

## Test 2 : La boîte noire

Dans de nombreux domaines industriels tels que l'aéronautique, l'automobile ou encore le ferroviaire, il est indispensable de disposer de systèmes capables d'enregistrer en temps réel les paramètres de fonctionnement des engins. Ces systèmes, appelés communément **boîtes noires**, permettent de conserver des informations critiques relatives à l'activité de l'appareil. En cas de dysfonctionnement ou d'accident, les données recueillies servent à analyser les événements ayant précédé l'incident et à comprendre ses causes.

### ✓ **Principe des boîtes noires industrielles :**

- Enregistrement de données physiques (vitesse, position, accélérations, etc.)
- Stockage des communications des opérateurs ou pilotes
- Conservation des données même après un arrêt brutal de l'appareil

Dans le cadre de la **Tekbot Robotics Challenge**, l'objectif de ce test est de concevoir une version simplifiée d'une boîte noire. Ce système devra être capable de :

- ✓ **Mesurer la position et la vitesse de déplacement dans l'espace** à l'aide d'un capteur combiné **gyroscope et accéléromètre (MPU6050)**.
- ✓ **Transmettre en temps réel ces données** à une station de contrôle par l'intermédiaire d'une communication **I2C**.
- ✓ **Afficher les mesures reçues** sur un écran **LCD** connecté à la station de contrôle.

Ce projet est divisé en deux parties principales :

- **La boîte noire** : une boîte autonome qui collecte les données de mouvement et les transmet.
- **La station de contrôle** : un récepteur qui reçoit les données et les affiche à l'utilisateur.

## **1. Objectifs de ce test :**

L'objectif pédagogique est de permettre aux participants de :

- Mettre en œuvre des communications entre microcontrôleurs.
- Utiliser des capteurs inertIELS pour mesurer des grandeurs physiques.
- Organiser un projet de conception électronique complet, du prototype au système fonctionnel.

## **2. Comprendre la communication I2C :**

Le protocole I2C (Inter-Integrated Circuit) est une méthode de communication série largement utilisée dans les systèmes embarqués pour connecter des microcontrôleurs, capteurs et périphériques sur un même bus, en utilisant seulement deux fils :

- ✓ **SDA (Serial Data Line)** : pour la transmission des données.
- ✓ **SCL (Serial Clock Line)** : pour la synchronisation de la transmission.

### **✓ Fonctionnement du bus I2C :**

- **Maître et esclaves** : un périphérique maître (ici, le microcontrôleur de la boîte noire) contrôle la communication et peut interroger un ou plusieurs périphériques esclaves (le capteur MPU6050 et la station de contrôle).
- **Adresses uniques** : chaque esclave possède une adresse unique, permettant au maître de communiquer individuellement avec chaque périphérique sans interférence.
- **Protocole de communication :**
  - Le maître initie la communication par un signal de démarrage (Start).

- Il envoie l'adresse de l'esclave ciblé, suivie d'un bit indiquant s'il souhaite lire ou écrire des données.
  - L'esclave concerné répond (ACK) et la transmission de données commence.
  - Après chaque octet transmis, le récepteur envoie un acquittement (ACK) ou un refus (NACK).
  - La communication se termine par un signal d'arrêt (Stop).
- **Simplicité et flexibilité** : Le bus I2C permet de connecter plusieurs composants avec un câblage minimal, ce qui simplifie l'intégration et la maintenance des systèmes embarqués.

### **3. Choix du matériel pour ce test :**

Composants	Référence / Broches	Rôles
<b>ATmega328P (Maître)</b>	DIP28, SDA (A4), SCL (A5), INT (2)	Lecture MPU6050, maître I2C vers esclave externe
<b>ATmega328P (Esclave)</b>	DIP28, SDA (A4), SCL (A5)	Réception I2C, pilotage écran LCD
<b>MPU6050 (GY-521)</b>	I2C, adresse 0x68	Accéléromètre + gyroscope, DMP intégré
<b>Module MD _MAX72XX avec l'écran matriciel</b>	SPI (DATA 11, CLK 13, CS 10)	Affichage matrice LED 8×8 (retour visuel de direction)
<b>Écran LCD 16×2 I2C</b>	SDA, SCL, VCC, GND	Affichage texte de la station de contrôle

<b>Convertisseur USB-TTL</b>	TX, RX, VCC, GND, DTR	Programmation et debug des ATmega328P
<b>Quartz 16 MHz + Condensateurs</b>	XTAL1/XTAL2	Horloge stable pour chaque ATmega328P
<b>Régulateur 5 V (AMS1117)</b>	VIN (7–12 V) → VCC, GND	Alimentation régulée
<b>LED témoin</b>	Broche 9 (Maître)	Indicateur visuel d'envoi I2C
<b>Veroboard</b>	—	Montage définitif, soudure des composants

Pour l'emplacement de notre alimentation, on a prévu deux sources. Une par adaptateur sur le secteur et l'autre une batterie de 7.4V. On a prévu qu'elle demeure dans la boite noire.

#### ✓ *Pourquoi cette décision ?*

Pour garantir le bon fonctionnement et la fiabilité de notre système, nous avons prévu deux sources d'alimentation distinctes et complémentaires comme dans le test 1 :

#### ➤ **Alimentation principale : adaptateur secteur**

- ✓ L'adaptateur secteur alimente l'ensemble du système, y compris la boîte noire.
- ✓ Cette solution centralisée permet de simplifier la gestion énergétique globale du système.

#### ➤ **Alimentation de secours : batterie interne (7.4V) intégrée dans la boîte noire.**

La batterie est placée directement dans la boîte noire pour assurer l'autonomie du système en cas de coupure d'alimentation principale (panne secteur, débranchement, déplacement du système, maintenance...).

➤ **Pourquoi avoir positionné la batterie directement dans la boîte noire ?**

**a) Réduction des pertes par effet Joule**

- Plus le courant doit parcourir une longue distance via des câbles, plus il subit des pertes par effet Joule ( $P = RI^2$ ).
- En plaçant la batterie directement dans la boîte noire, on supprime la traversée de câbles longue distance pour l'alimentation de secours.
- Cela limite les pertes énergétiques, augmente l'efficacité énergétique et améliore la stabilité de l'alimentation.

**b) Sécurité et fiabilité accrues**

- En cas de choc, de déplacement ou de coupure accidentelle du câble secteur, la batterie interne garantit que la boîte noire reste alimentée sans aucune dépendance externe.
- La présence de la batterie dans la boîte noire réduit les risques de faux contacts ou de déconnexion accidentelle.

**c) Portabilité et simplicité de maintenance**

- La boîte noire peut être déplacée ou isolée sans nécessiter de câblage externe supplémentaire.
- Les opérations de maintenance sont facilitées.

**d) Robustesse globale du système**

- La batterie interne est protégée mécaniquement dans la boîte noire contre les agressions extérieures (poussière, humidité, chocs...).

Par ailleurs, on a ajouté un écran matriciel pour le projet au niveau de la boîte noire.

✓ *Quel est en fait son importance ?*

L'ajout d'un écran matriciel 8x8 dans la boîte noire apporte plusieurs avantages significatifs :

✓ **Une visualisation instantanée des directions**

L'écran permet d'afficher des flèches directionnelles en temps réel, offrant une représentation visuelle immédiate des mouvements détectés par le capteur. Cela aide les utilisateurs à comprendre rapidement dans quelle direction la boîte noire se déplace.

✓ **Un retour d'information immédiat**

Avant la transmission des données à la station de contrôle, l'écran matriciel fournit un retour d'information direct. Cela permet de valider rapidement le bon fonctionnement du système et de détecter d'éventuelles erreurs avant que les données ne soient envoyées.

✓ **Une simplicité d'utilisation**

L'affichage de flèches sur un écran matriciel est simple et intuitif. Cela rend le système plus accessible aux utilisateurs inexpérimentés, qui peuvent facilement interpréter les mouvements sans avoir besoin de comprendre des données techniques complexes.

✓ **Une intégration avec le système**

L'écran est intégré dans le même boîtier que le microcontrôleur et le capteur, favorisant une conception compacte et efficace. Cela permet de réduire le câblage et d'optimiser l'espace disponible à l'intérieur de la boîte noire.

✓ **Esthétique du projet**

L'ajout d'un écran matriciel améliore également l'esthétique du projet, rendant la boîte noire plus attrayante visuellement.

Pour conclure, on utilisera, les matériels du test 1 ajoutés de quelques autres nécessaires pour la communication notamment un Atmega328P et ses composants, un écran matriciel, ...

Dans le cadre de ce test, nous avons réalisé l'impression 3D de trois boîtes destinées à la boîte noire, à la station de contrôle et à l'alimentation. Cette approche présente de nombreux avantages tant pratiques qu'esthétiques :

✓ **Une personnalisation et adaptabilité**

L'impression 3D permet de concevoir des boîtes sur mesure pour chaque composant, garantissant un ajustement parfait. Cela optimise l'espace intérieur et facilite l'assemblage des éléments.

✓ **Une protection des composants**

Les boîtes imprimées en 3D protègent efficacement les composants électroniques contre les chocs, la poussière et l'humidité, ce qui augmente la durabilité et la fiabilité du système.

✓ **Une esthétique et présentation**

Une conception soignée améliore l'apparence générale du projet. Des boîtes bien réalisées contribuent à rendre le projet plus professionnel et attrayant lors de présentations ou d'expositions.

✓ **Une facilité d'accès**

Les boîtes peuvent être dotées d'ouvertures ou de panneaux amovibles, facilitant ainsi l'accès aux composants pour la maintenance ou les mises à jour.

✓ **Une optimisation du câblage**

En intégrant des supports pour le câblage et les connexions, les boîtes imprimées en 3D aident à organiser et gérer le câblage de manière efficace, réduisant ainsi les risques d'erreurs ou de courts-circuits.

#### **4. Réalisation du circuit :**

##### **A. Test du microcontrôleur ATmega328P**

Avant d'intégrer le microcontrôleur ATmega328P dans le circuit final de la boîte noire, il est recommandé de le tester sur une breadboard (plaqué d'essai).

❖ **Montage minimal** : placer le microcontrôleur sur la breadboard avec :

- Une alimentation de 5 V.
- Un cristal de 16 MHz.
- Deux condensateurs de 22 pF.
- Une résistance de 10 kΩ entre la broche RESET et Vcc.

❖ **Test de fonctionnement** : téléchargez un code simple (par exemple, un clignotement de LED) pour vérifier que le microcontrôleur fonctionne correctement.

##### **B. Gravure du bootloader**

Le bootloader permet à l'ATmega328P d'être programmé comme un Arduino classique. Si vous utilisez une puce neuve, il faut graver ce bootloader :

- **Utilisation d'un programmeur** : utilisez une carte Arduino comme programmeur ou un programmeur externe.
- **Branchements** : connectez les broches MOSI, MISO, SCK, RESET, Vcc et GND de l'Arduino à celles de l'ATmega328P.
- **Gravure** :

- Dans l'IDE Arduino, sélectionnez : Arduino as ISP, puis "Burn Bootloader".
- Choisissez le modèle approprié selon votre montage :
  - ATmega328 sur une breadboard (oscillateur interne de 8 MHz) ou
  - Arduino Uno si vous utilisez un cristal de 16 MHz.

### **C. Conception du schéma sous KiCad**

Le schéma du circuit sera conçu pour mesurer les mouvements à l'aide du capteur MPU6050 (accéléromètre et gyroscope) tout en intégrant l'écran matriciel 8x8 pour afficher les flèches directionnelles.

- ❖ Les mesures seront traitées par le microcontrôleur ATmega328P.
- ❖ Les résultats seront affichés sur un écran LCD et sur l'écran matriciel.
- ❖ Tous les composants communiqueront via le protocole I2C, réduisant ainsi le nombre de connexions nécessaires.

### **D. Schéma, composants et assemblage**

Le schéma a été réalisé dans KiCad, en utilisant des composants de la librairie officielle. Les connexions sont nommées de façon explicite pour faciliter la lecture.

➤ *Liste des composants :*

- **ATmega328P**
- **MPU6050**
- **Écran LCD I2C**
- **Écran matriciel 8x8**
- **Quartz, condensateurs et résistances**

- Le quartz génère l'horloge du microcontrôleur.
- Les condensateurs (22 pF) et la résistance (10 kΩ) sont typiques.
- **Bouton poussoir** : Pour réinitialiser manuellement le microcontrôleur.

### ➤ *Fonctionnement global*

- À la mise sous tension, le microcontrôleur maître initialise le capteur MPU6050 ainsi que la communication avec le microcontrôleur esclave.
- Il lit en continu les valeurs d'accélération et de rotation fournies par le capteur.
- Ces données sont ensuite traitées et converties dans un format compréhensible.
- L'écran matriciel affiche une direction (par exemple flèche) correspondant à l'orientation détectée par le capteur.
- Le microcontrôleur maître interroge le microcontrôleur esclave pour vérifier s'il est prêt à recevoir des données : l'esclave répond par un 1 s'il est prêt, ou par un 0 sinon.
- Une fois la confirmation obtenue, le maître envoie les données au second microcontrôleur : un entier représentant la direction, ainsi qu'une variable flottante représentant l'accélération.
- Le microcontrôleur esclave reçoit ces informations, décode le code directionnel, enregistre les données, puis affiche la direction sur un écran LCD.
- Enfin, le maître effectue une nouvelle interrogation pour obtenir la confirmation que le message a bien été reçu par l'esclave.
- Une LED clignote à chaque appel du microcontrôleur maître, permettant ainsi de visualiser l'activité de communication.

➤ **Alimentation :**

- Tous les composants sont alimentés en +5 V (VCC).
- Le MPU6050 est compatible avec des logiques de 3.3 V ou 5 V.
- L'alimentation est réalisée par un adaptateur ou une batterie.

➤ **Protocole de communication utilisé :**

Les capteurs d'accélération et de déplacement transmettent des données précises et lisibles via les broches I2C :

- **SDA** : reçoit, traite et transmet les données vers l'écran LCD.
- **SCL** : génère un signal d'horloge pour synchroniser l'échange de données.

## **E. Soudure des composants**

– **Préparation :**

- Préchauffez le fer à souder (300 - 350 °C).
- Nettoyez la panne.
- Préparez les composants et le veroboard.

– **Soudure des composants traversants (THT) :**

- Placez chaque composant.
- Soudez une patte, ajustez, puis soudez les autres.
- Coupez l'excédent.

– **Soudure des composants SMD (si applicable) :**

- Utilisez de la pâte à souder et du flux.

- Fixez un coin, puis soudez les autres broches (drag soldering ou point par point).

**NB:** utilisez un fil à souder fin (0,38 mm recommandé), évitez les ponts de soudure, et vérifiez que chaque joint est brillant et conique. Inspectez visuellement et corrigez les défauts éventuels.

## F. Programmation et test du circuit assemblé

- **Connexion du convertisseur USB-série** : branchez le module FTDI (ou équivalent) aux broches RX, TX, Vcc, GND (et DTR/RESET si présent) de l'ATmega328P.
- **Téléversement du code** : utilisez l'IDE Arduino pour charger un programme de test (par exemple, un clignotement de LED ou une lecture du capteur).
- **Test fonctionnel** : vérifiez le bon fonctionnement :
  - Du microcontrôleur.
  - De la communication I2C (entre le MPU6050, l'écran LCD).
  - De l'alimentation.

Cette section décrit en détail le processus de réalisation du circuit pour le projet, en intégrant les nouveaux éléments tels que l'écran matriciel 8x8, et en mettant en avant chaque étape clé, de la préparation à la programmation et aux tests.

Vidéo de démonstration

•

## **5. Explication détaillée du code MPU6050 + LCD I2C avec ATmega328P**

### **✓ Rappels sur les objectifs du projet**

- ✓ **Lire l'orientation** via le MPU6050 (accéléromètre + gyroscope) ;
- ✓ **Afficher des flèches directionnelles** sur une matrice LED 8x8 (contrôlée par MAX72xx) ;
- ✓ **Transmettre les données** à un autre microcontrôleur via I2C ;
- ✓ **Recevoir une confirmation** de l'esclave via une LED ;

Nous rappelons que le test 2 est une suite du test 1. Il y aura certaines lignes sur lesquelles nous ne détaillerons pas l'explication.

### **i. Code MASTER**

Dans ce système, le **MASTER** (maître) est le microcontrôleur qui :

- lit les données du capteur de mouvement **MPU6050** (accéléromètre et gyroscope),
- détermine la direction et l'accélération,
- envoie ces informations à l'esclave (**SLAVE**) via le protocole **I2C**,
- vérifie si l'esclave est prêt à recevoir ou a bien reçu les données.

On utilise la communication **I2C**, un protocole permettant à plusieurs appareils de communiquer ensemble avec seulement deux fils (SDA et SCL).

### **1. Inclusion des bibliothèques**

```
1 #include <Wire.h>
2 #include "I2Cdev.h"
3 #include "MPU6050_6Axis_MotionApps20.h"
4 #include <MD_MAX72xx.h>
5 #include <SPI.h>
```

- ❖ Wire.h : gestion de la communication I2C entre le microcontrôleur et les périphériques (MPU6050, esclave).
- ❖ I2Cdev.h : simplifie les opérations I2C, offre des fonctions utilitaires pour lire et écrire sur le bus.
- ❖ MPU6050\_6Axis\_MotionApps20.h : fournit l'accès aux fonctionnalités avancées du DMP du MPU6050 pour obtenir quaternion et angles.
- ❖ MD\_MAX72xx.h : contrôle de la matrice LED 8x8 via le driver MAX7219.
- ❖ SPI.h : nécessaire pour la communication SPI avec le driver MAX7219.

## 2. Définition du matériel et des objets

```
 1 // Type de matrice : FC16_HW pour modules classiques 8x8
 2 #define HARDWARE_TYPE MD_MAX72XX::FC16_HW
 3 #define MAX_DEVICES 1 // 1 module de 8x8
 4 #define CLK_PIN 11
 5 #define DATA_PIN 12
 6 #define CS_PIN 10
 7
 8 MD_MAX72XX mx = MD_MAX72XX(HARDWARE_TYPE, DATA_PIN, CLK_PIN, CS_PIN, MAX_DEVICES);
 9
10
11 MPU6050 mpu;
12
13 int send_dir = 0;
14 float send_acc = 0;
15 const uint8_t addr_slav = 0x08;
16 int ledConfirm = 9;
17
18 #define INTERRUPT_PIN 2
19 volatile bool mpuInterrupt = false;
```

- Constantes de l'écran matricielle : configuration du type de module, nombre de modules et broches de commande (CLK, DATA, CS).
- mx : objet pour piloter la matrice LED.
- mpu : objet pour communiquer avec le capteur MPU6050.
- addr\_slav : adresse I2C de l'esclave.
- send\_dir / send\_acc : variables pour stocker les données à envoyer.
- ledConfirm : broche de la LED de confirmation.
- INTERRUPT\_PIN / mpuInterrupt : gestion de l'interruption matérielle du DMP.

### 3. Déclaration des variables de statut et buffers

```
1 void dmpDataReady() { mpuInterrupt = true; }
2
3 bool dmpReady = false;
4 uint8_t mpuIntStatus;
5 uint8_t devStatus;
6 uint16_t packetSize, fifoCount;
7 uint8_t fifoBuffer[64];
8
9 Quaternion q;
10 VectorFloat gravity;
11 float ypr[3];
12 VectorInt16 aa, aaReal;
13
14 float prevYaw = 0, prevPitch = 0, prevRoll = 0;
15 const float seuilAngle = 5.0;           // Seuil en degrés pour rotations
16 const int   seuilAccZ   = 600;          // À ajuster après calibration (ex. 500-1200)
17 // float zFilt      = 0;                // Valeur filtrée de l'accélération Z
18 // const float alpha = 0.2;              // Coef. filtre passe-bas (0<alpha<1)
19 float accZ_filtered = 0;
20 const float alpha = 0.3; // coefficient filtre passe-bas
21 String direction = "";
22
```

- dmpReady, devStatus : indiquent l'état d'initialisation et de disponibilité du DMP.
- packetSize, fifoCount, fifoBuffer : gèrent la lecture depuis le FIFO du DMP.
- q, gravity, ypr : pour calculer quaternion, vecteur gravité et angles yaw/pitch/roll.
- aa, aaReal : accélérations brutes et linéaires.
- prevYaw/Pitch/Roll : pour calculer les variations et détecter le mouvement.
- seuilAngle, seuilAccZ : seuils de détection.
- accZ\_filtered, alpha : filtre passe-bas sur l'accélération verticale.
- direction : chaîne représentant le mouvement détecté.

#### 4. Initialisation - setup()

La fonction `setup()` configure le matériel et prépare le système en quatre étapes distinctes.

##### i. Initialisation des communications

```
1 void setup() {  
2     Wire.begin();  
3     Serial.begin(115200);  
4     Serial.print("Init MPU6050...");  
5     delay(1000);
```

- Wire.begin() : initialise le bus I2C pour communiquer avec le capteur et l'esclave.
- Serial.begin() : active la console série à 115200 bauds pour afficher le statut et les erreurs.

### ii. Configuration des broches et du capteur

```
1 mpu.initialize();  
2 pinMode(INTERRUPT_PIN, INPUT);  
3 pinMode(ledConfirm, OUTPUT);
```

- mpu.initialize() : lance la communication avec le MPU6050.
- pinMode(INTERRUPT\_PIN, INPUT) : prépare la broche d'interruption du DMP.
- pinMode(ledConfirm, OUTPUT) : configure la LED pour indiquer les transmissions.

### iii. Calibration et activation du DMP

```

1 // Offsets calibrés (à ajuster pour ton module)
2 mpu.setXAccelOffset(874);
3 mpu.setYAccelOffset(-853);
4 mpu.setZAccelOffset(-21);
5 mpu.setXGyroOffset(-59);
6 mpu.setYGyroOffset(20);
7 mpu.setZGyroOffset(-57);
8
9 devStatus = mpu.dmpInitialize();
10 if (devStatus == 0) {
11     mpu.CalibrateAccel(6);
12     mpu.CalibrateGyro(6);
13     mpu.setDMPEnabled(true);
14     attachInterrupt(digitalPinToInterrupt(INTERRUPT_PIN), dmpDataReady, RISING);
15     mpuIntStatus = mpu.getIntStatus();
16     dmpReady = true;
17     packetSize = mpu.dmpGetFIFOPacketSize();
18     // lcd.clear();
19     // lcd.setCursor(0,0);
20     Serial.print("MPU6050 prêt!");
21     delay(1000);
22 }
23 else {
24     // lcd.clear();
25     // lcd.setCursor(0,0);
26     Serial.print("Erreur DMP:");
27     Serial.print(devStatus);
28     while (1);
29 }

```

- setXAccelOffset(...), etc. : corrigent les biais du capteur pour des mesures précises.
- dmpInitialize() : prépare le Digital Motion Processor du MPU6050.
- CalibrateAccel/Gyro : affinent la précision.
- attachInterrupt(...) : lie l'interruption matérielle à la remise à jour des données.
- packetSize et dmpReady : indiquent que le DMP est opérationnel.

#### iv. Initialisation de la matrice LED

```
1 mx.begin();
2   mx.control(MD_MAX72XX::INTENSITY, 5);
3   mx.clear();
```

- mx.begin() : initialise le driver MAX7219 et la matrice.
- control(INTENSITY, 5) : régle l'intensité d'affichage.
- clear() : vide l'affichage avant la première utilisation.

## 5. Boucle Principale — loop()

La fonction loop() s'exécute en continu et se décompose en cinq étapes clés pour assurer la lecture, le traitement et la transmission des données du MPU6050.

### i. Vérifications initiales

```
1 void loop() {
2   if (!dmpReady) return;
3   if (!mpuInterrupt && fifoCount < packetSize) return;
4
5   mpuInterrupt = false;
```

- dmpReady : indique si le DMP a été correctement initialisé.
- mpuInterrupt & fifoCount : vérifient qu'une nouvelle trame est prête dans le FIFO.

## ii. Gestion du FIFO

```
1 mpuIntStatus = mpu.getIntStatus();
2 fifoCount    = mpu.getFIFOCount();
3
4 // Gestion overflow FIFO
5 if ((mpuIntStatus & 0x10) || fifoCount == 1024) {
6     mpu.resetFIFO();
7     Serial.print("Overflow FIFO!");
8     return;
9 }
10
11 // Lecture du FIFO
12 while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();
13 mpu.getFIFOBytes(fifoBuffer, packetSize);
14 fifoCount -= packetSize;
```

- mpu.getIntStatus() : lit le code d'interruption pour détecter erreurs ou overflow.
- mpu.resetFIFO() : vide le FIFO si débordement.
- mpu.getFIFOBytes() : copie les données prêtes pour traitement.

## iii. Acquisition et calcul des données

```
1 // Calcul des angles
2 mpu.dmpGetQuaternion(&q, fifoBuffer);
3 mpu.dmpGetGravity(&gravity, &q);
4 mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
5 float yaw = ypr[0] * 180/M_PI;
6 float pitch = ypr[1] * 180/M_PI;
7 float roll = ypr[2] * 180/M_PI;
8
9 // Accélération linéaire
10 mpu.dmpGetAccel(&aa, fifoBuffer);
11 mpu.dmpGetLinearAccel(&aaReal, &aa, &gravity);
12
13 float acc = sqrt(pow(aaReal.x, 2) + pow(aaReal.y, 2) + pow(aaReal.z, 2));
14
15 // Filtrage passe-bas simple sur accZ
16 accZ_filtered = alpha * aaReal.z + (1 - alpha) * accZ_filtered;
```

- dmpGetQuaternion / dmpGetYawPitchRoll : calculent orientations (yaw/pitch/roll).
- dmpGetLinearAccel : obtient l'accélération en supprimant l'effet gravité.
- Filtre passe-bas : réduit le bruit sur l'accélération verticale.

#### iv. Détection de la direction et affichage

```
1  detectDirection(yaw, pitch, roll, accZ_filtered);
2
3  // Envoi de l'entier caractéristique de la direction
4  if (direction == "Haut") {
5      send_dir = 0;
6      displayArrowUp();
7  }
8  else if (direction == "Bas") {
9      send_dir = 1;
10     displayArrowDown();
11 }
12 else if (direction == "Droite" || direction == "Rot Droite") {
13     send_dir = 2;
14     displayArrowRight();
15 }
16 else if (direction == "Gauche" || direction == "Rot Gauche") {
17     send_dir = 3;
18     displayArrowLeft();
19 }
20 else if (direction == "Avant") {
21     send_dir = 4;
22     displayCenterDot();
23 }
24 else if (direction == "Arriere") {
25     send_dir = 5;
26     displayCenterDot();
27 }
28
```

- detectDirection() : compare variations d'angles et seuil d'accélération.
- displayArrowXXX() : dessine la flèche correspondante sur la matrice.

## v. Transmission I2C vers l'esclave

```
1
2     if(slaveConfirm()) {
3         clignotLed();
4
5         Wire.beginTransmission(addr_slav);
6         Wire.write((uint8_t*)&send_dir, sizeof(send_dir));
7         Wire.write((uint8_t*)&acc, sizeof(acc));
8         Wire.endTransmission();
9
10    if(slaveConfirm()) return;
11 }
12 else digitalWrite (ledConfirm, HIGH);
13 }
```

- slaveConfirm() : vérifie que l'esclave est prêt ou a bien reçu les données.
- clignotLed() : retour visuel d'envoi réussi.

## 6. Fonctions utilitaires

### i. Interruption DMP

```
1 void dmpDataReady() { mpuInterrupt = true; }
2
```

Elle marque qu'une nouvelle trame est disponible.

### ii. Détection de direction

```
1 void detectDirection(float yaw, float pitch, float roll, float accZf) {  
2     float dPitch = pitch - prevPitch;  
3     float dRoll  = roll  - prevRoll;  
4     float dYaw   = yaw   - prevYaw;  
5  
6     if (abs(accZf) > seuilAccZ) {  
7         direction = (accZf > 0) ? "Haut" : "Bas";  
8     }  
9     else if (abs(dPitch) > abs(dRoll) && abs(dPitch) > seuilAngle) {  
10        direction = (dPitch > 0) ? "Avant" : "Arriere";  
11    }  
12    else if (abs(dRoll) > seuilAngle) {  
13        direction = (dRoll > 0) ? "Droite" : "Gauche";  
14    }  
15    else if (abs(dYaw) > seuilAngle) {  
16        direction = (dYaw > 0) ? "Rot Droite" : "Rot Gauche";  
17    }  
18    else {  
19        direction = "Stable";  
20    }  
21  
22    prevYaw   = yaw;  
23    prevPitch = pitch;  
24    prevRoll  = roll;  
25 }
```

Compare variations d'angles et seuil d'accélération pour définir la direction.

### iii. Affichage de l'écran matriciel

```
1 // Fonction : point central  
2 void displayCenterDot() {  
3     mx.clear();  
4     mx.setPoint(3, 3, true);  
5     mx.setPoint(3, 4, true);  
6     mx.setPoint(4, 3, true);  
7     mx.setPoint(4, 4, true);  
8 }
```

```
1 // Flèche gauche
2 void displayArrowLeft() {
3     mx.clear();
4     mx.setPoint(3, 3, true);
5     mx.setPoint(4, 3, true);
6     mx.setPoint(3, 2, true);
7     mx.setPoint(4, 2, true);
8     mx.setPoint(2, 1, true);
9     mx.setPoint(5, 1, true);
10    mx.setPoint(3, 0, true);
11    mx.setPoint(4, 0, true);
12 }
```

```
1 // Flèche droite
2 void displayArrowRight() {
3     mx.clear();
4     mx.setPoint(3, 4, true);
5     mx.setPoint(4, 4, true);
6     mx.setPoint(3, 5, true);
7     mx.setPoint(4, 5, true);
8     mx.setPoint(2, 6, true);
9     mx.setPoint(5, 6, true);
10    mx.setPoint(3, 7, true);
11    mx.setPoint(4, 7, true);
12 }
```

```
1 // Flèche haut
2 void displayArrowUp() {
3     mx.clear();
4     mx.setPoint(3, 3, true);
5     mx.setPoint(3, 4, true);
6     mx.setPoint(2, 3, true);
7     mx.setPoint(2, 4, true);
8     mx.setPoint(1, 2, true);
9     mx.setPoint(1, 5, true);
10    mx.setPoint(0, 3, true);
11    mx.setPoint(0, 4, true);
12 }
13
```

```
1 // Flèche basse
2 void displayArrowDown() {
3     mx.clear();
4     mx.setPoint(4, 3, true);
5     mx.setPoint(4, 4, true);
6     mx.setPoint(5, 3, true);
7     mx.setPoint(5, 4, true);
8     mx.setPoint(6, 2, true);
9     mx.setPoint(6, 5, true);
10    mx.setPoint(7, 3, true);
11    mx.setPoint(7, 4, true);
12 }
```

- displayCenterDot() : un point central.
- displayArrowUp/Down/Left/Right() : permet de dessiner des flèches.

#### iv. Clignotement LED



```
1 void clignotLed() {  
2     for (int i = 0; i < 4; i++) {  
3         digitalWrite (ledConfirm, HIGH);  
4         delay(500);  
5         digitalWrite (ledConfirm, LOW);  
6         delay(500);  
7     }  
8 }
```

Cette ligne indique visuellement l'envoi de données.

#### v. Confirmation esclave



```
1 bool slaveConfirm() {  
2     Wire.requestFrom(addr_slav, 1);  
3     if (Wire.available()) {  
4         uint8_t status = Wire.read();  
5         return status == 1;  
6     }  
7     return false;  
8 }
```

Elle vérifie si l'esclave est prêt ou si la donnée a été reçue.

## ii. Code SLAVE

Dans ce système, le **SLAVE** (esclave) est un microcontrôleur qui :

- ✓ attend les données envoyées par le **MASTER** (maître),
- ✓ affiche les données reçues sur un écran LCD,
- ✓ confirme au MASTER s'il est prêt à recevoir ou s'il a bien traité les données.

On utilise la communication **I2C**, un protocole permettant à plusieurs appareils de communiquer ensemble avec seulement deux fils (SDA et SCL).

### 1. Les bibliothèques utilisées

```
1 #include <Wire.h>
2 #include <LiquidCrystal_I2C.h>
```

- Wire.h : pour gérer la communication I2C.
- LiquidCrystal\_I2C.h : pour piloter l'écran LCD via I2C.

### 2. Déclaration et initialisation de l'écran LCD

```
1 LiquidCrystal_I2C LCD(0x27, 16, 2);
```

L'écran LCD a l'adresse I2C 0x27, et possède 16 colonnes et 2 lignes.

### 3. Déclaration des variables globales

```
1 const uint8_t addr_slav = 0x08;
2 int rec_dir = 0;
3 float rec_acc = 0;
4
5 volatile bool dataReceived = false;
6 bool readyToReceive = true;
7 bool receivedOK = false;
8
9 String direction = "";
```

- addr\_slav : adresse I2C de l'esclave (ici 0x08).
- rec\_dir : reçoit un code entier représentant la direction.
- rec\_acc : reçoit la valeur de l'accélération.
- dataReceived : indique si de nouvelles données ont été reçues.
- readyToReceive : indique si l'esclave est prêt à recevoir.
- receivedOK : indique si les données ont été traitées.
- direction : texte correspondant à la direction à afficher.

### 4. Fonction setup() - Initialisation

```
1 void setup() {  
2     // put your setup code here, to run once:  
3     Wire.begin(addr_slav);  
4     Wire.onReceive(receiveData);  
5     Wire.onRequest(confirm);  
6     LCD.begin(16, 2);  
7     LCD.backlight();  
8     LCD.clear();  
9     LCD.setCursor(0,0);  
10    LCD.print("Station active! ...");  
11 }
```

Cet ensemble active le bus I2C, initialise l'écran LCD et affiche un message de démarrage : « Station active ! ».

## 5. Fonction loop() — Affichage des données

```
1 void loop() {
2     // put your main code here, to run repeatedly:
3
4     if (dataReceived && !receivedOK) {
5         direction = getDirection(rec_dir);
6
7         LCD.clear();
8         LCD.setCursor(0, 0);
9         LCD.print("Dir: ");
10        LCD.print(direction);
11        LCD.setCursor(0, 1);
12        LCD.print("a = ");
13        LCD.print(rec_acc, 2);
14        delay(500);
15
16        // Indique que les données ont été traitées
17        receivedOK = true;
18        readyToReceive = true;
19    }
20 }
```

- Lorsque de nouvelles données arrivent (dataReceived passe à vrai), il affiche la direction et l'accélération.
- Ensuite, il indique qu'il a terminé le traitement (receivedOK = true).

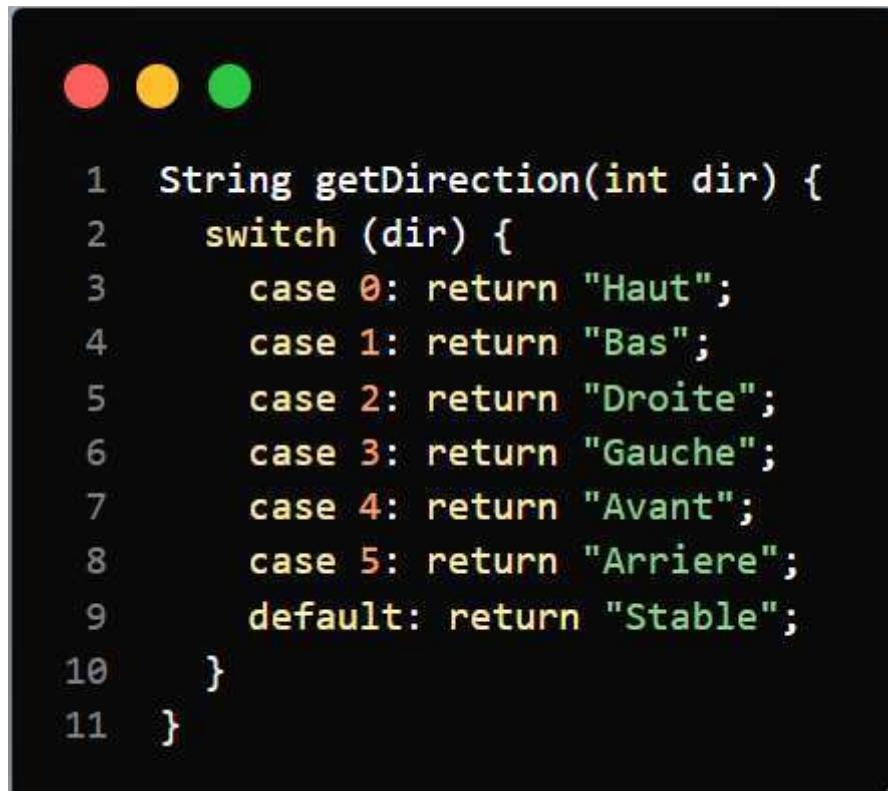
## 6. Fonction receiveData() - Réception des données I2C

```
1 void receiveData (int taille) {
2     if (taille == sizeof(int) + sizeof(float)) {
3         Wire.readBytes((char*)&rec_dir, sizeof(rec_dir));
4         Wire.readBytes((char*)&rec_acc, sizeof(rec_acc));
5         dataReceived = true;
6         readyToReceive = false;
7         receivedOK = false;
8     }
9 }
```

- Quand le maître envoie les données, cette fonction est automatiquement appelée.
- Elle lit d'abord un entier (rec\_dir), puis un flottant (rec\_acc).
- Met à jour les drapeaux pour indiquer que de nouvelles données sont prêtes.

Les drapeaux sont les variables dataReceived, readyToReceive, receivedOK.

## 7. Fonction getDirection() — Conversion du code en texte



```

1 String getDirection(int dir) {
2     switch (dir) {
3         case 0: return "Haut";
4         case 1: return "Bas";
5         case 2: return "Droite";
6         case 3: return "Gauche";
7         case 4: return "Avant";
8         case 5: return "Arriere";
9         default: return "Stable";
10    }
11 }
```

Transforme le code entier rec\_dir en un texte lisible à afficher sur le LCD.

## 8. Fonction confirm() - Communication de statut au maître



```
1 void confirm() {
2     if (!dataReceived) {
3         Wire.write(readyToReceive ? 1 : 0);
4     }
5     else {
6         Wire.write(receivedOK ? 1 : 0);
7     }
8 }
```

Quand le maître interroge l'esclave, cette fonction lui répond :

- 1 si l'esclave est prêt ou a bien reçu/traité.
- 0 sinon.

Pour tout résumer, ce code assure

- **Une communication fiable** : grâce aux drapeaux (readyToReceive, receivedOK), le système évite les conflits de transmission.
- **Une simplicité** : la structure est très claire : réception → traitement → affichage → confirmation.
- **Une robustesse** : les fonctions onReceive et onRequest permettent une synchronisation automatique avec le maître.

## CONCLUSION

Le test 2 de la boîte noire illustre l'application concrète des systèmes de collecte de données en temps réel, essentiels pour la sécurité et l'analyse post-incident dans divers secteurs industriels. En concevant un prototype simplifié intégrant le capteur MPU6050, la communication I2C, des écrans LCD et matriciel, les participants ont acquis des

compétences pratiques en électronique embarquée, lecture de capteurs et interfaces utilisateur. L'intégration d'une alimentation autonome et de boîtiers imprimés en 3D renforce la robustesse et la portabilité du système. Ce projet démontre ainsi la synergie entre matériel et logiciel, tout en ouvrant des perspectives d'innovation pour la collecte et l'analyse de données critiques.