



Sommaire

I.	Introduction.....	2
II.	Réalisation du projet	3
A.	Rappel de la tâche de l'étudiant.....	3
B.	Communication i2c entre Arduinos et Raspberry	4
C.	Structure des messages.....	5
III.	Réalisation du programme Mécanisme 4 : le Feu	6
A.	Schéma câblage	6
B.	Programme.....	6
IV.	Réalisation du programme Mécanisme 8 : le Riz	9
A.	Montage	9
B.	Programme.....	9
V.	Réalisation du programme i2c	12
A.	Script Python sur Raspberry	12
B.	Programme Arduino	17
VI.	Test unitaire.....	18
VII.	Fiches recettes.....	19
A.	Observer le bon fonctionnement du mécanisme 4 : le Feu	19
B.	Observer le bon fonctionnement du mécanisme 8 : le Riz	20
C.	Observer l'envoi des informations mécanismes en BDD	21
D.	Observer l'envoi des ordres sur les Arduinos.....	22
	Conclusion	23
A.	Communication de groupe.....	23
B.	Regard critique du projet	23
C.	Connaissances apportées.....	23
D.	Poursuite d'étude.....	24
	Annexe.....	25



I. Introduction

Rappel du cahier des charges

Le client souhaite que le système technique actuellement en place soit recréé entièrement. La finalité du projet est la suivante :

- Chaque mécanisme du système doit pouvoir fonctionner de manière indépendante sur une carte Arduino nano.
- Une Raspberry doit pouvoir :
 - Récupérer les informations mécanismes via un bus i2c et les envoyer en base de données.
Les informations mécanismes sont les suivantes :
 - 🚦 Pour l'état des mécanismes : A chaque changement et toutes les 60s
 - 🚦 Pour l'état des actionneurs : A chaque changement et toutes les 60s
 - 🚦 Pour l'état/la valeur des capteurs : A chaque changement et toutes les 5s max
 - Récupérer des messages d'ordre via un serveur socket et les envoyer aux Arduinos via un bus i2c.
- Une base de données doit pouvoir stocker les informations mécanismes
- Une application Web doit pouvoir visualiser les informations mécanismes en temps réel.
- Une application Web doit pouvoir activer/désactiver l'état de chaque actionneur et chaque mécanisme.



II. Réalisation du projet

A. Rappel de la tâche de l'étudiant

Aperçu des tâches réalisées

Dans le système Escape Game, ma partie de développement consiste à créer plusieurs programmes :

- Deux programmes pour deux mécanismes sur Arduino :
 - Le mécanisme 4 : le Feu, avec un interrupteur à clef (Langage Arduino ~C++)
 - Le Mécanisme 8 : le Riz, avec un capteur de poids (Langage Arduino ~C++)
- Deux programmes pour une communication Arduino-Raspberry via i2c afin de :
 - Recevoir les informations mécanismes sur la Raspberry (Langage Python et Arduino)
 - Envoyer les ordres aux Arduinos et les exécuter (Langage Python et Arduino)

Pour la partie i2c le langage python est utilisé avec différentes librairies, comme smbus pour la communication i2c.

De même le langage Arduino est utilisé avec Wire pour l'i2c et avec HX711 pour le capteur de poids.

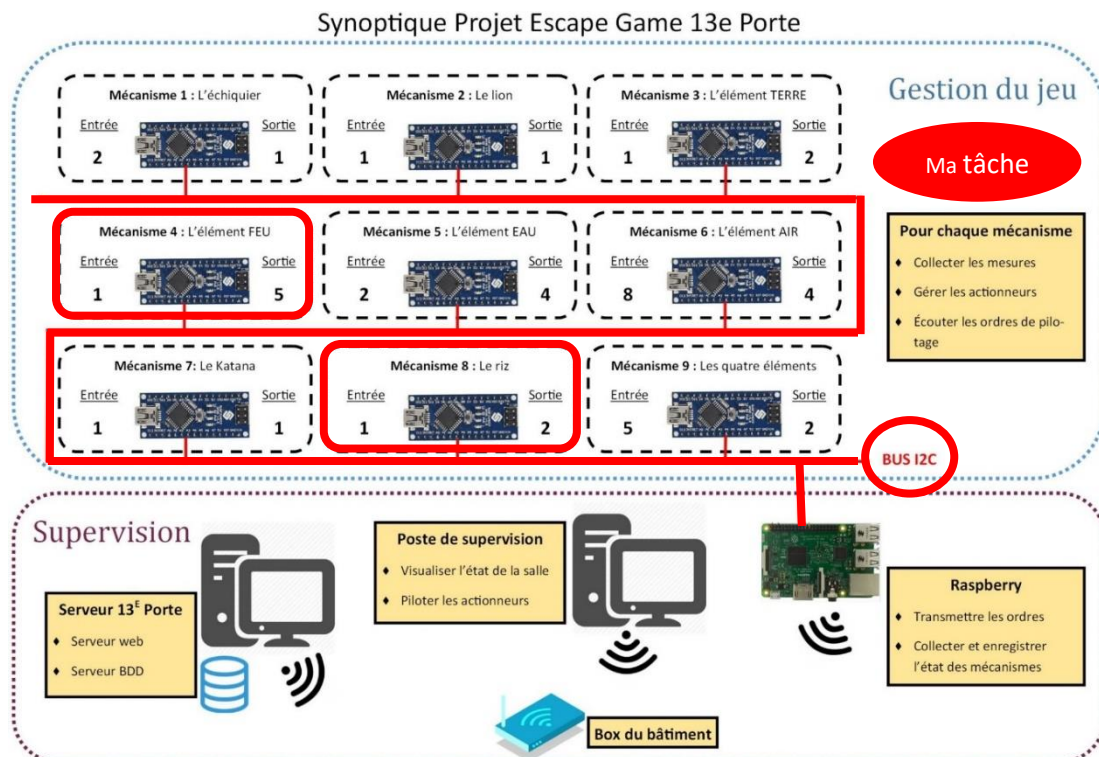


Image 1 : Ma tâche personnelle

Contraintes liées au développement

A cause du confinement le programme concernant le mécanisme 4 et le mécanisme 8 n'as pas pu être testé par manque de relais et de capteur de poids.



B. Communication i2c entre Arduinos et Raspberry

Synoptique

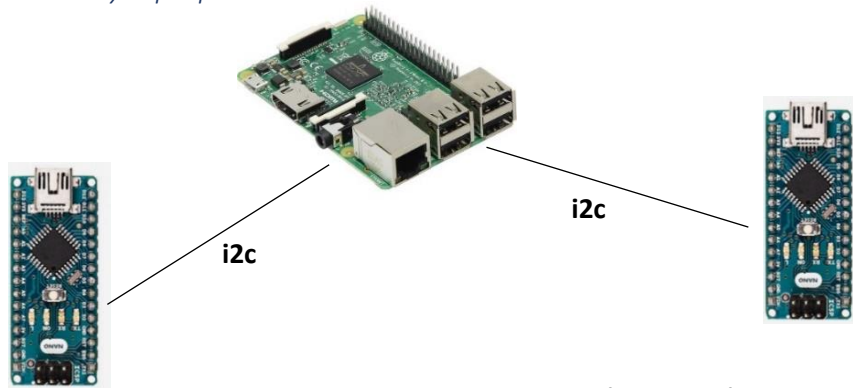


Image 2 : Communication Raspberry - Arduinos

Schéma Câblage

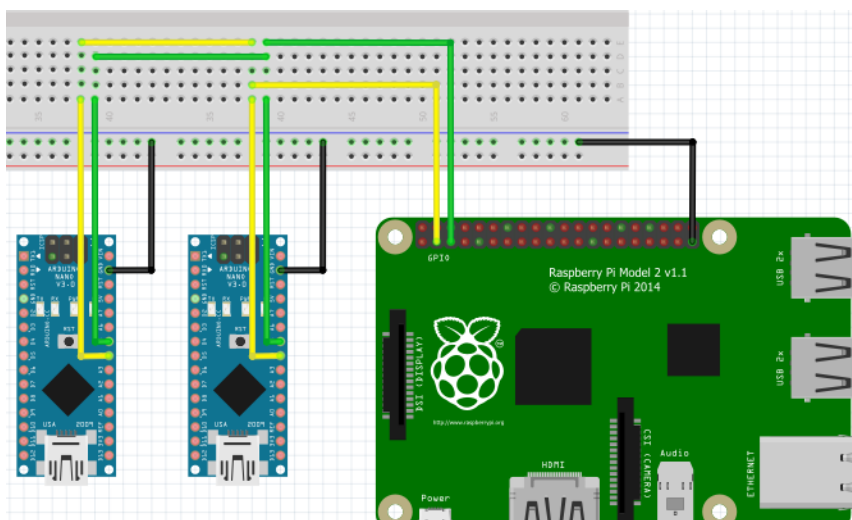


Image 3 : Schéma câblage avec 1 Raspberry et 2 Arduinos

Communication

Avec l'i2c il y a un principe de maître et esclave. Ici nous utiliserons le Raspberry Pi en tant que maître et les Arduino en tant qu'esclave.

Le maître (la Raspberry) est le composant qui initialise un transfert, génère le signal d'horloge et termine le transfert. Dans notre cas il sera récepteur et émetteur.

Les esclaves (Les Arduinos) sont les composants adressés par un maître. Dans notre cas ils seront récepteurs et émetteurs.



C. Structure des messages

Message i2c des informations mécanismes

Afin d'envoyer les informations mécanismes à la Raspberry on construit un message bien défini :

ms[F ou T]as[F ou T][F ou T ou X][F ou T ou X][F ou T ou X]sd[F ou T ou {Valeur Numérique}X]...

3eme caractère 6eme – 9eme caractère 12eme caractère et +

Le 3^{ème} caractère : correspond à l'état du mécanisme :

- Si le mécanisme est validé on envoie **T**
- Si le mécanisme n'est pas validé on envoie **F**

Le 6^{ème} - 9^{ème} caractère : correspond à l'état des actionneurs du mécanisme :

- Si l'état de l'actionneur est validé on envoie **T**
- Si l'état de l'actionneur n'est pas validé on envoie **F**
- S'il n'y a plus d'actionneur on envoie **X**

Le 12^{ème} caractère et + : correspond à la valeur / état des capteurs du mécanisme

- Si le capteur possède un état booléen :
 - Si l'état du capteur est validé on envoie **T**
 - Si l'état du capteur n'est pas validé on envoie **F**
- Si le capteur possède une valeur numérique on envoie la **valeur numérique + X**

Message Socket/i2c d'ordre

Afin d'envoyer un ordre à une Arduino on construit un message avec une structure bien définie :

[1- 9] [0 ou 1 ou 2] [0 ou 1 ou 2]...

1er caractère 2eme caractère 3eme caractère et +

Le 1^{er} caractère : correspond au numéro du mécanisme

Le 2^{ème} caractère : correspond à l'état du mécanisme :

- Si l'état de l'actionneur doit être validé il y a un **1**
- Si l'état de l'actionneur doit être invalidé il y a un **0**
- Si on ne touche pas à l'état du mécanisme il y a un **2**

Le 3^{ème} caractère et + : correspond à l'état des actionneurs du mécanisme :

- Si l'état de l'actionneur doit être validé il y a un **1**
- Si l'état de l'actionneur doit être invalidé il y a un **0**
- Si on ne touche pas à l'actionneur il y a un **2**



III. Réalisation du programme Mécanisme 4 : le Feu

A. Schéma câblage

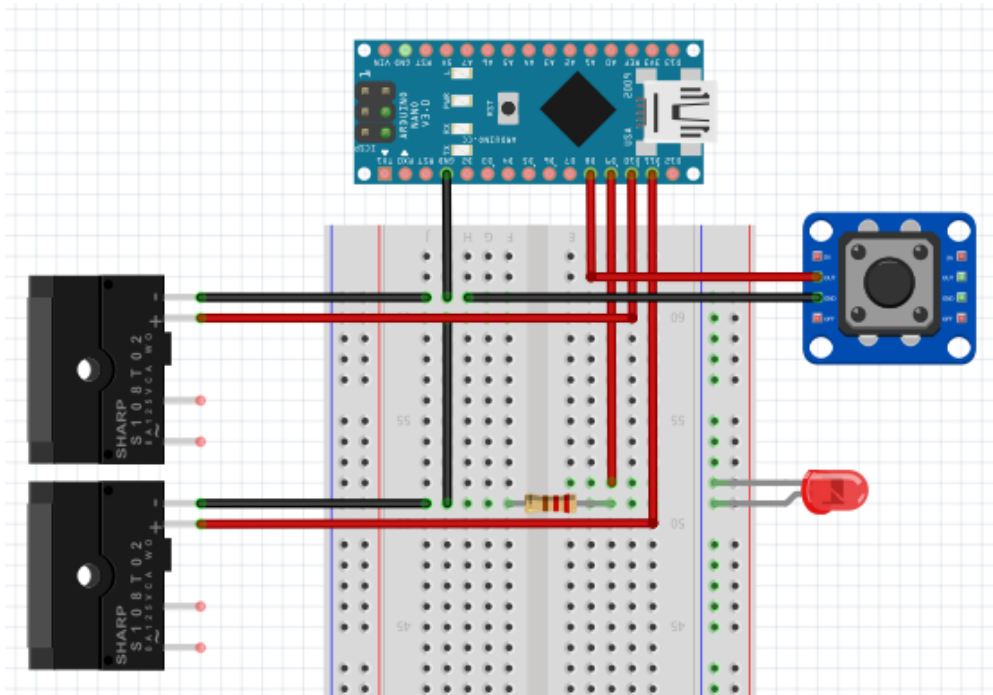


Image 4 : Schéma câblage du mécanisme 4

B. Programme

Initialisation

On définit des variables pour les pins utilisées de l'Arduino :

```
#define CInterupteur_PIN 8    //interrupteur a clef sur pin 8
#define SLed_PIN 9           //led controle sur pin 9
#define SDragon_PIN 10       //relais de l'électroaimant de la ventouse dragon sur pin10
#define SFumee_PIN 11        //relais de la machine à fumee sur pin 11
```

Image 5 : Code en tête de feu.ino

On initialise le matériel utilisé pour le mécanisme :

```
pinMode(CInterupteur_PIN , INPUT_PULLUP);    //On initialise le pin de l'interrupteur à clef
pinMode(SLed_PIN, OUTPUT);                    //On initialise le pin de la led
digitalWrite(SLed_PIN , LOW);                 //On éteint la led par défaut

pinMode(SDragon_PIN, OUTPUT);                 //On initialise le pin du relais de l'électroaimant de la ventouse dragon
digitalWrite(SDragon_PIN, LOW);               //On active le relais de l'électroaimant par défaut

pinMode(SFumee_PIN, OUTPUT);                 //On initialise le pin du relais de la machine à fumée
digitalWrite(SFumee_PIN, HIGH);               //On désactive le relais machine à fumée par défaut
```

Image 6 : Code fonction setupMechanism() dans feu.ino



Le main

Le programme exécute principalement les instructions suivantes :

```

Feu mechanism = Feu(); //On instancie un objet de type Feu

void setup() {
    mechanism.setupMechanism(); //On donne une configuration de base au mécanisme
}

void loop() {
    delay(100); //On attends 0.1 seconde
    mechanism.execute(); //On exécute le mécanisme
}

```

Image 7 : Code main dans feu.ino

La classe Feu

Principaux attributs

Les principaux attributs de la classe sont les suivants :

```

bool S_Dragon; //actionneur qui active/désactive l'électroaimant de la ventouse dragon
bool S_Fumee; //actionneur qui active/désactive la fumée
bool S_Led; //actionneur qui active/désactive la led de controle
bool S_Feu; //actionneur de l'élément FEU sur la tablette des 4 éléments
bool C_Interupteur; //état de la position détecter par l'interrupteur à clef
bool mechanism_status; //indique si le mécanisme est activé ou non

```

Image 8 : Code classe feu dans feu.ino

Principales méthodes

Les principales méthodes sont les suivantes :

```

public :
    Feu(); //constructeur de la classe
    void setupMechanism(); //configuration de base du mécanisme
    void execute(); //méthode qui fait fonctionner le mécanisme

```

Image 9 : Code classe feu dans feu.ino

La méthode execute

Synopsis

La méthode « **execute** » est la méthode qui fait fonctionner le mécanisme. Elle est appelée dans la fonction loop du programme et donc exécutée en boucle.

Récupérer l'état de l'interrupteur à clef

Pour récupérer l'état de l'interrupteur à clef on utilise l'instruction suivante :

```

sd_reading = digitalRead(CInterupteur_PIN); //On récupère la valeur du capteur interrupteur à clef

```

Image 10 : Code fonction execute() dans feu.ino



Valider le mécanisme

Pour valider le mécanisme on exécute les instructions suivantes :

```
if (mechanism_status == false) {                                //Si il y a eu le premier changement de position
    S_Dragon = true;                                           //On active l'électroaimant de la ventouse dragon
    S_Fumee = true;                                           //On active la fumée
    S_Led = true;                                             //On allume la led verte
    S_Feu = true;                                             //On allume l'element FEU des 4 éléments
    mechanism_status = true;                                  //On valide le mécanisme
}
```

Image 11 : Code fonction execute() dans feu.ino



IV. Réalisation du programme Mécanisme 8 : le Riz

A. Montage

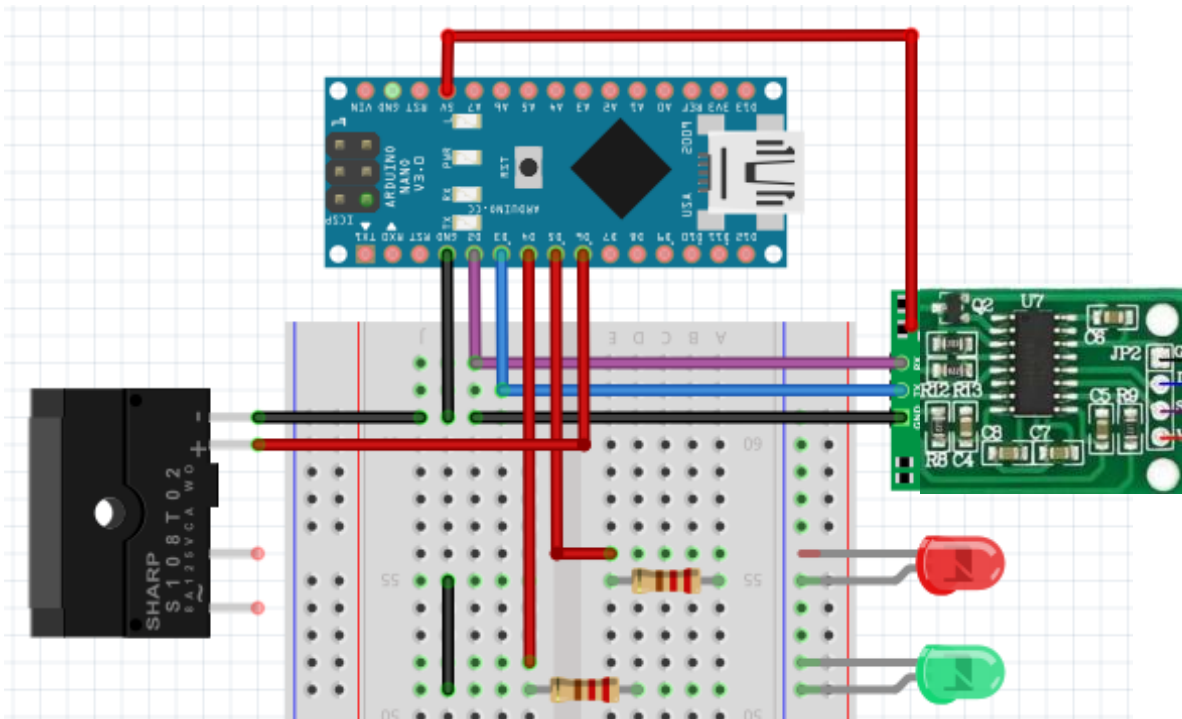


Image 12 : Schéma montage du mécanisme 8

B. Programme

Initialisation

Afin d'utiliser le capteur de poids il a fallu utiliser la librairie HX711 :

```
#include "HX711.h" //librairie nécessaire au capteur de poids
```

Image 13 : Code en tête de riz.ino

On définit des variables pour les pins utilisées de l'Arduino :

```
#define CPoids_CLK 2 //clk du capteur de poids sur pin 2
#define CPoids_DOUT 3 //dout du capteur de poids sur pin 3
#define SLedV_PIN 4 //led verte sur pin 4
#define SLedR_PIN 5 //led rouge sur pin 5
#define STableau_PIN 6 //relais de l'électroaimant de la chute de tableau sur pin 6
```

Image 14 : Code en tête de riz.ino



On initialise le matériel utilisé pour le mécanisme :

```
HX711 scale(CPoids_DOUT, CPoids_CLK); //On initialise les pins du capteur de poids
scale.set_scale(calibration_factor); //On ajuste le capteur de poids au facteur d'étalonnage

pinMode(SLedV_PIN, OUTPUT); //On initialise le pin de la led verte
digitalWrite(SLedV_PIN, LOW); //On éteint la led verte par défaut

pinMode(SLedR_PIN, OUTPUT); //On initialise le pin de la led rouge
digitalWrite(SLedR_PIN, HIGH); //On éteint la led rouge par défaut

pinMode(STableau_PIN, OUTPUT); //On initialise le pin du relais de l'électroaimant de la chute de tableau
digitalWrite(STableau_PIN, HIGH); //On désactive le relais de l'électroaimant par défaut
```

Image 15 : Code fonction setupMechanism() dans riz.ino

Le main

Le programme exécute principalement les instructions suivantes :

```
Riz mechanism = Riz(); //On instancie un objet de type Riz

void setup() {
    mechanism.setupMechanism(); //On donne une configuration de base au mécanisme
}

void loop() {
    delay(100); //On attends 0.1 seconde
    mechanism.execute(); //On exécute le mécanisme
}
```

Image 16 : Code fonction main dans riz.ino

La classe Riz

Principaux attributs

Les principaux attributs de la classe sont les suivants :

```
bool S_Tableau; //actionneur qui active/désactive l'électroaimant de la chute de tableau
bool S_Led; //actionneur qui active/désactive la led de controle
int C_Poids; //valeur mesuré par le capteur de poids
bool mechanism_status; //indique si le mécanisme est activé ou non
```

Image 17 : Code classe riz dans riz.ino

Principales méthodes

Les principales méthodes sont les suivantes :

```
Riz(); //constructeur de la classe
void setupMechanism(); //configuration de base du mécanisme
```

Image 18 : Code classe riz dans riz.ino



La méthode execute

Synopsis

La méthode « **execute** » est la méthode qui fait fonctionner le mécanisme. Elle est appelée dans la fonction loop du programme et donc exécutée en boucle.

Récupérer la valeur du capteur de poids

Pour récupérer la valeur mesurée par le capteur de poids on utilise l'instruction suivante :

```
sd_reading = (scale.get_units() / 2.0); //On récupère une lere mesure du capteur de poids
```

Image 19 : Code fonction execute() dans riz.ino

Valider le mécanisme

Pour valider le mécanisme on exécute les instructions suivantes :

```
if ( (sd_reading >= 0.48)
&& (sd_reading <= 0.52 )
&& mechanism_status == false ) { //Si la mesure est comprise entre 0.48 et 0.52 pour la lere fois
    S_Led = true; //On allume la led verte et on éteint la rouge
    S_Tableau = true; //On libère la chute du tableau
    mechanism_status = true; //On valide le mécanisme
}
```

Image 20 : Code fonction execute() dans riz.ino



V. Réalisation du programme i2c

A. Script Python sur Raspberry

Synoptique

Ce script python consiste à :

- Recevoir la valeur des capteurs de chaque mécanisme à chaque changement ou sinon toutes les 5s max. Et ensuite envoyer les valeurs à la BDD.
- Recevoir l'état des actionneurs de chaque mécanisme à chaque changement ou sinon toutes les 60s. Et ensuite envoyer les valeurs à la BDD.
- Recevoir l'état de chaque mécanisme à chaque changement ou sinon toutes les 60s. Et ensuite envoyer les valeurs à la BDD.
- Envoyer les ordres reçus depuis l'application Web au mécanisme correspondant

Programme

La Raspberry est maître dans la communication i2c. Et la communication i2c est établie grâce à la bibliothèque **smbus** :

```
import smbus                                     #bibliothèque nécessaire pour la communication i2c
```

Image 21 : Code en tête de i2c.py

Les dictionnaires

Pour stocker les informations relatives aux Arduinos dans le programme il a été judicieux de créer une liste de 9 dictionnaires correspondant aux 9 Arduinos. Les clés de ces dictionnaires sont identiques.

Chaque dictionnaire Arduino possède les clés suivantes :

- « **id** » : correspond au numéro du mécanisme assigné à l'Arduino
- « **address** » : correspond à l'adresse esclave assigné à l'Arduino pour l'i2c
- « **mechanism_status** » : correspond à l'état du mécanisme (valeur True/False)
- « **ms_noTimer** » : correspond à une valeur entière
Nécessaire à l'exécution du thread MTimer pour l'envoi des valeurs actionneurs toutes les 60s
- « **actuator_status** » : correspond à l'état des actionneurs du mécanisme
C'est une liste des différents actionneurs (ayant chacun une valeur True/False).
- « **as_noTimer** » : correspond à une valeur entière
Nécessaire à l'exécution du thread ATimer pour l'envoi des valeurs actionneurs toutes les 60s
- « **sensor_data** » : correspond à l'état des capteurs du mécanisme
C'est une liste des différents capteur (ayant chacun une valeur entière ou True/False).
- « **sd_noTimer** » : correspond à une valeur entière
Nécessaire à l'exécution du thread SDTimer pour l'envoi des valeurs capteurs toute les 4s.
- « **order** » : correspond à une valeur entière, à l'ordre envoyé par le PC de supervision.



Voici un fragment de la liste de dictionnaires :

```
arduinios = [
    {'id': 1, 'address': 0x12,
     'mechanism_status': False, 'ms_noTimer': 0,
     'actuator_status' : {'S_Echiquier': False}, 'as_noTimer' : 0,
     'sensor_data' : {'C_EffetHall1': 0, 'C_EffetHall2': 0}, 'sd_noTimer': 0,
     'order' : 0},

    {'id': 2, 'address': 0x13,
     'mechanism_status': False, 'ms_noTimer': 0,
     'actuator_status' : {'S_Lion': False}, 'as_noTimer' : 0,
     'sensor_data' : {'C_EffetHall': 0}, 'sd_noTimer': 0,
     'order' : 0},
```

Image 22 : Code définition dictionnaire arduinos[] dans i2c.py

Ce sont ces dictionnaires qui sont modifiés à chaque changement dans les mécanismes.

Communication Arduino

Afin de recevoir les informations des mécanismes en temps réel il a fallu utiliser des threads. De sorte que toutes les Arduinos communiquent avec la Raspberry en même temps.

Un thread est une séquence d'instructions qui s'exécute parallèlement aux autres.

Pour cela il a d'abord fallu créer le thread **ArduinoCom** qui correspond à la communication de la Raspberry avec une Arduino ; ensuite il a fallu assigner à chaque Arduino un thread ArduinoCom.

Le thread ArduinoCom consiste à effectuer en boucle les instructions suivantes :

```
while self.running:
    try :
        print("Arduino %s : communication" %self.arduino['id'])

        send_order(self.arduino)           #On envoie le message d'ordre a l'Arduino s'il y en a un
        message = get_message(self.arduino) #On recupere le message i2c venant de l'Arduino
        get_mechanism_status(message, self.arduino) #On verifie l'etat du mecanisme
        get_actuator_status(message, self.arduino) #On verifie l'etat des actionneurs
        get_sensor_data(message, self.arduino) #On verifie la valeur des capteurs

        time.sleep(2)                       #On attend 2 secondes

    except :                                #Si il y a une erreur quelconque dans l'execution du thread
        self.running = False                #On arrete le thread
        thread = ArduinoCom(self.arduino)   #On cree un nouveau thread
        thread.start()                       #On lance le nouveau thread
```

Image 23 : Code thread ArduinoCom dans i2c.py



Les principales instructions du thread ArduinoCom sont les suivantes :

- Envoyer un ordre à une Arduino

On appelle la fonction **send_order()**.

Elle exécute principalement les instructions suivantes :

```
if arduino['order'] != 0 :                                #On verifie si l'Arduino a reçu un ordre

    order = convertStrToListHex(str(arduino['order']))    #On convertit le message socket en Hexadecimal
    bus.write_i2c_block_data(arduino['address'], 0, order) #On envoie le message socket a l'Arduino assignee

    print("Mechanism %s : ORDER SENT : %s" %(arduino['id'],arduino['order']))
    arduino['order'] = 0                                  #On reset l'ordre
```

Image 24 : Code fonction send_order() dans i2c.py

- Récupérer le message i2c des informations mécanismes

On appelle la fonction **get_message()**.

Elle exécute les instructions suivantes :

```
answer = bus.read_i2c_block_data(arduino['address'],0x32) #On recupere le message i2c

l = []
for letter in answer:
    if letter == 255:
        break
    l.append(chr(letter))

message="".join(l)                                         #On converti le message i2c en une chaine de caractere

return message #msFasFFFFsdF/msFasFFXXsd0X
```

Image 25 : Code fonction get_message() dans i2c.py

- Gérer l'état des mécanismes

On appelle la fonction **get_mechanism_status()**.

Elle exécute principalement les instructions suivantes :

```
cpt = 2
if message[cpt] == "T" :                                  #La partie capteur du message commence au 3eme caractere
    if arduino['mechanism_status'] == False :             #Si le mecanisme vient d'etre valide
        arduino['mechanism_status'] = True               #On change la valeur du mecanisme dans le dictionnaire
        ResetTimer = True                                 #On reset le timer

elif message[cpt] == "F" :
    if arduino['mechanism_status'] == True :              #Si le mecanisme vient d'etre invalide
        arduino['mechanism_status'] = False              #On change la valeur du mecanisme dans le dictionnaire
        ResetTimer = True                                 #On reset le timer

if ResetTimer == True :
    arduino['ms_noTimer'] += 1                             #Pour reset le timer
    thread = MTimer(arduino)                               #On incremente la valeur "_noTimer" dans le dictionnaire
    thread.start()                                           #On cree un nouveau thread
                                                            #On lance le nouveau thread

    send_MStoDataBase(arduino, console_message)           #On envoie l'etat du mecanisme en BDD
```

Image 26 : Code fonction get_mechanism_status() dans i2c.py



D'après le diagramme d'exigence, s'il n'y a pas de changement d'état pendant 60 secondes il faut envoyer l'état en BDD.

Pour ce faire il est donc judicieux de faire fonctionner un timer de 60s en parallèle. Il a donc fallu utiliser une nouvelle fois des threads.

Ainsi chaque Arduino possède son propre thread qui correspond au timer de l'état mécanisme. Ce thread est appelé **MSTimer**.

Le thread MSTimer exécute principalement les instructions suivantes :

```
no_thread = self.arduino['ms_noTimer'] #On definit le numero du thread

while self.running:
    time.sleep(60) #On attend 60 secondes

    if self.arduino['ms_noTimer'] != no_thread :
        self.running = False #Si le thread n'est plus le timer actuel
    else : #On arrete le thread
        #Sinon
        send_MStoDataBase(self.arduino, console_message) #On envoie l'etat des mecanismes en BDD
```

Image 27 : Code thread MSTimer dans i2c.py

- Gérer l'état des actionneurs

On appelle la fonction **get_actuator_status()**.

Elle exécute principalement les instructions suivantes :

```
cpt = 5 #La partie capteur du message commence au 6eme caractere
for actuator_name in arduino['actuator_status'] : #Pour chaque actionneur du mecanisme

    if message[cpt] == "T" :
        if arduino['actuator_status'][actuator_name] != True : #Si l'actionneur selectionne vient d'etre valide
            arduino['actuator_status'][actuator_name] = True #On change la valeur de l'actionneur dans le dictionnaire
            ResetTimer = True #On reset le timer

    elif message[cpt] == "F" :
        if arduino['actuator_status'][actuator_name] != False : #Si l'actionneur selectionne vient d'etre invalide
            arduino['actuator_status'][actuator_name] = False #On change la valeur de l'actionneur dans le dictionnaire
            ResetTimer = True #On reset le timer

    cpt+=1

if ResetTimer == True : #Pour reset le timer
    arduino['as_noTimer'] += 1 #On incremente la valeur "_noTimer" dans le dictionnaire
    thread = ASTimer(arduino) #On cree un nouveau thread
    thread.start() #On lance le nouveau thread

send_AStoDataBase(arduino, console_message) #On envoie la valeur des actionneurs en BDD
```

Image 28 : Code fonction get_actuator_status() dans i2c.py

De même que pour l'état d'un mécanisme, d'après le diagramme d'exigence s'il n'y a pas de changement d'état pendant 60 secondes il faut envoyer l'état en BDD. Pour ce faire il a donc fallu utiliser une nouvelle fois des threads.

Ainsi chaque Arduino possède son propre thread qui correspond au timer de l'état des actionneurs. Ce thread est appelé **ASTimer**. Le Thread ASTimer est presque le même que MSTimer.



- Gérer la valeur des capteurs

On appelle la fonction `get_sensor_data()`.

Elle exécute principalement les instructions suivantes :

```

cpt = 11
for sensor_name in arduino['sensor_data'] :
    if message[cpt] == "T" :
        if arduino['sensor_data'][sensor_name] != True :
            arduino['sensor_data'][sensor_name] = True
            ResetTimer = True
    elif message[cpt] == "F" :
        if arduino['sensor_data'][sensor_name] != False :
            arduino['sensor_data'][sensor_name] = False
            ResetTimer = True
    elif message[-1] == "X":
        data_sensor, cpt = get_sensor_int_data(message, cpt)
        if data_sensor != arduino['sensor_data'][sensor_name] :
            arduino['sensor_data'][sensor_name] = data_sensor
            ResetTimer = True
    cpt += 1

if ResetTimer == True :
    arduino['sd_noTimer'] += 1
    thread = SDTimer(arduino)
    thread.start()

send_SDtoDataBase(arduino, console_message)

```

#La partie capteur du message commence au 12eme caractere
#Pour chaque capteur du mecanisme
#Si le capteur selectionne vient d'etre valide
#On change la valeur du capteur dans le dictionnaire
#On reset le timer
#Si le capteur selectionne vient d'etre invalide
#On change la valeur du capteur dans le dictionnaire
#On reset le timer
#Si le capteur possede une valeur numerique
#On recupere la valeur numerique
#Si la valeur numerique vient de changer
#On change la valeur du capteur dans le dictionnaire
#On reset le timer
#Pour reset le timer
#On incremente la valeur "_noTimer" dans le dictionnaire
#On cree un nouveau thread
#On lance le nouveau thread
#On envoie la valeur des actionneurs en BDD

Image 29 : Code fonction `get_sensor_data()` dans `i2c.py`

D'après le diagramme d'exigence s'il n'y a pas de changement d'état pendant 5 secondes max il faut envoyer l'état en BDD.

Pour ce faire il est judicieux de faire fonctionner un timer de 4s en parallèle. Il a donc fallu utiliser une nouvelle fois des threads.

Ainsi chaque Arduino possède son propre thread qui correspond au timer de la valeur des capteurs. Ce thread est appelé **SDTimer**.

Le thread `SDTimer` exécute principalement les instructions suivantes :

```

no_thread = self.arduino['sd_noTimer']
while self.running:
    time.sleep(4)
    if self.arduino['sd_noTimer'] != no_thread :
        self.running = False
    else :
        send_SDtoDataBase(self.arduino, console_message)

```

#On definit le numero du thread
#On attend 4 secondes
#Si le thread n'est plus le timer actuel
#On arrete le thread
#Sinon
#On envoie la valeur des capteurs en BDD

Image 30 : Code thread `SDTimer` dans `i2c.py`



B. Programme Arduino

La partie i2c du programme de chaque Arduino consiste à :

- Envoyer les informations du mécanisme à la Raspberry
- Exécuter les ordres reçus

Programme

Afin de communiquer avec la Raspberry il a d'abord fallu assigner une adresse esclave unique à chaque Arduino. De telle sorte que la Raspberry identifie les Arduinos suivant leur adresse.

Une adresse est définie de cette manière en tête de tous les programmes Arduino :

```
#define SLAVE_ADDRESS 0x15 //initialisation de l'Arduino avec l'adresse 0x15
```

Image 31 : Code en tête de tous les programmes Arduinos

De plus la communication i2c est établie grâce à la bibliothèque **Wire** :

```
#include "Wire.h" //bibliothèque nécessaire pour la communication i2c
```

Image 32 : Code en tête de tous les programmes Arduinos

Envoyer les informations du mécanisme à la Raspberry

Pour envoyer le message i2c on exécute les instructions suivantes :

```
void send_status() {
    Wire.write(getMessagei2c().c_str()); //On écrit le message i2c dans l'objet Wire
}

void setup() {
    Serial.begin(9600);
    Wire.begin(SLAVE_ADDRESS); //On indique à l'objet Wire l'adresse esclave utilisé par l'Arduino
    Wire.onRequest(send_status); //On envoie le messagei2c sur le bus i2c
}
```

Image 33 : Code dans tous les programmes Arduinos

Exécuter les ordres reçus depuis la Raspberry

Pour exécuter les ordres envoyés depuis la Raspberry et a posteriori depuis le PC de supervision, le programme nécessite les instructions suivantes :

```
void receive_order(int numBytes) {
    String data_received;

    while(Wire.available() > 0) { //Tant que le message i2c reçu n'est pas fini
        char c = Wire.read(); //On lit le caractère suivant du message sur le bus i2c
        data_received += String(c); //On ajoute le caractère du message aux données reçues
    }

    if(data_received != "2") { //Si les données reçues sont bien un message d'ordre
        execute_order(data_received); //On exécute les ordres du message d'ordre
    }
}

void setup() {
    Serial.begin(9600);
    Wire.begin(SLAVE_ADDRESS); //On indique à l'objet Wire l'adresse esclave utilisé par l'Arduino
    Wire.onReceive(receive_order); //On récupère le message s'ordre reçu sur le bus i2c via la fonction receive_order
}
```

Image 34 : Code dans tous les programmes Arduinos



VI. Test unitaire

Pour garantir la fiabilité du programme, un test unitaire a été mis en place. En cette période de confinement il repose uniquement sur la partie i2c puisqu'il n'a pas été possible de tester les 2 mécanismes par manque de matériels.

Eléments testés :			Le script I2c.py et la partie i2c des programmes Arduinos						
Objectif du test :			Récupérer sur la Raspberry les informations mécanismes suivant le diagramme d'exigence + Exécuter les ordres sur les Arduinos						
Nom du testeur :			Constantin MINOS			Date :	16/05/2020		
Moyens mis en œuvre :			Logiciel : Putty/Arduino			Matériel : une Raspberry et deux Arduinos	Outil de développement : Python/Arduino		
-----TEST -----							Longueur du câble de la liaison i2c :	20 cm	1m
Etape	Arduino 1			Arduino 2			Résultat		
	Etat mécanisme	Etat d'1 actionneur	Etat capteur interrupteur à clef	Etat mécanisme	Etat d'1 actionneur	Valeur Capteur de Poids			
1	Validé L'Arduino 1 a validé le statut	Invalidé	Invalidé	Invalidé	Validé La Raspberry a donné un ordre	0g	OK Annexe 1	OK Annexe 2	
2	Validé	Validé L'Arduino 1 a validé le statut	Invalidé	Invalidé	Validé	0g	OK	OK	
3	Invalidé La Raspberry a donné un ordre	Validé	Validé L'Arduino 1 a validé le statut	Invalidé	Validé	0g	OK	OK	
4	Invalidé	Validé	Invalidé L'Arduino 1 a invalidé le statut	Validé L'Arduino 2 a validé le statut	Validé	0g	OK	OK	
5	Validé La Raspberry a donné un ordre	Invalidé L'Arduino 1 a invalidé le statut	Invalidé	Validé	Invalidé La Raspberry a donné un ordre	50g L'Arduino 2 a détecté 50g	OK Annexe 3	OK Annexe 4	
6	Invalidé La Raspberry a donné un ordre	Validé La Raspberry a donné un ordre	Validé L'Arduino 1 a validé le statut	Invalidé La Raspberry a donné un ordre	Validé L'Arduino 2 a validé le statut	20g L'Arduino 2 a détecté 20g	OK	OK	
7	Attendre 4 secondes (les états/valeurs capteurs sont envoyés en BDD)						OK Annexe 5	OK Annexe 6	
	Invalidé	Validé	Validé	Invalidé	Validé	20g			
8	Attendre 56 secondes (les états mécanismes/actionneurs sont envoyés en BDD)						OK Annexe 7	OK Annexe 8	
	Invalidé	Validé	Validé	Invalidé	Validé	20g			



VII. Fiches recettes

Dans cette partie, les quatre fiches recettes sont destinées au client. Elles permettent de valider le fonctionnement de ma tâche dans le système.

A. Observer le bon fonctionnement du mécanisme 4 : le Feu

Nom : Observer le fonctionnement du mécanisme 4 : le Feu	
Recette : Technique	
Objectif	Lancer le mécanisme afin qu'il s'exécute correctement
Élément à tester	Feu.ino
Pré requis	Ouvrir Feu.ino dans l'IDE Arduino et réaliser les câblages

Scénario				
Id	Démarche	Données	Comportement attendu	OK
1	Téléverser le programme dans une Arduino nano		Un message indique « téléversement terminé »	<input type="checkbox"/>
2	Changer la position de l'interrupteur à clef sur "on"		Le mécanisme s'exécute correctement	<input type="checkbox"/>

Rapport de test	Testé par : L'Etudiant 1	
Conformité <input type="checkbox"/> Excellente <input type="checkbox"/> Moyenne <input type="checkbox"/> Faible		
Commentaire :		Approbation :



B. Observer le bon fonctionnement du mécanisme 8 : le Riz

Nom : Observer le fonctionnement du mécanisme 8 : le Riz	
Recette : Technique	
Objectif	Lancer le mécanisme afin qu'il s'exécute correctement
Élément à tester	Riz.ino
Pré requis	Ouvrir Riz.ino dans l'IDE Arduino et réaliser les câblages

Scénario				
Id	Démarche	Données	Comportement attendu	OK
1	Téléverser le programme dans une Arduino nano		Un message indique « téléversement terminé ».	<input type="checkbox"/>
2	Placer entre 48g et 52g sur la balance		Le mécanisme s'exécute correctement.	<input type="checkbox"/>

Rapport de test	Testé par : L'Etudiant 1	
Conformité <input type="checkbox"/> Excellente <input type="checkbox"/> Moyenne <input type="checkbox"/> Faible		
Commentaire :		Approbation :



C. Observer l'envoi des informations mécanismes en BDD

Nom : Observer l'envoi des informations mécanisme en BDD	
Recette : Technique	
Objectif	Lancer le script python afin que les infos mécanismes s'envoient en BDD
Elément à tester	i2c.py
Pré requis	Configurer la Raspberry, réaliser les câblages et avoir BDD fonctionnel

Scénario				
Id	Démarche	Données	Comportement attendu	OK
1	Lancer le fichier i2c.py	Entrer la commande « python i2c.py ».	Le système est maintenant lancé.	<input type="checkbox"/>
2	Changer l'état/valeur d'un capteur et d'un actionneur		Le nouvel état/valeur du capteur et de l'actionneur a été envoyé en BDD.	<input type="checkbox"/>
3	Attendre 2 minutes		L'état/valeur des capteurs du mécanisme est envoyé en BDD toutes les 4s, celui des actionneurs toutes les minutes.	<input type="checkbox"/>

Rapport de test	Testé par : L'Etudiant 1	
Conformité <input type="checkbox"/> Excellente <input type="checkbox"/> Moyenne <input type="checkbox"/> Faible		
Commentaire :		Approbation :



D. Observer l'envoi des ordres sur les Arduinos

Nom : Observer l'envoi des ordres sur les Arduinos	
Recette : Technique	
Objectif	Lancer le script python afin que les ordres s'envoient aux Arduinos
Élément à tester	i2c.py
Pré requis	Configurer la Raspberry, réaliser les câblages et Appli Web fonctionnel

Scénario				
Id	Démarche	Données	Comportement attendu	OK
1	Lancer le fichier i2c.py	Entrer la commande « python i2c.py ».	Le système est maintenant lancé.	<input type="checkbox"/>
2	Changer l'état d'un actionneur sur l'appli Web		L'état de l'actionneur a changé.	<input type="checkbox"/>

Rapport de test	Testé par : L'Etudiant 1	
Conformité <input type="checkbox"/> Excellente <input type="checkbox"/> Moyenne <input type="checkbox"/> Faible		
Commentaire :		Approbation :



Conclusion

A. Communication de groupe

Durant tout le projet nous avons mis en place une cohésion de groupe. Pour cela, plusieurs mesures ont été prises avec notamment la réalisation de :

- Diagramme de Gantt pour se partager le travail et s'organiser dans la réalisation des tâches du projet
- Répertoire sur la plateforme GitHub pour partager son travail et échanger documents et programmes
- Chartes graphiques (Word, PowerPoint) identiques pour tous les étudiants du groupe
- Réunions régulières sur Discord (moyen de communication) pour travailler, rendre compte de son travail et s'aider

Ces mesures efficaces nous ont permis de réaliser un travail achevé et rigoureux en équipe.

D'autant plus qu'ayant dû effectuer notre projet en confinement nous avons pu faire l'expérience du travail de groupe à distance, une expérience enrichissante.

Je remercie les étudiants Joshua PINEAU, Thomas CADEAU et Corentin BRENNY pour leur participation au projet.

B. Regard critique du projet

J'ai choisi ce projet parce que je voulais découvrir l'i2c et l'univers Arduino que je n'avais encore jamais utilisé concrètement et j'ai beaucoup apprécié ma tâche. J'ai découvert et expérimenté plusieurs notions qui m'ont beaucoup intéressé. J'ai donc un avis extrêmement positif sur le projet.

C. Connaissances apportées

Ce projet m'a apporté bien des connaissances.

Au niveau Hardware j'ai appris à :

- Monter des modules électroniques sur une Arduino
- Établir une connexion i2c entre Arduinos-Raspberry via breadbord et relais
- Réaliser des schémas-montages électroniques

Et au niveau Software dans mes programmes j'ai appris à :

- Configurer (pleinement) une Raspberry
- Réaliser des threads en Python
- Programmer une Arduino
- Faire communiquer Arduinos-Raspberry via i2c

Des connaissances importantes qui me seront sûrement utiles plus tard.



De plus j'avais déjà de bonnes bases en python et en C++ (~ langage Arduino) donc j'ai pu aider mes camarades dans la réalisation de leurs tâches quand ils avaient un problème.

D. Poursuite d'étude

À la suite d'un entretien concluant, ma candidature a été retenue par l'ESAIP une école d'ingénieurs située à Angers.

Là-bas je ne sais pas encore vers quelle spécialité m'orienter mais mon choix porterait plutôt vers la cybersécurité ou pourquoi pas l'IOT(objets connectés) dans la continuité de ce projet.



Annexe

Annexe 1 :

Test unitaire Etape 1 : 20 cm de câble

=> L'Arduino 1 a validé l'état de son mécanisme

=> L'Arduino 2 a reçu l'ordre de valider l'état de l'un de ses actionneurs

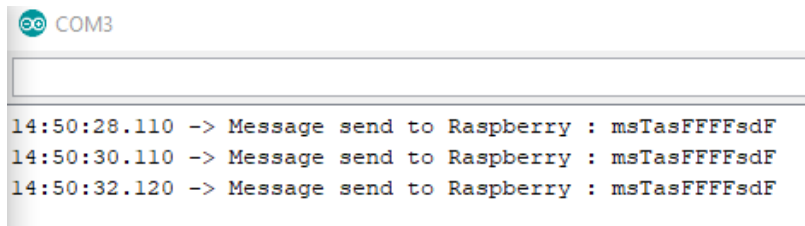


Image 35 : Capture Arduino 1 du test unitaire

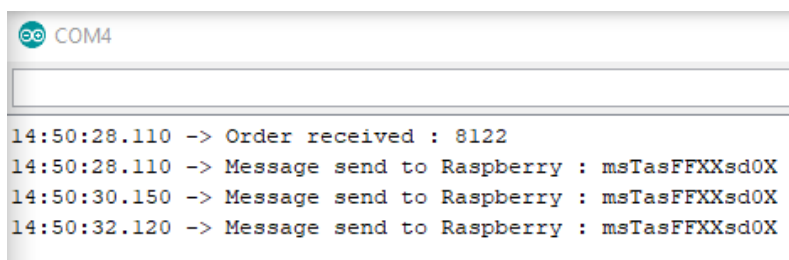


Image 36 : Capture Arduino 2 du test unitaire

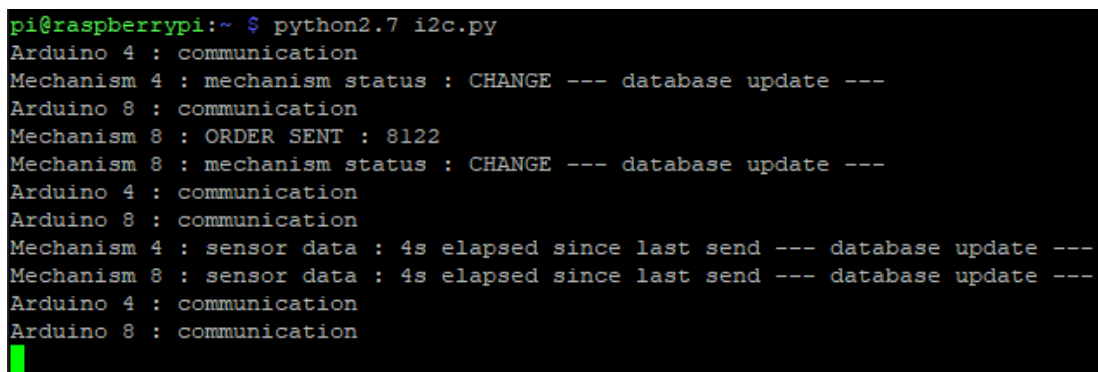


Image 37 : Capture Putty du test unitaire



Annexe 2 :

Test unitaire Etape 1 : 1m de câble

=> L'Arduino 1 a validé l'état de son mécanisme

=> L'Arduino 2 a reçu l'ordre de valider l'état de l'un de ses actionneurs

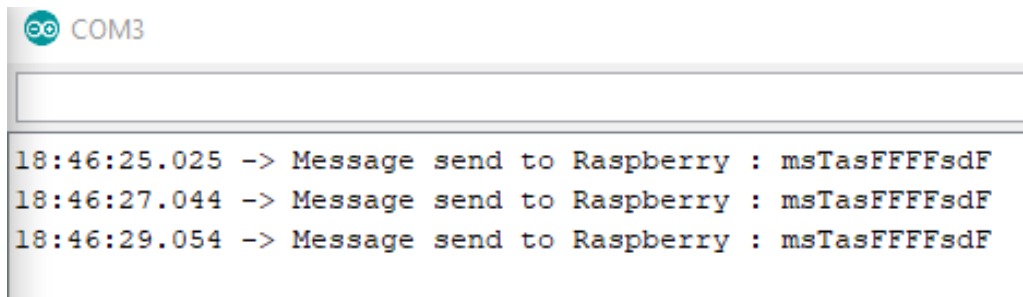


Image 38 : Capture Arduino 1 du test unitaire

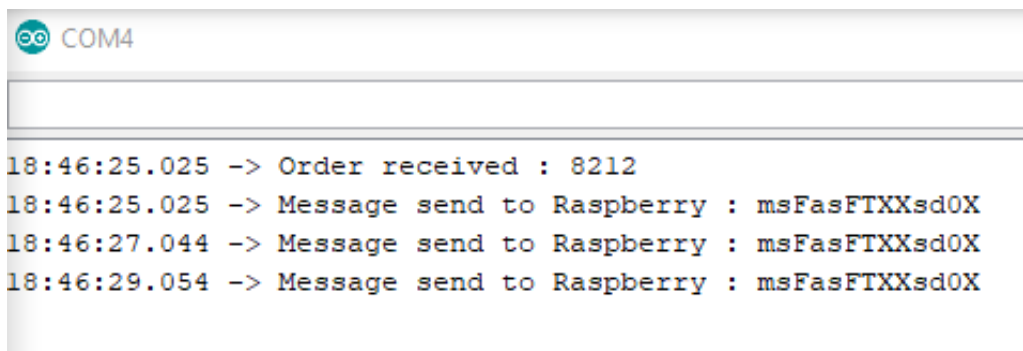


Image 39 : Capture Arduino 2 du test unitaire

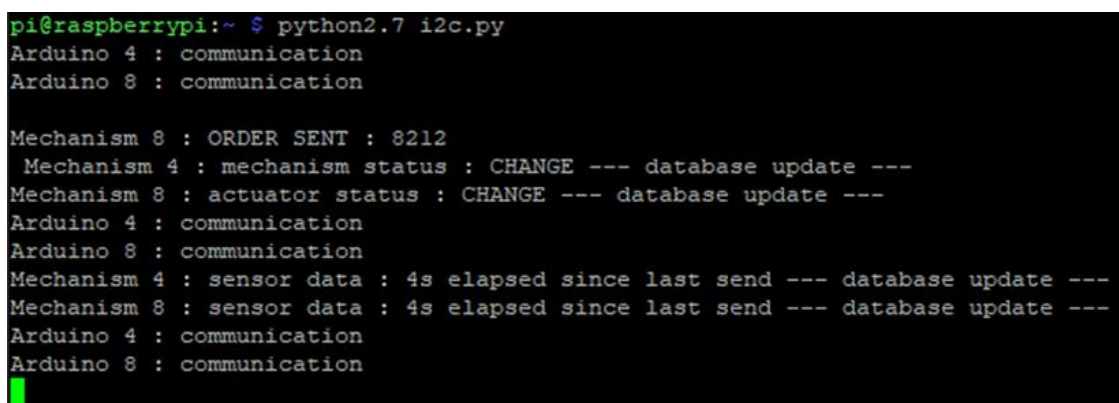


Image 40 : Capture Putty du test unitaire



Annexe 3 :

Test unitaire Etape 5 : 20 cm de câble

=> L'Arduino 1 a invalidé l'état de l'un de ses actionneurs

=> L'Arduino 1 a reçu l'ordre de valider l'état de son mécanisme

=> L'Arduino 2 a détecté 50g sur le capteur de poids

=> L'Arduino 2 a reçu l'ordre d'invalider l'état de l'un de ses actionneurs

```
COM3
17:50:57.644 -> Order received : 412222
17:50:57.644 -> Message send to Raspberry : msTasFFFFsdF
17:50:59.652 -> Message send to Raspberry : msTasFFFFsdF
17:51:01.688 -> Message send to Raspberry : msTasFFFFsdF
```

Image 41 : Capture Arduino 1 du test unitaire

```
COM4
17:50:57.647 -> Order received : 8202
17:50:57.647 -> Message send to Raspberry : msTasFFXXsd50X
17:50:59.689 -> Message send to Raspberry : msTasFFXXsd50X
17:51:01.699 -> Message send to Raspberry : msTasFFXXsd50X
```

Image 42 : Capture Arduino 2 du test unitaire

```
pi@raspberrypi:~ $ python2.7 i2c.py
Arduino 4 : communication
Mechanism 4 : ORDER SENT : 412222
Arduino 8 : communication

Mechanism 4 : mechanism status : CHANGE --- database update ---
Mechanism 8 : ORDER SENT : 8202
Mechanism 4 : actuator status : CHANGE --- database update ---
Mechanism 8 : actuator status : CHANGE --- database update ---
Mechanism 8 : sensor data : CHANGE --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
```

Image 43 : Capture Putty du test unitaire



Annexe 4 :

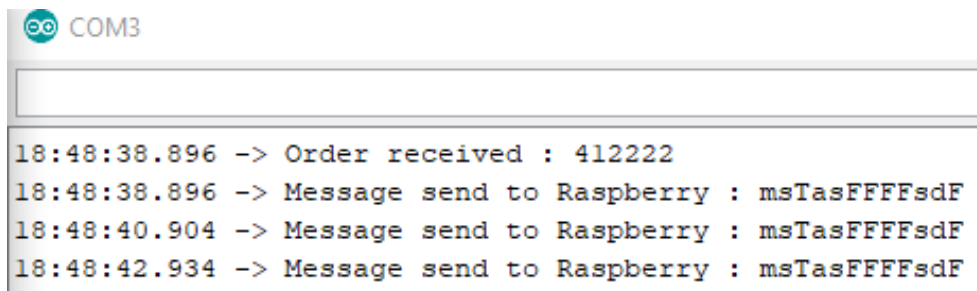
Test unitaire Etape 5 : 1m de câble

=> L'Arduino 1 a invalidé l'état de l'un de ses actionneurs

=> L'Arduino 1 a reçu l'ordre de valider l'état de son mécanisme

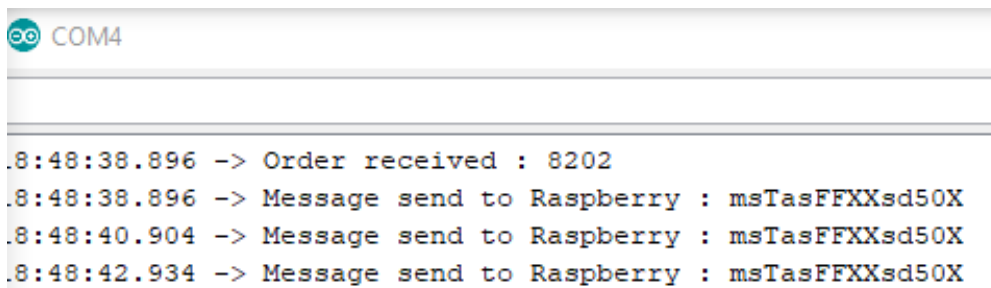
=> L'Arduino 2 a détecté 50g sur le capteur de poids

=> L'Arduino 2 a reçu l'ordre d'invalider l'état de l'un de ses actionneurs



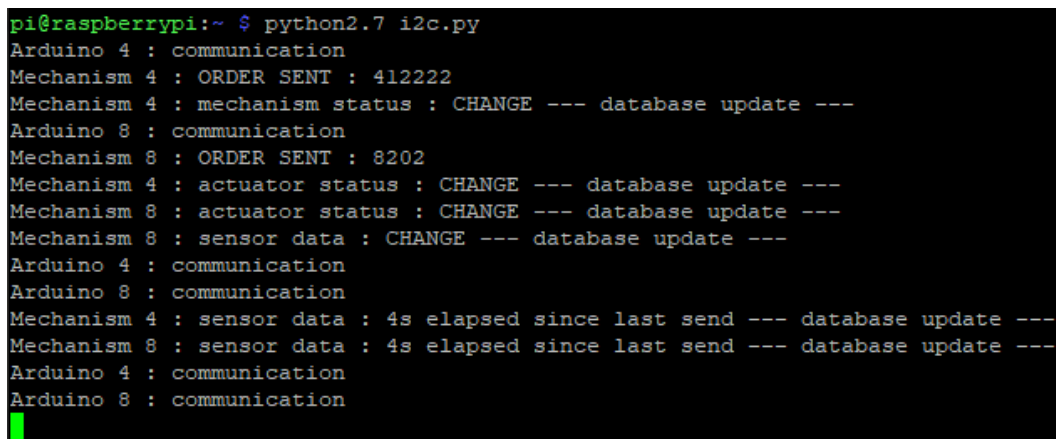
```
COM3
18:48:38.896 -> Order received : 412222
18:48:38.896 -> Message send to Raspberry : msTasFFFFsdF
18:48:40.904 -> Message send to Raspberry : msTasFFFFsdF
18:48:42.934 -> Message send to Raspberry : msTasFFFFsdF
```

Image 44 : Capture Arduino 1 du test unitaire



```
COM4
18:48:38.896 -> Order received : 8202
18:48:38.896 -> Message send to Raspberry : msTasFFXXsd50X
18:48:40.904 -> Message send to Raspberry : msTasFFXXsd50X
18:48:42.934 -> Message send to Raspberry : msTasFFXXsd50X
```

Image 45 : Capture Arduino 2 du test unitaire



```
pi@raspberrypi:~ $ python2.7 i2c.py
Arduino 4 : communication
Mechanism 4 : ORDER SENT : 412222
Mechanism 4 : mechanism status : CHANGE --- database update ---
Arduino 8 : communication
Mechanism 8 : ORDER SENT : 8202
Mechanism 4 : actuator status : CHANGE --- database update ---
Mechanism 8 : actuator status : CHANGE --- database update ---
Mechanism 8 : sensor data : CHANGE --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
```

Image 46 : Capture Putty du test unitaire



Annexe 5 :

Test unitaire Etape 6-7 : 20 cm de câble

=> Dernier envoi état/valeur capteurs il y a 4s : La Raspberry les envoie en BDD

```
pi@raspberrypi:~ $ python2.7 i2c.py
Arduino 4 : communication
Mechanism 4 : ORDER SENT : 402212
Mechanism 4 : mechanism status : CHANGE --- database update ---
Arduino 8 : communication
Mechanism 8 : ORDER SENT : 8022
Mechanism 4 : actuator status : CHANGE --- database update ---
Mechanism 8 : mechanism status : CHANGE --- database update ---
Mechanism 8 : sensor data : CHANGE --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
```

Image 47 : Capture Putty du test unitaire



Annexe 6 :

Test unitaire Etape 6-7 : 1m de câble

=> Dernier envoi état/valeur capteurs il y a 4s : La Raspberry les envoie en BDD

```
pi@raspberrypi:~ $ python2.7 i2c.py
Arduino 4 : communication
Mechanism 4 : ORDER SENT : 402212
Mechanism 4 : mechanism status : CHANGE --- database update ---
Mechanism 4 : actuator status : CHANGE --- database update ---
Arduino 8 : communication
Mechanism 8 : ORDER SENT : 8022
Mechanism 8 : mechanism status : CHANGE --- database update ---
Mechanism 8 : sensor data : CHANGE --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 8 : communication
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
```

Image 48 : Capture Putty du test unitaire



Annexe 7 :

Test unitaire Etape 8 : 20 cm de câble

=> Dernier envoi état mécanismes/actionneurs il y a 60s : La Raspberry les envoie en BDD

```
pi@raspberrypi: ~
Arduino 8 : communication
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Arduino 4 : communication
Arduino 8 : communication
Mechanism 8 : actuator status : 60s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 4 : mechanism status : 60s elapsed since last send --- database update ---
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 4 : actuator status : 60s elapsed since last send --- database update ---
Mechanism 8 : mechanism status : 60s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Arduino 4 : communication
Arduino 8 : communication
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Arduino 4 : communication
Arduino 8 : communication
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
```

Image 49 : Capture Putty du test unitaire



Annexe 8 :

Test unitaire Etape 8 : 1m de câble

=> Dernier envoi état mécanismes/actionneurs il y a 60s : La Raspberry les envoie en BDD

```

pi@raspberrypi: ~
Arduino 8 : communication
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : mechanism status : 60s elapsed since last send --- database update ---
Mechanism 4 : actuator status : 60s elapsed since last send --- database update ---
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 8 : actuator status : 60s elapsed since last send --- database update ---
Mechanism 8 : mechanism status : 60s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication
Arduino 4 : communication
Arduino 8 : communication
Mechanism 4 : sensor data : 4s elapsed since last send --- database update ---
Mechanism 8 : sensor data : 4s elapsed since last send --- database update ---
Arduino 4 : communication
Arduino 8 : communication

```

Image 50 : Capture Putty du test unitaire