utilisation de ces informations :

— Les odomètres donnent une valeur entre 0 et 65535. Attention, ces odomètres décomptent lorsque le robot avance. En marche avant, lorsque le compteur arrive à 0, il continue à 65535. En marche arrière, lorsque le compteur arrive à 65535, il continue à 0.

— Le sens de rotation est de 0 en arrière et de 1 en avant.

— La tension batterie est calculée à partir d'une tension v mesurée entre 0 et 5 V. Cette tension v est acquise par un convertisseur analogique numérique sur 10 bits et le bus I2C renvoie cette valeur numérique. La tension de la batterie est obtenue par un pont diviseur de tension : vbat = v * 430/100.

Pour accèder à ces capteurs, nous allons utiliser le canevas suivant (programme encoders.py dans le répertoire py/drivers) :

```
import time
import sys
import os

# rear wheels encoders and direction, battery level
# encoders reset

class EncodersIO ():
    def __init__(self, bus_nb = 2, addr = 0x14, sim=False, vsv=None):
        self.__sim = sim
        if self.__sim:
            self.vsv = vsv

        self.__bus_nb = 2
        self.__addr = 0x14

        # conditional i2c setup
        # if real robot , then we use actual i2c
        # if not , we are on simulated i2c
        sys.path.append('./drivers')
        if self.__sim:
            sys.path.append('../vDartV2')
            import i2csim as i2c
            self.__dev_i2c=i2c.i2c(self.__addr,self.__bus_nb,vsv=self.vsv)
        else:
            import i2creal as i2c
            self.__dev_i2c=i2c.i2c(self.__addr,self.__bus_nb)

        # place your new class variables here

    # place your encoder functions here
```

q.6 Dans le programme encoders.py, écrire et tester les fonctions permettant de lire les odomètres arrières et la tension de la batterie.

Un exemple d'utilisation de la classe **i2c** peut peut-être vous être utile. Cette fonction écrit un octet de valeur v=0 sur le bus I2C à l'adresse 2 :

```
def reset_rear_encoder_left (self):
    v = 0
    self.__write_byte(0x02,v)
```

Si il faut écrire plus d'un octet, la variable v doit être un tableau d'octets. Pour que le robot (objet **mybot**) utilise cette fonction il faut simplement écrire :

```
mybot.encoders.reset_rear_encoder_left()
```

q.7 Reprendre le programme de test précédent (aller-retour avec deux demi-tours) et effectuer les modifications suivantes :
- afficher la version du code en début de programme
- afficher la tension de la batterie en fin de programme
- utiliser les odomètres arrière à la place des odomètres avant

3.5 Acquisition des sonars

Les 4 sonars cardinaux (avant, arrière, droite et gauche) sont pilotés par les entrées-sorties (E/S) numériques d'une carte micro-contrôleur connectée à la carte PCDUINO 3 par le bus I2C. Les 2 sonars avant diagonaux (avant gauche et avant droit) sont directement connectés au bus I2C, chacun ayant une adresse différente.

Pour le contrôle des sonars, nous allons utiliser le canevas suivant (programme sonars.py dans le répertoire py/drivers) :

```
import time
import sys
import os
import platform

class SonarsIO ():
    def __init__(self,sim=False,vsv=None):
        self.__sim = sim
        if self.__sim:
            self.vsv = vsv
        #print ("vsv",vsv,self.vsv)
        self.__bus_nb = 2
        self.__addr_4_sonars = 0x21
        self.__addr_front_left = 0x070
        self.__addr_front_right = 0x072

        # conditional i2c setup
        # if real robot , then we use actual i2c
        # if not , we are on simulated i2c
        sys.path.append('./drivers')
        if self.__sim:
            sys.path.append('../vDartV2')
            sys.path.append('../../vDartV2') # to run test below in __main__
            import i2csim as i2c
            self.__dev_i2c_4_sonars=i2c.i2c(self.__addr_4_sonars,bus_nb=self.__bus_nb,vsv=
            self.__dev_i2c_front_left=i2c.i2c(self.__addr_front_left,bus_nb=self.__bus_nb,
            self.__dev_i2c_front_right=i2c.i2c(self.__addr_front_right,bus_nb=self.__bus_nb
        else:
            import i2creal as i2c
            self.__dev_i2c_4_sonars=i2c.i2c(self.__addr_4_sonars,self.__bus_nb)
            self.__dev_i2c_front_left=i2c.i2c(self.__addr_front_left,self.__bus_nb)
            self.__dev_i2c_front_right=i2c.i2c(self.__addr_front_right,self.__bus_nb)

        # place your new class variables here

    # place your sonar functions here
```

Les fonctions liées aux sonars sont à ajouter dans la classe **SonarsIO** (fichier sonars.py dans le répertoire py/drivers). Les circuits ("devices") I2C sont déjà créés dans la fonction **__init__(self)** :

- self.__dev_i2c_4_sonars : device pour les 4 sonars cardinaux à l'adresse 0x21 sur le bus I2C
- self.__dev_i2c_front_left : device pour sonar diagonal avant gauche à l'adresse 0x70 sur le bus I2C
- self.__dev_i2c_front_right : device pour sonar diagonal avant droit à l'adresse 0x72 sur le bus I2C

Pour acquérir les distances mesurées par les sonars cardinaux, vous pouvez utiliser le document de travail ("draft"), disponible sous Moodle. Pour plus d'information, vous pouvez aussi lire les explications qui suivent.

Les quatres sonars cardinaux sont numérotés [1] comme suit :

- sonar n°1 : sonar frontal
- sonar n°2 : sonar arrière
- sonar n°3 : sonar droit
- sonar n°4 : sonar gauche

1. en principe, à vérifier sur le simulateur et sur le robot

Les sonars cardinaux possèdent 4 modes de fonctionnement :
— mode 0 : le sonar est en attente ("standby" ou "idle"). Il n'effectue pas de mesures.
— mode 1 : le sonar est en mode "one shot", lorsqu'un sonar est programmé en mode 1, il démarre une seule mesure lorsque le mode passe à 1. Après avoir terminé la mesure, le sonar se remet en mode 0. Attention à bien attendre que le temps de trajet de l'onde ultrasonar entre l'obstacle et le robot soit écoulé avant de lire la mesure. La vitesse de propagation des ondes ultrasonores dans l'air est d'environ 340 $m\ s^{-1}$.
— mode 2 : mode synchrone ou les 4 sonars emettent en même temps et mesurent en permanence la distance aux obstacles. A priori, mais à vérifier sur le robot réel, il ne semble pas possible de mélanger le mode 2 avec les autres modes car les 4 sonars doivent être synchrones.
— mode 3 : mode assynchrone, le sonar mesure en permanence la distance aux obstacles.

Le sonar n'attend pas un temps infini avant de renvoyer sa mesure. Le temps maximum d'attente est défini par une distance maximum de mesure qui peut être modifiée.

La table suivante définit les principales commandes à envoyer sur le bus I2C pour contrôler les sonars cardinaux :

adresse du registre	R/W	fonction
0x00	R	mesure du sonar n°1 sur deux octets [poids faible, poids fort]
0x00	R	mesure du sonar n°1 : octet de poids faible
0x01	R	mesure du sonar n°1 : octet de poids fort
0x02	R	mesure du sonar n°2 sur deux octets [poids faible, poids fort]
0x02	R	mesure du sonar n°2 : octet de poids faible
0x03	R	mesure du sonar n°2 : octet de poids fort
0x04	R	mesure du sonar n°3 sur deux octets [poids faible, poids fort]
0x04	R	mesure du sonar n°3 : octet de poids faible
0x05	R	mesure du sonar n°3 : octet de poids fort
0x06	R	mesure du sonar n°4 sur deux octets [poids faible, poids fort]
0x06	R	mesure du sonar n°4 : octet de poids faible
0x07	R	mesure du sonar n°4 : octet de poids fort
0x08	R	changement de valeur indiquant une nouvelle mesure des sonars, bit i à 1 si changement pour sonar i+1 bit 0 : nouvelle mesure sur sonar n°1 bit 1 : nouvelle mesure sur sonar n°2 bit 2 : nouvelle mesure sur sonar n°3 bit 3 : nouvelle mesure sur sonar n°4
0xA0	R W	distance maximum pour la mesure du sonar n°1 en cm sur deux octets [poids faible, poids fort]
0xA0	R W	distance maximum sonar n°1 : octet de poids faible
0xA1	R W	distance maximum sonar n°1 : octet de poids fort
0xA2	R W	distance maximum pour la mesure du sonar n°2 en cm sur deux octets [poids faible, poids fort]
0xA2	R W	distance maximum sonar n°2 : octet de poids faible
0xA3	R W	distance maximum sonar n°2 : octet de poids fort
0xA4	R W	distance maximum pour la mesure du sonar n°3 en cm sur deux octets [poids faible, poids fort]
0xA4	R W	distance maximum sonar n°3 : octet de poids faible
0xA5	R W	distance maximum sonar n°3 : octet de poids fort
0xA6	R W	distance maximum pour la mesure du sonar n°4 en cm sur deux octets [poids faible, poids fort]
0xA6	R W	distance maximum sonar n°4 : octet de poids faible
0xA7	R W	distance maximum sonar n°4 : octet de poids fort
0xA8	R W	distance maximum pour les mesure synchrones (mode 2) en cm sur deux octets [poids faible, poids fort]
0xA8	R W	distance maximum mesures synchrones : octet de poids faible
0xA9	R W	distance maximum mesures synchrones : octet de poids fort
0xB0	R W	mode du sonar n°1 sur un octet
0xB1	R W	mode du sonar n°2 sur un octet
0xB2	R W	mode du sonar n°3 sur un octet
0xB3	R W	mode du sonar n°4sur un octet
0xC0	R	version du code sur un octet

R et W indiquent respectivement que les valeurs sont lues ou écrites sur le bus I2C.

q.8 En choissisant le ou les modes de fonctionnement qui vous intéressent, définir dans sonars.py les fonctions permettant d'accèder aux sonars cardinaux.

q.9 Completer le driver sonar.py en ajoutant les fonctions d'acquisition de distance aux obstacles

pour les deux sonars diagonaux. La fiche technique du capteur SRF02 disponible en fin du sujet (§ 5.3) indique comment piloter ces sonars. Programmer ce capteur pour qu'il renvoie une distance en centimetres. Attention à bien attendre le temps de trajet de l'onde ultrasonore entre l'émission et la réception.

4 Fonctions de déplacement du robot DART

Ces fonctions sont en principe à ajouter dans la classe **DartV2**. Vous pouvez aussi créer une autre classe faisant appel à **DartV2** pour avoir accès au robot.

Les fonctions de déplacement sont d'abord testées sur le robot virtuel puis sur le robot réel. Vous n'êtes pas obligés d'attendre d'avoir testé toutes les fonctions sur le simulateur pour commencer les tests sur le robot réel. Par exemple, lorsque le premier mouvement (rotation d'un angle donné) fonctionne sur le simulateur, vous pouvez directement le tester sur un robot réel disponible. Pour transférer et exécuter vos programmes sur le robot réel, reportez vous à la section 5.

4.1 Rotation d'un angle donné (méthode très simple)

Pour orienter le robot selon un cap prédéfini, nous allons utiliser les odomètres. Créer la fonction **setTurnSimple (cnt)** avec **cnt** la valeur de consigne, c'est à dire le nombre de tours que doit compter l'odomètre pour que le robot fasse une rotation sur lui-même d'un angle donné. Pour tourner d'un angle donné, vous pouvez déterminer **cnt** empiriquement avec le simulateur ou par calcul en utilisant le diamètre (12.5 cm) et l'écartement (26 cm env.) des roues et en condidérant 300 ticks par tour de roue.

Pour faire tourner le robot sur lui-même et, ainsi modifier son cap, il faut faire tourner les moteurs gauche et droit selon des sens opposés. On considère une fonction **set_speed(speed_left,speed_right)** pour commander les moteurs gauche et droits (en principe vous avez implémenté cette fonction en début de BE). Un exemple d'algorithme pour la fonction **setTurnSimple (cnt)** est présenté dans **Algorithm 1** :

Algorithm 1 Fonction - setTurnSimple (cnt)

```
cntOk ← False
odoMes ← Mesure des odomètres
odoEnd = odoMes + cnt Valeur de fin des odomètres
Définir un sens opposé de rotation pour les 2 moteurs (pour que le robot tourne sur lui-même dans le sens horaire)
Définir la vitesse des moteurs (exemple 120)
Appliquer vitesse et direction des moteurs au robot
while not cntOk do
    odoMes ← Mesure des odomètres
    odoErr ← odoEnd − odoMes
    if odoErr < 0 then
        cntOk ← True
        Mettre la vitesse des moteurs à zéro
    else
        Attendre un temps dt (typiquement entre 5 et 50 ms)
    end if
end while
```

Attention aux sens de comptage des odomètres (positif et négatif) et à la valeur renvoyée modulo 16 bits.
Nous observons que la fonction **setTurnSimple (head)** utilise l'erreur (**odoErr**) entre la mesure (**odoMes**) et la consigne (**odoEnd**) pour continuer ou interrompre la rotation du robot.

4.2 Régulation en ligne droite en utilisant les mesures issues des deux odomètres

Ecrire la fonction **goLineOdo (speed, duration)** qui permettra au robot d'exécuter une ligne droite en utilisant l'écart entre les variations des odomètres gauche et droit. Si cet écart est nul ou faible le robot avance en ligne droite. Cette fonction suppose que le robot est déja orienté au bon cap avec la fonction **setTurnSimple (cnt)** La variation d'un odomètre est calculée en mesurant successivment les valeurs de l'odomètre avec un intervalle de temps de ΔT.

4.3 Régulation en ligne droite en suivant un mur à droite ou à gauche

Ecrire la fonction **wallFollow (speed, duration, distance)** qui permettra au robot d'exécuter une ligne droite en utilisant l'écart par rapport au mur (à gauche ou à droite) mesuré par le sonar.

5 Passage au robot DART réel

Dans cette partie, nous allons tester sur le robot réel, les programmes et fonctions écrites et testées sur le robot virtuel. Avec la nouvelle version des microcodes de odomètres arrières et des sonars cardinaux, il faut s'attendre à quelques surprises ! ? !

5.1 Important - Programmes de tests

Les microcontrolleurs des robots DARTV2 sont dans une nouvelle version logicielle. Ils n'ont pas été testés systèmatiquement. Il est donc important, qu'avant de travailler sur un robot que vous ne connaissez, vous testiez les fonctions principales du robot avant de mettre en doute vos programmes. Ecrivez de petits programmes de tests permettant :

— la commande des moteurs
— la lecture des encodeurs avants
— la lecture des encodeurs arrières
— la mesure des distances avec les sonars

Il est aussi recommandé :

— d'afficher la tension de la batterie à la fin de chaque programme
— d'écrire un petit programme stop.py permettant de couper les moteurs

5.2 Prise en main du robot DART

Chaque robot DART est repéré par un numéro xx compris entre 01 et 18. Le numéro xx sert à définir l'adresse IP du robot sur le réseau de l'école :

```
172.20.25.1xx
```

Le robot DART possède deux interrupteurs de mise en marche :

— un petit interrupteur à l'arrière gauche pour la carte principale PCDuino3
— un gros interrupteur rouge avec voyant au dessus du robot pour la partie puissance

Si votre robot part en "live" vous pouvez couper la partie puissance en appuyant sur l'interrupteur rouge. Il est conseillé d'avoir un petit programme qui juste l'arrêt des moteurs appellé stop.py à exécuter en cas d'urgence.

La connexion au robot réel se fait à travers un terminal à l'aide de la commande ssh. Cette commande permet "de déplacer" votre terminal "à l'intérieur" du robot en utilisant son adresse IP. La communication se fait en WI-FI. Sur votre PC de salle info, la commande à exécuter est :

```
ssh uv27@172.20.25.1xx
```

le mot de passe est uv27 et xx correspond au numéro du robot de 01 à 18. Si tout se passe bien, vous êtes maintenant "dans le robot" dans le répertoire /home/uv27. La commande pwd doit indiquer /home/uv27 Afin de ne pas interférer avec les autres utilisateurs, créer votre propre répertoire :

```
mkdir grpXXXX
```

avec XXXX le pseudo personnalisé de votre groupe. Créer un second terminal et placez vous dans le répertoire ou se trouve votre script python de commande du robot DART (dartv2). Pour copier vos fichiers sur le robot, placez vous dans le dossier **dartv2** sur votre PC et exécutez la commande suivante :

```
scp -rp py uv27@172.20.25.1xx:grpXXXX
```

avec xx le numéro du robot et XXXX votre pseudo. Le mot de passe est uv27. La totalité du contenu de **py** sera recopiée (par WI_FI) dans le dossier /**home**/**uv27**/**grpXXXX** de votre robot Dart.

Repasser dans le premier terminal (i.e. sur le robot) et exécuter votre programme (ex robot) en tapant :

```
cd grpXXXX/py
python3 robot_cmd_mystuff.py
```

Prévoir quelqu'un à coté du robot pour coupure de l'alimentation des moteurs (bouton rouge) en cas de perte de contrôle ! ! !

L'idéal est que chaque binôme (ou trinôme) conserve le même robot pendant tout le cours. Mais les aléas électroniques ou mécaniques peuvent rendre le robot inutilisable le temps de sa réparation. Il faudra exécuter la procédure précédente sur en autre robot en gardant bien votre pseudo de groupe grpXXXX.

Si vous modifiez le code python directement sur le robot, gardez toujours sur votre ordinateur une copie du code développé sur le robot, car en cas de gros problèmes, le robot est remis à zéro dans sa version de base.

5.3 Annexes

SRF02 Ultrasonic range finder

Technical Specification

I2C Mode

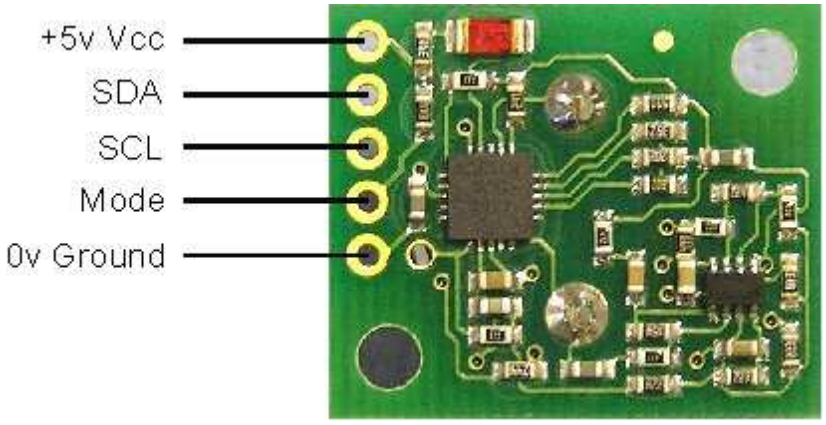
For Serial mode click here

I2C Communication

To use the SRF02 in I2C mode, make sure nothing is connected to the mode pin, it must be left unconnected.
The I2C bus is available on popular controllers such as the OOPic, Stamp BS2p, PicAxe etc. as well as a wide variety of micro-controllers. To the programmer the SRF02 behaves in the same way as the ubiquitous 24xx series EEPROM's, except that the I2C address is different. The default shipped address of the SRF02 is 0xE0. It can be changed by the user to any of 16 addresses E0, E2, E4, E6, E8, EA, EC, EE, F0, F2, F4, F6, F8, FA, FC or FE, therefore up to 16 sonar's can be used.

Connections

The connections to the SRF02 are identical to the SRF08 and SRF10 rangers. The "Mode" pin should be left unconnected, it has an internal pull-up resistor. The SCL and SDA lines should each have a pull-up resistor to +5v somewhere on the I2C bus. You only need one pair of resistors, not a pair for every module. They are normally located with the bus master rather than the slaves. The SRF02 is always a slave - never a bus master. If you need them, I recommend 1.8k resistors. Some modules such as the OOPic already have pull-up resistors and you do not need to add any more.



Registers

The SRF02 appears as a set of 6 registers.

Location	Read	Write
0	Software Revision	Command Register

1	Unused (reads 0x80)	N/A
2	Range High Byte	N/A
3	Range Low Byte	N/A
4	Autotune Minimum - High Byte	N/A
5	Autotune Minimum - Low Byte	N/A

Only location 0 can be written to. Location 0 is the command register and is used to start a ranging session. It cannot be read. Reading from location 0 returns the SRF02 software revision. The ranging lasts up to 65mS, and the SRF02 will not respond to commands on the I2C bus whilst it is ranging.

Locations, 2 and 3, are the 16bit unsigned result from the latest ranging - high byte first. The meaning of this value depends on the command used, and is either the range in inches, or the range in cm or the flight time in uS. A value of 0 indicates that no objects were detected. Do not initiate a ranging faster than every 65mS to give the previous burst time to fade away.

Locations, 4 and 5, are the 16bit unsigned minimum range. This is the approximate closest range the sonar can measure to. See the Autotune section below for full details.

Commands

The are three commands to initiate a ranging (80 to 82), to return the result in inches, centimeters or microseconds. Another set of three commands (86 to 88) do the same, but without transmitting the burst. These are used where the burst has been transmitted by another sonar. It is up to you to synchronize the commands to the two sonar's. There is a command (92) to transmit a burst without doing the ranging and also a set of commands to change the I2C address.

Command		Action
Decimal	Hex	
80	0x50	Real Ranging Mode - Result in inches
81	0x51	Real Ranging Mode - Result in centimeters
82	0x52	Real Ranging Mode - Result in micro-seconds
86	0x56	Fake Ranging Mode - Result in inches
87	0x57	Fake Ranging Mode - Result in centimeters
88	0x58	Fake Ranging Mode - Result in micro-seconds
92	0x5C	Transmit an 8 cycle 40khz burst - no ranging takes place
96	0x60	Force Autotune Restart - same as power-up. You can ignore this command.

160	0xA0	1st in sequence to change I2C address
165	0xA5	3rd in sequence to change I2C address
170	0xAA	2nd in sequence to change I2C address

Ranging
To initiate a ranging, write one of the above commands to the command register and wait the required amount of time for completion and read the result. The echo buffer is cleared at the start of each ranging. The ranging lasts up to 66mS, after this the range can be read from locations 2 and 3.

Checking for Completion of Ranging
You do not have to use a timer on your own controller to wait for ranging to finish. You can take advantage of the fact that the SRF02 will not respond to any I2C activity whilst ranging. Therefore, if you try to read from the SRF02 (we use the software revision number a location 0) then you will get 255 (0xFF) whilst ranging. This is because the I2C data line (SDA) is pulled high if nothing is driving it. As soon as the ranging is complete the SRF02 will again respond to the I2C bus, so just keep reading the register until its not 255 (0xFF) anymore. You can then read the sonar data. Your controller can take advantage of this to perform other tasks while the SRF02 is ranging. The SRF02 will always be ready 70mS after initiating the ranging.

LED
The red LED is used to flash out a code for the I2C address on power-up (see below). It also gives a brief flash during the "ping" whilst ranging.

Changing the I2C Bus Address
To change the I2C address of the SRF02 you must have only one sonar on the bus. Write the 3 sequence commands in the correct order followed by the address. Example; to change the address of a sonar currently at 0xE0 (the default shipped address) to 0xF2, write the following to address 0xE0; (0xA0, 0xAA, 0xA5, 0xF2). These commands must be sent in the correct sequence to change the I2C address, additionally, No other command may be issued in the middle of the sequence. The sequence must be sent to the command register at location 0, which means 4 separate write transactions on the I2C bus. When done, you should label the sonar with its address, however if you do forget, just power it up without sending any commands. The SRF02 will flash its address out on the LED. One long flash followed by a number of shorter flashes indicating its address. The flashing is terminated immediately on sending a command the SRF02.

Address		Long Flash	Short flashes
Decimal	Hex		
224	E0	1	0
226	E2	1	1
228	E4	1	2
230	E6	1	3
232	E8	1	4
234	EA	1	5
236	EC	1	6

238	EE	1	7
240	F0	1	8
242	F2	1	9
244	F4	1	10
246	F6	1	11
248	F8	1	12
250	FA	1	13
252	FC	1	14
254	FE	1	15

Take care not to set more than one sonar to the same address, there will be a bus collision and very unpredictable results.

Note - there is only one module address stored in the SRF02. If you change it, the equivalent Serial Mode address will also change:

```
0xE0, 0xE2, 0xE4, 0xE6, 0xE8, 0xEA, 0xEC, 0xEE, 0xF0, 0xF2, 0xF4, 0xF6, 0xF8, 0xFA, 0xFC, 0xFE
I2C addresses
0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F
Equivalent Serial addresses
```

AutoTune

The SRF02 does not require any user calibration. You power up and go right ahead and use the SRF02.
Internally, there are tuning cycles happening automatically in the background. After the ultrasonic burst has been transmitted, the transducer keeps on ringing for a period of time. It is this ringing which limits the closest range the SRF02 can measure. This time period varies with temperature and from transducer to transducer, but is normally the equivalent of 11 to 16cm (4" to 6"), a bit more if the transducer is warm. The SRF02 is able to detect the transducer ring time and move its detection threshold right up to it, giving the SRF02 the very best performance possible. On power up, the detection threshold is set to 28cm (11"). The tuning algorithms quickly back this right up to the transducer ring. This happens within 5-6 ranging cycles - less than half a second at full scan speed. After this the tuning algorithms continue to monitor the transducer, backing the threshold up even further when possible or easing it out a bit when necessary. The tuning algorithms work automatically, in the background and with no impact on scan time.
The minimum range can be checked, if required by reading registers 4 and 5. This value is returned in uS, cm or inches, the same as the range. It is also possible to make the SRF02 re-tune by writing command 96 but you can ignore this command. It is used during our testing.