Université Lille1
Sciences et Technologies

POLYTECH
LILLE

Projet IMA 4 2012 – 2013

Vision déportée

Par

Adel ALJANE & Nicolas HUSSE

Encadrant : Nicolas DEFRANCE & Alexandre BOE

Remerciement

Dans le cadre de ce projet réalisé en collaboration avec le département Conception Mécanique, nous tenons à remercier M. Rodolphe Astori et M. Aurelien Benechet pour leur participation.

Nous tenons également à remercier M. Defrance et M. Boé pour avoir encadré ce projet.

Sommaire

Introduction

A l'occasion du projet de fin d'année IMA4 à Polytech'Lille, nous avons choisi de réaliser un système permettant la vision à distance. Ce système a un intérêt pour des applications civiles ou militaires demandant l'inspection des lieux ou la commande des appareils à distance.
Ce projet comporte plusieurs domaines de compétences telles que la détection des mouvements, la communication sans fil, la commande des moteurs et le traitement des données vidéo.

Ayant déjà réalisé un suivit de notre projet sur la page internet suivante : http://projets-ima.plil.net/mediawiki/index.php?title=Vision_d%C3%A9port%C3%A9e , ce dossier s'orientera plus sur la partie technique du projet.

Chapitre 1 : Spécifications

Avant de commencer la conception et le développement du système, nous avons réalisé une spécification fonctionnelle du système et par la suite une spécification technique.

1. Spécifications fonctionnelles

Le système est composé de 2 parties:

1.1. Une carte d'acquisition avec un capteur fixé sur la tête

Cette partie doit réaliser les tâches suivantes:

- Relever les données relatives à la position de la tête.
- Traiter les données.
- Envoyer les données via une communication sans fil.
- Recevoir le flux vidéo.

1.2. Une tourelle mécanique équipée d'une caméra.

La 2ème partie doit effectuer les tâches suivantes:

- Recevoir les données via la communication sans fil.
- Commander les moteurs
- Acquérir le flux vidéo.
- Envoyer le flux vidéo.

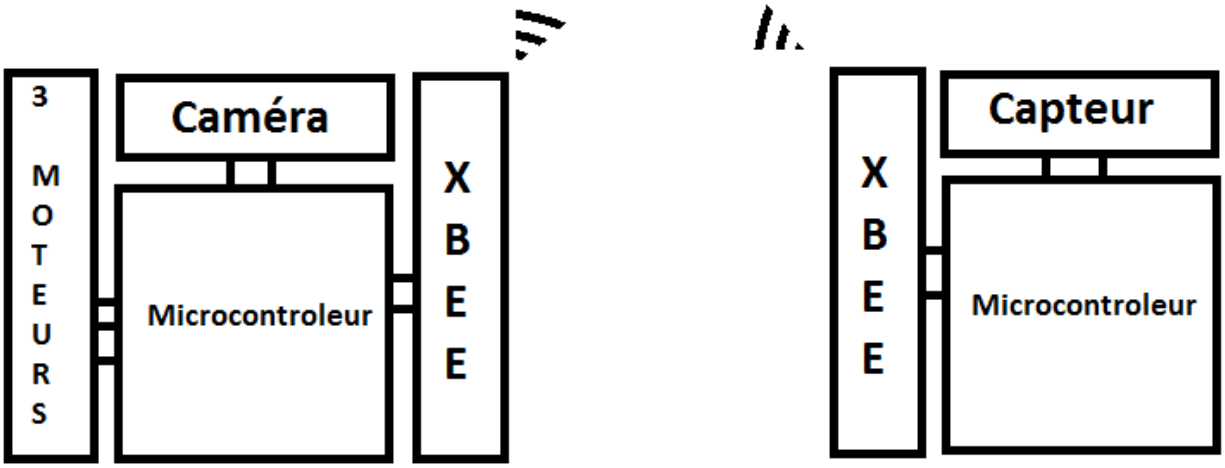


Schéma simplifié du système

2. Spécifications techniques

Dans cette partie, nous avons réalisé une spécification de l'architecture de notre système à travers la spécification fonctionnelle détaillée ci-dessus.

2.1 - Choix du microcontrôleur

Le microcontrôleur disponible pour ce projet est le module Arduino construit autour d'un microcontrôleur Atmel AVR (ATmega328 ou ATmega2560).

2.2 - Communication Sans fil

L'échange des informations entre le capteur et la tourelle ne nécessitant que peu d'information et se faisant à courte distance, nous nous sommes orientés vers la communication sans fil Xbee.

2.3 - Choix du capteur

Ayant besoin de récupérer la position de la tête, nous avons commencé le projet avec un accéléromètre ADXL345 (Sparkfun). Cependant, ce capteur ne pouvant pas détecter les mouvements de lacet (utile à la détection de l'orientation gauche-droite de la tête) nous nous sommes orientés vers un capteur regroupant un accéléromètre et un gyroscope. En effet, on a utilisé un composant Sparkfun 6DOF avec (ADXL345 et ITG3200).

2.4 - Choix du moteur

Pour pouvoir facilement orienter la tourelle, nous souhaitons commander les moteurs par une commande angulaire. Pour répondre à cette consigne, nous pouvons utiliser soit des moteurs pas-à-pas soit des servomoteurs. Les moteurs pas-à-pas fournissant une certaine puissance que l'Arduino ne peut apporter, nous devons utiliser un circuit de puissance alors que les servomoteurs non pas besoin de ce circuit. Les servomoteurs sont des systèmes motorisés capables d'atteindre des positions prédéterminées, puis de les maintenir. Les avantages de celui-ci nous on pousser à sélectionner ce type de moteur.

Chapitre 2 : Communication Xbee

Pour permettre un transfert de données sans fil entre les capteurs et la tourelle, nous avons utilisé une communication sans fil de la technologie Xbee.

1. Présentation

La technologie Xbee repose sur le protocole de communication ZigBee basée sur la norme IEEE 802.15.4 pour les réseaux à dimension personnelle (Wireless Personal Area Networks : WPAN).

Les modules utilisés dans ce projet sont les modules RF Xbee 802.15.4 pour OEM dont les caractéristiques sont les suivantes:

- Fréquence de 2,4 GHz
- Débit de données RF de 250 Kbits/s
- Réseaux sans fils multipoints.

2. Configuration

La configuration du Xbee consiste à choisir :

- Un identifiant réseau: 3005 (commande ATID)
- Une adresse pour chaque module: (adresses : 10 et 11)
  1er Module ATMY10 - ATDL11
  2éme Module ATMY11 - ATDL10
- Enregistrement en Mémoire ROM: ATWR
- Canal de communication dans la bande de Xbee : (canal par défaut)

Nous avons développé un programme qui se lance au démarrage du microcontrôleur permettant la configuration automatique du module Xbee à travers une écriture sur la liaison série par défaut de la carte.
Pour entrer dans le mode configuration, il existe le mot clé “+++” et pour retourner au mode communication “ATCN”.
Pour chaque commande, on doit vérifier le bon fonctionnement par réception de la chaîne de caractères “OK”.

3. Utilisation

Pour communiquer à travers une liaison Xbee, On écrit et on lit sur la liaison série par défaut du microcontrôleur. En effet, durant la communication, nous ne pouvons pas utiliser la même liaison pour vérifier les valeurs ou débugger le programme.
Par besoin de suivre le déroulement de notre programme à des fin de débogage, nous avons réalisé une liaison série supplémentaire à l’aide de la bibliothèque Software serial.

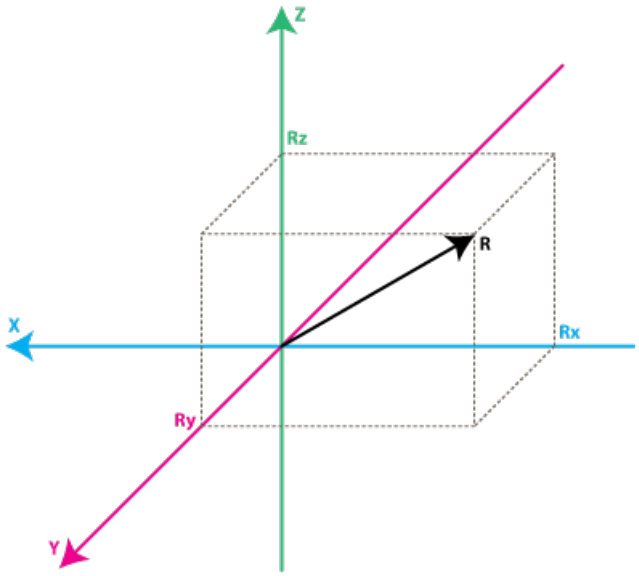
Chapitre 3 : Acquisition des données accéléromètre / gyroscope

Les mouvements de la tête humaines se décomposent en trois composantes : inclinaison gauche-droite, inclinaison avant-arrière et orientation gauche-droite (ou mouvement de lacet). Pour pouvoir recréer le mouvement de la tête, il nous faut déterminer les mouvements de la tête sur ces trois axes afin de les convertir en angle compréhensible par les servomoteurs.

Nous nous sommes d'abords orientés sur l'utilisation d'un accéléromètre car c'est un des capteurs de "position" les plus couramment utilisés.

1. L'accéléromètre

Un accéléromètre donne une accélération en G (1 G = 9,80665 $m.s^{-2}$), il prend en compte la pesanteur en plus de la variation de vitesse. Dans le cas d'un accéléromètre trois axes, on peut déterminer la direction de l'accélération comme le montre le modèle 1 suivant. En effet l'accéléromètre nous fournis les trois composantes (Rz, Ry, et Rx) de l'accélération R.



Modèle 1 représentant les accélérations mesurées par un accéléromètre
Source : http://www.instructables.com/id/Accelerometer-Gyro-Tutorial/

1. 1. La communication SPI

L'ADXL345 est un accéléromètre qui communique soit en SPI soit en I2C. Ces deux bus sont régulièrement rencontrés dans des montages à base de microcontrôleurs et de capteurs. Ayant déjà utilisé le SPI auparavant, je me suis naturellement orienté vers se bus en début de projet. Cependant, l'ajout du gyroscope, par la suite, m'a imposé l'utilisation du bus I2C comme nous le verrons plus tard.

Dans le cas d'une communication SPI, la communication entre le maître (ici, l'Arduino) et le ou les esclaves s'effectue au moyen de quatre liaisons minimum : l'horloge fourni par le maitre pour la synchronisation, la réception de donnée, l'émission de donnée et enfin la sélection de l'esclave.

Pour pouvoir communiquer avec l'ADXL345, il faut mettre le pin de sélection de l'esclave associé à l'état haut. Une fois les messages reçus et/ou envoyés, il faut mettre se pin à l'état bas.
Pour pouvoir configurer, il suffit d'écrire dans le registre voulut en envoyant d'abord le registre puis la valeur à y stocker.
De la même façon, pour pouvoir récupérer une valeur, il faut écrire n'importe quelle valeur dans le registre que nous souhaitons lire et ensuite récupérer la valeur. C'est donc au maître de toujours demander la donnée dont il a besoin.
Il est à noter que les valeurs des accélérations renvoyées par le capteur sont codées sur 10 bits mais envoyés sur deux octets (0x32 et 0x33 pour X, 0x34 et 0x35 pour Y et 0x36 et 0x37 pour Z).

1. 2. La configuration et le calibrage

L'ADXL345 peut-être paramétré sur une sensibilité de 2G, 4G, 8G ou bien encore 16G qui permet d'adapter la précision de la mesure en fonction du besoin. Dans notre cas, 4G fut le meilleur compromis après plusieurs tests. Pour cela, nous envoyons la valeur 0x01 dans le registre 0x31.

Avant toute utilisation des valeurs reçues, il faut d'abord effectuer un calibrage afin de déterminer la position initiale du capteur. En effet, les calculs que nous verrons par la suite ne sont valables que dans le cas où les trois axes valent 0 à l'état initial. Hors, le capteur ne sera pas forcément bien positionné sur la tête et les valeurs ne seront alors pas exactes. On se retrouvera alors avec une erreur dans la position calculé. Il faut donc prendre en compte cette différence.
Pour pouvoir réaliser cette calibration, nous considérons qu'au démarrage de l'Arduino, la tête de l'utilisateur se trouve en position initiale. Il suffit alors de récupérer plusieurs mesures et d'en faire la moyenne après avoir lancer les mesures en envoyant 0x08 dans le registre 0x2D. La moyenne obtenue sera soustraie aux mesures effectuées par la suite.

1. 3. Le traitement des données

Pour pouvoir récupérer les angles qui nous sont utiles, il faut appliquer une règle de math : dans un plan 0xy, l'angle d'un vecteur AB par rapport à l'axe 0x vaut :

$$\theta_{AB} = \arctan\left(\frac{(y_B - y_A)}{(x_B - x_A)}\right)$$

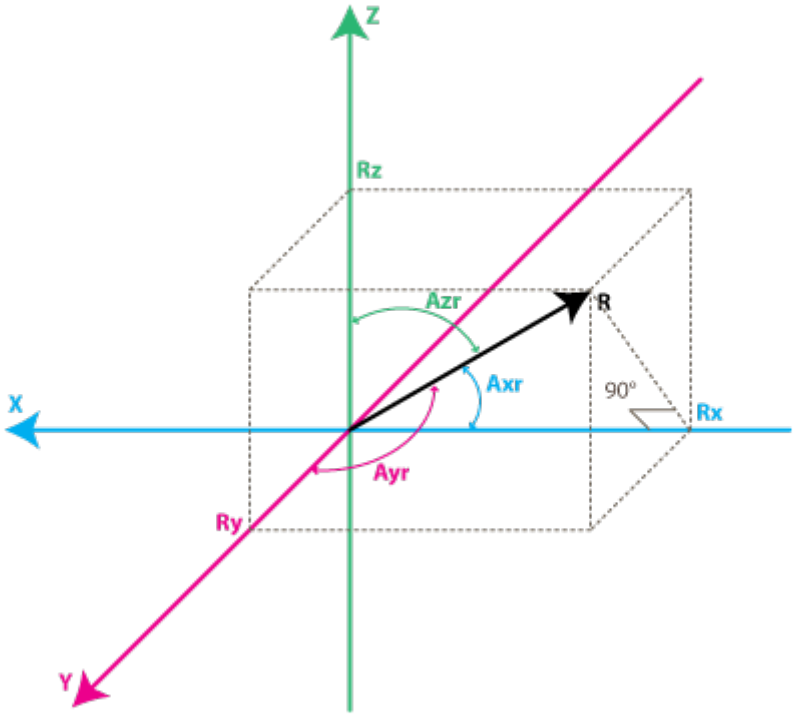
Sachant que dans le cas ou $x_B - x_A$ vaut 0, il faut appliquer une autre formule.
La bibliothèque "math" propose la fonction atan2 qui prend en compte le cas du $x_B - x_A = 0$.
Appliquer à notre programme, on a donc :
- Pour l'inclinaison avant-arriéré : Θ_{ZX} = atan2(zRaw,xRaw)
- Pour l'inclinaison gauche-droite : Θ_{ZY} = atan2(zRaw,yRaw)

Où xRaw, yRaw et zRaw sont les trois valeurs renvoyés par le capteur.

De la même façon, nous aurions pu penser que l'orientation gauche-droite de la tête (mouvement de lacet) aurait pu être: Θ_{XY} = atan2(xRaw, yRaw) mais l'accéléromètre n'étant pas physiquement capable de détecter un mouvement de lacet, cette équation est fausse.
Suite à ce problème, nous nous sommes orientés vers l'utilisation d'un gyroscope en plus de l'accéléromètre.

2. Le gyroscope

Un gyroscope l'orientation absolue d'un système dans l'espace et fournis une vitesse angulaire en $°.s^{-1}$.



Modèle 2 représentant les vitesses angulaires mesurées par un gyroscope
Source : http://www.instructables.com/id/Accelerometer-Gyro-Tutorial/

Nous utilisons donc maintenant une platine regroupant un gyroscope ITG3200 et le même accéléromètre que précédemment. Cependant, la platine ne propose que la connexion I2C, il a donc fallut retravailler la communication avec l'accéléromètre.

Platine avec ADXL345 et ITG3200

2.1. La communication I2C

Contrairement au SPI, l'I2C ne nécessite que l'utilisation de deux liaisons : l'horloge qui, comme pour le bus SPI, est fournis pas le maître pour la synchronisation et l'échange de données. Pour pouvoir choisir le périphérique esclave, il suffit de connaître son adresse. Celle-ci est fixe et est fournis par le constructeur.

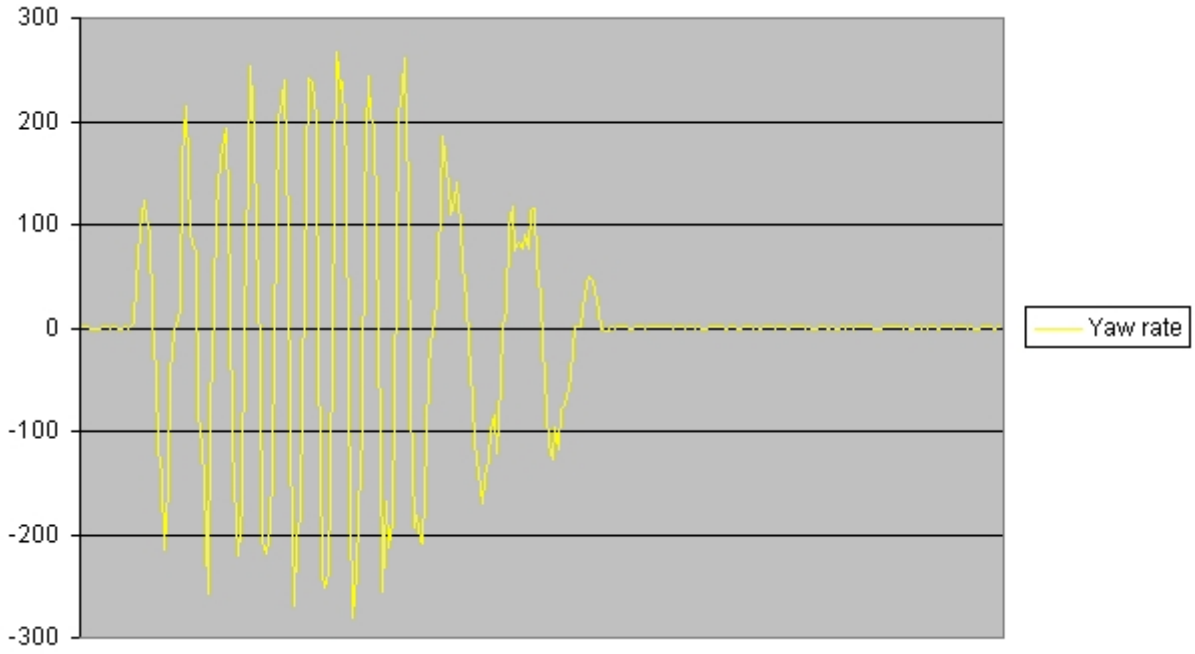
Pour communiquer avec les capteurs, le protocole se rapproche de celui du SPI à part qu'il ne faut pas mettre à l'état haut pour sélectionner un composant mais d'envoyer son adresse sur le bus. Le reste des étapes restent les mêmes que pour le bus concurrent.

Contrairement à l'accéléromètre, le gyroscope code ses valeurs sur deux octets et les stocks dans deux octets : 0x1D et 0x1E pour X, 0x1F et 0x20 pour Y et 0x21 et 0x22 pour Z.

2. 2. La configuration et le calibrage

Comme pour l'ADXL345, l'ITG3200 à besoin d'une calibration avant d'être utilisé. Celle-ci s'effectuant de la même façon, elle ne sera pas de nouveau détaillé ici.

Cette étape a d'abord été codée à la main mais après observation des résultats, les valeurs obtenues étaient très parasité comme vous pouvez le voir sur le graphique suivant en effectuant un mouvement de 90°.



Le choix c'est alors porté sur une bibliothèque qui à l'aide d'un filtre, réduit ces parasite.

2. 3. Le traitement des données

Ayant une vitesse angulaire, multiplier la vitesse par l'intervalle de temps entre les mesures (imposés ici à 200ms) résulte à un angle parcouru pendant cet intervalle de temps. Il faut donc additionner les intervalles pour obtenir l'angle correspondant à l'orientation de la tête

$$\Theta_{XY} = \Theta_{XY\_prec} + zRaw * dt$$

Où Θ_{XY_prec} est l'angle précédemment calculé et zRaw la valeur récupérer par le gyroscope.

Cette somme d'angle implique aussi la somme des erreurs et implique dans le temps des erreurs de plus en plus importantes.

C'est pour cette raison que nous avons décidé de garder les angles calculés en se basant sur les valeurs de l'accéléromètre car elles ne sont pas sujettes à ce problème.

3. Le traitement des angles calculés

Les angles fraichement calculer doivent subir un dernier traitement avant l'envoie. En effet, ceux-ci sont compris entre -90° et 90°, hors les servomoteurs sont commandés par une consigne comprise entre 0° et 180°. L'Arduino possède la fonction *map* permettant de réaliser facilement ce calcul.

Un deuxième traitement a dû être réalisé lors de la fabrication du prototype. Le sens du capteur étant inversé par rapport à l'indication des axes sur la platine, pour faciliter le montage, il a fallu inverser les angles.

Chapitre 4 : Contrôles des servomoteurs

Pour reproduire le mouvement de la tête, nous utilisons trois servomoteurs représentant chacun une composante du mouvement. Comme expliqué dans les spécifications techniques du projet, les servomoteurs sont capables d'atteindre des positions prédéterminées grâce à l'utilisation d'un contrôleur à l'intérieur de celui-ci. Les servomoteurs utilisés pour notre prototype sont limités par une position allant de 0° à 180°. La tête humaine pouvant difficilement dépasser ces positions, cette limite n'est pas problématique.

1. Récupération des angles

Pour pouvoir commander les servomoteurs, l'Arduino reçoit sur son port série la chaine de caractère suivante en continue :

$$x<\Theta_{ZX}>y<\Theta_{ZY}>z<\Theta_{XY}>$$

L'utilisation des "<" et ">" permettent de séparer les angles des axes et évitent d'"écraser" les informations non traité de la chaine.
Les angles reçus peuvent directement être exploités par les servomoteurs.

2. Commande de servomoteurs

La commande du moteur est basée sur une librairie écrite en C++, permettant à travers l'instanciation d'un objet Servo de commander les servomoteurs à travers la génération d'une PWM (*Pulse Width Modulation)* avec les étapes suivantes:

- Association d'une broche de commande à l'objet Servo avec la méthode attach().
- Commande par la méthode write() à travers un angle donné.

Chapitre 5 : Acquisition des données caméra

A travers cette partie du projet nous avons essayé d'acquérir les données vidéo capturées par la caméra présente sur la tourelle mécanique et de l'envoyer via une communication sans fil vers l'utilisateur pour un affichage sur un écran.

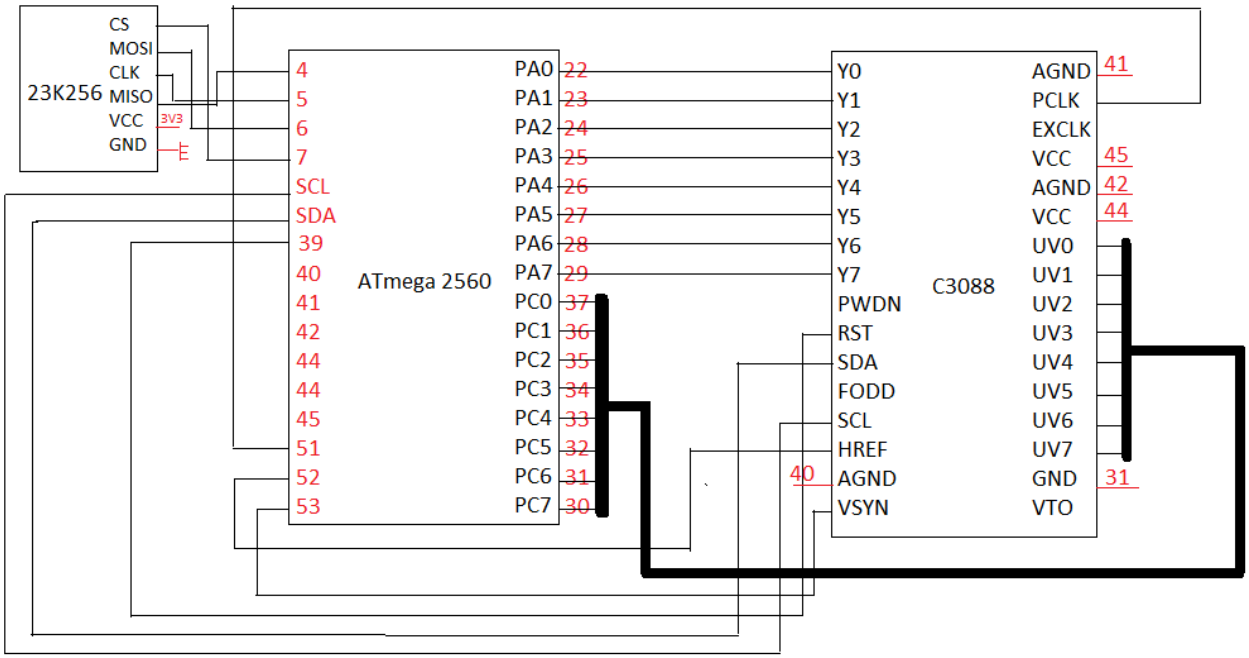


Schéma des connexions Caméra Arduino

1. Communication I2C

Pour configurer les registres internes de la caméra OV6620, la technologie utilisée est l'I2C (Protocole de communication développé par PHILIPS).
Ce protocole de communication utilise deux fils:

- SDA : bus de données
- SCL : bus d'horloge

De plus, ce protocole est basé sur une architecture Maître - esclave. Dans notre cas, le microcontrôleur est le maître (il fournit l'horloge) et la caméra est l'esclave.

1. 1. Vitesse de communication

Avant de commencer la communication, on initialise la vitesse de communication.
On a choisi de communiquer à une fréquence de 200 KHz.

1. 2. Écriture

Lors d'une écriture sur un registre les opérations suivantes sont effectuées:

1. Envoie de l'adresse de l'esclave en écriture (ici un seul).
2. Envoie de l'adresse de registre à modifier.
3. Envoie d'un octet de données.

1. 3. Lecture

Lors d'une lecture, on procède de la même façon:

1 Envoie sur le bus de l'adresse de l'esclave en lecture.
2 Envoie de l'adresse de registre à lire.
3 Lecture d'un octet de données.

2. Configuration de la caméra

Pour utiliser la caméra avec un microcontrôleur spécifique (ici ATMega 2560), il faut configurer les paramètres de la caméra.

En effet, nous avons réalisé les initialisations suivantes:

- RESET matériel : à travers la broche RESET de la caméra.
- RESET logiciel avant configuration des registres :
  Écriture sur le registre 0x12 avec la valeur 0x80.
- Mode QCIF:
  La camera peut fonctionner avec plusieurs formats de vidéo :
  CIF *Common Intermediate Format* 352 × 288
  QCIF *Quarter Common Intermediate Format* 176 × 144

  Par soucis de temps de calcul et d'espace mémoire nous avons utilisé la format QCIF.

- AGC *Automatic Gain Control*
  Le contrôle automatique de gain améliore la qualité de l'image.

- Format de données YCrCb
  YCbCr est une manière de représenter l'espace colorimétrique en vidéo.

$$Y = 0,299 \cdot R + 0,587 \cdot G + 0,114 \cdot B$$
$$Cb = -0,1687 \cdot R - 0,3313 \cdot G + 0,5 \cdot B + 128$$
$$Cr = 0,5 \cdot R - 0,4187 \cdot G - 0,0813 \cdot B + 128$$

  Ensuite pour passer à une représentation RGB, on utilise les équations suivantes:

$$R = Y + 1,402 \cdot (Cr - 128)$$
$$G = Y - 0,34414 \cdot (Cb - 128) - 0,71414 \cdot (Cr - 128)$$
$$B = Y + 1,772 \cdot (Cb - 128)$$

- Réduction de nombre d'image par seconde:
  Par soucis de fonctionnement sous microcontrôleur Atemega2560 qui tourne à une fréquence maximale de 16 MHz, nous avons passé le nombre d'image par seconde (frame rate per second) de 60 fps (frame per second) à seulement 5 fps (fonctionnement minimal avec un diviseur de fréquence maximal).

- Mode caméra Maître
  Utilisation de caméra en mode maître nous évite de fournir les signaux de synchronisation.

3. Acquisition des données:

Le signal vidéo est représenté sur 16 bits:

- Y : 8 bits
- Cr/Cb : 8 bits

La capture d'une image vidéo se fait par lecture pixel par pixel sur le port A (Y) et port C (CrCb).Les signaux de synchronisation sont les suivants:

- VSYNC : signal de synchronisation verticale permettant de signaler le début d'une image.
- HREF : signal de synchronisation horizontale permettant de signaler le début d'une ligne.
- PCLK : signal permettant de signaler un pixel à travers un front montant.

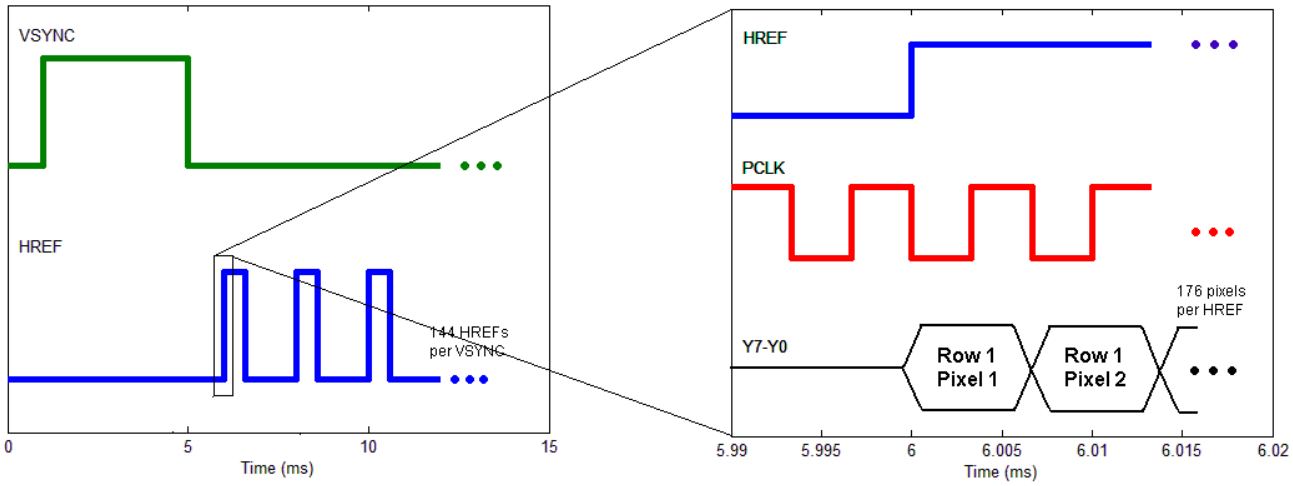


Schéma des signaux de synchronisation

Le déroulement de la capture se fait par lecture d'une ligne pixel par pixel pendant que le signal HREF est en position haut. Ensuite, on envoie la ligne sur le port série pour une transmission vers un ordinateur (phase de test) pendant que HREF passe en position bas.

Cette partie n'est pas achevée car on n'a pas arrivé à trouver l'origine de problème de synchronisation et on retrouve des images bruités.

4. Solutions alternatives

Chapitre 6 : Limites du système

Les tests unitaires ainsi que les tests d'intégration nous ont montré certaines limites de notre système électronique:

- Les servomoteurs sont commandables suivant une consigne variant entre 0° et 180°, dans le cas d'une rotation dépassant cet intervalle des valeurs, les servomoteurs se bloquent.
- Pour des valeurs d'inclinaison limites (tête penchée totalement en avant par exemple), l'accéléromètre n'est pas totalement stable et précis dans les valeurs données. Il risque alors d'y avoir quelques perturbations (servomoteur allant à 0° puis à 180° et de nouveau à 0°).

Chapitre 7 : Difficultés rencontrées

Durant ce projet, nous avons rencontré un certain nombre des difficultés:

- Une perte de temps au début de projet à cause de l'utilisation d'un simple accéléromètre pour détecter toutes les rotations alors qu'il est physiquement impossible de détecter un mouvement de lacet.
- Un problème de synchronisation lors de l'acquisition du signal vidéo à partir de la caméra. Malgré la diminution de la fréquence des images, l'optimisation des expressions d'affectation et lecture et l'utilisation d'un mémoire, nous n'avons pas arrivé à avoir des images correctes.
- On a rencontré aussi un problème d'incompatibilité entre librairies utilisées dans le même programme. Par exemple, la librairie SoftwareSerial.h n'est pas compatible avec la librairie Servo.h car elle bloque les timers de l'Arduino ce qui influe sur la stabilité des moteurs qui rentrent en oscillations qui peuvent l'endommager.

Conclusion

Ce projet d’apparence facile c’est retrouvé ralentit par la présence de petits problèmes aux niveaux des capteurs et de la vision. Malgré cela, un prototype fonctionnel de la tourelle a quand même pu être réalisé. En ajoutant la vision et remplaçant les servomoteurs, l’ensemble pourra être exploité avec une véritable tourelle.

Prototype de la tourelle (à droite) et du capteur (sur la casquette à gauche)

Annexes

Annexe 1 : Code partie Capteurs
Annexe 2 : Code partie tourelle
Annexe 3 : Code Caméra

Annexe 1 : Code partie Capteurs

```
#include <Wire.h>
#include <ITG3200.h>

#define ACC_ADDR 0x53
#define ACC_POWER_CTL 0x2D
#define ACC_DATA_FORMAT 0x31
#define ACC_DATA 0x32
#define ACC_NB_OCTETS 6

#define G0 9.812865328
#define toDegrees(x) (x * 57.2957795)

ITG3200 gyro = ITG3200();
float  x,y,z;
float timeStep = 0.2;          //200ms.
unsigned long timer;

int acc_data[3], acc_init[3];
float acc_g[3];

float pitchGyro = 0, rollGyro = 0, yawGyro = 0, pitchAcc = 0, rollAcc = 0;
byte pitchServo, rollServo, yawServo;

// Fonction permettant d'écrire une valeur à une adresse de registre donnée.
// Prend en paramétré l'adresse i2c de l'esclave, l'adresse sur registre que l'on doit lire et la valeur à copier.
void ecritureI2C(int device, byte address, byte val)
{
   Wire.beginTransmission(device);
   Wire.write(address);
   Wire.write(val);
   Wire.endTransmission();
}

// Fonction permettant de lire la valeur à une adresse de registre donnée.
// Prend en paramétré l'adresse i2c de l'esclave, l'adresse sur registre que l'on doit lire, le nombre de bit à
lire et le tableau de byte dans lequel on retourne le résultat.
void lectureI2C(int device, byte address, int num, byte buff[])
{
  Wire.beginTransmission(device);
  Wire.write(address);
  Wire.endTransmission();

  Wire.beginTransmission(device);
  Wire.requestFrom(device, num);

  int i = 0;
  while(Wire.available ())
  {
    buff[i] = Wire.read();
    i++;
```

```
  }
  Wire.endTransmission();
}

// Fonction d'initialisation de l'accéléromètre.
void initAcc()
{
  ecritureI2C(ACC_ADDR, ACC_DATA_FORMAT, 0x01); // Mode 4G
  ecritureI2C(ACC_ADDR, ACC_POWER_CTL, 0x08); // Mesure activé
}

// Fonction de lecture de l'accéléromètre sur le bus i2c
// Prend en paramétré le tableau data qu'il retourne
void lectureAcc(int * data)
{
  byte buff[ACC_NB_OCTETS];

  lectureI2C(ACC_ADDR, ACC_DATA, ACC_NB_OCTETS, buff);

  data[0] = ((int)(buff[1] << 8) | buff[0]) - acc_init[0];
  data[1] = ((int)(buff[3] << 8) | buff[2]) - acc_init[1];
  data[2] = ((int)(buff[5] << 8) | buff[4]) - acc_init[2];
}

// Fonction de calibration de l'accéléromètre
void accCalibration(void)
{
  int acc[3] = {0};
  do lectureAcc(acc_init);
  while (acc_init[0] == 0 && acc_init[1] == 0 && acc_init[2] == 0);
  for (int inc = 0; inc < 250; inc++)
  {
    do lectureAcc(acc_init);
    while (acc_init[0] == 0 && acc_init[1] == 0 && acc_init[2] == 0);
    acc[0] += acc_init[0];
    acc[1] += acc_init[1];
    acc[2] += acc_init[2];
  }
    acc_init[0] = abs(acc[0] / 250);
    acc_init[1] = abs(acc[1] / 250);
    acc_init[2] = abs(acc[2] / 250);
}

// Fonction ayant pour but de vérifier si la réponse reçu par le Xbee est bien un OK
// Renvoie 1 si la réponse est OK sinon 0
int XbeeOK()
{
  while(!Serial.available()){}; // On boucle tant que l'on a pas de réponse
  if(Serial.read() == 'O')
  {
    while(!Serial.available()){}; // On boucle tant que l'on a pas de réponse
    if(Serial.read() =='K')
```

```
    {
      while(!Serial.available()){}; // On boucle tant que l'on a pas de réponse
      if(Serial.read() == '\r')
      {
        return 1;
      }
    }
  }
  return 0;
}

// Fonction permettant l'exécution d'une commande pour le Xbee
// Prend en paramètre la commande, la valeur associé à la commande (s'il y a sinon -1; -1 par défaut) et si un retour chariot est nécessaire après la commande (sinon 0; 1 par défaut)
// Renvoie 1 si la commande a bien été effectué sinon 0
int commandeXbee(String aff, int val = -1, int retour = 1)
{
  Serial.print(aff);
  if(val >= 0) Serial.print(val);
  if(retour) Serial.println();
  if(!XbeeOK()) return 0; // Si la réponse n'est pas OK, on renvoie 0
  return 1;
}

// Fonction permettant de configurer le Xbee
// Prend en paramètre l'adresse source (ATMY), l'adresse de destination (ATDL, octet de poids faible) et l'identifiant du réseau (ATID)
// Renvoie 1 si la configuration a bien été effectué sinon 0
int configXbee(int atmy, int atdl, int atid)
{
  if(!commandeXbee("+++", -1, 0)) return 0; // On indique que l'on veut configurer le Xbee
  if(!commandeXbee("ATMY", atmy)) return 0; // Adresse source
  if(!commandeXbee("ATDL", atdl)) return 0; // Adresse de destination, octet de poids faible
  if(!commandeXbee("ATDH", 0)) return 0; // Adresse de destination, octet de poids fort (fixé)
  if(!commandeXbee("ATID", atid)) return 0; // Identifiant du réseau
  if(!commandeXbee("ATCN")) return 0; // Fin de configuration

  return 1;
}

void calibration() {
  accCalibration();
  gyro.zeroCalibrate(2500, 2);
}

void setup(void) {
  Serial.begin(9600);
  Wire.begin();

  delay(1000);

  // Configuration du Xbee
```

```
  while(!configXbee(11, 10, 3005)); // ATMY, ATDL, ATID

  initAcc();
  gyro.init(ITG3200_ADDR_AD0_LOW);

  calibration();

  //attachInterrupt(1, calibration, RISING);
}

void loop(void) {

    timer = millis();
    gyro.readGyro(&x,&y,&z);
    lectureAcc(acc_data);

    pitchAcc = atan2(acc_data[1], acc_data[2]);
    pitchAcc = toDegrees(pitchAcc);
    pitchServo = map(pitchAcc, -90, 90, 0, 180);
    if(pitchServo < 0) pitchServo = 0;
    else if(pitchServo > 180) pitchServo = 180;
    rollAcc = atan2(acc_data[0], acc_data[2]);
    rollAcc = toDegrees(rollAcc);
    rollServo = map(rollAcc, -90, 90, 0, 180);
    if(rollServo < 0) rollServo = 0;
    else if(rollServo > 180) rollServo = 180;
    if(rollServo < 90) rollServo = 90 + (90 - rollServo)  ;
    else if(rollServo > 90) rollServo = 90 - (rollServo - 90);

    yawGyro = yawGyro + ( z ) * timeStep;
    yawServo = map(yawGyro, -90, 90, 0, 180);
    if(yawServo < 0) yawServo = 0;
    else if(yawServo > 180) yawServo = 180;
    if(yawServo < 90) yawServo = 90 + (90 - yawServo)  ;
    else if(yawServo > 90) yawServo = 90 - (yawServo - 90);

    Serial.print("x<");
    Serial.print(pitchServo);
    Serial.print(">y<");
    Serial.print(rollServo);
    Serial.print(">z<");
    Serial.print(yawServo);
    Serial.println(">");

    timer = millis() - timer;
    timer = (timeStep * 1000) - timer;
    delay(timer); // On impose un temps d’exécution de 200ms
}
```

Annexe 2 : Code partie tourelle

```
#include <Servo.h>

Servo pitchServo, rollServo, yawServo;

float timeStep = 0.2;
unsigned long timer;

// Fonction ayant pour but de vérifier si la réponse reçu par le Xbee est bien un OK
// Renvoie 1 si la réponse est OK sinon 0
int XbeeOK()
{
  while(!Serial.available()){}; // On boucle tant que l'on a pas de réponse
  if(Serial.read() == 'O')
  {
    while(!Serial.available()){}; // On boucle tant que l'on a pas de réponse
    if(Serial.read() =='K')
    {
      while(!Serial.available()){}; // On boucle tant que l'on n'a pas de réponse
      if(Serial.read() == '\r')
      {
        return 1;
      }
    }
  }
  return 0;
}

// Fonction permettant l'exécution d'une commande pour le Xbee
// Prend en paramètre la commande, la valeur associé à la commande (s'il y a sinon -1; -1 par défaut) et si un retour chariot est nécessaire après la commande (sinon 0; 1 par défaut)
// Renvoie 1 si la commande a bien été effectué sinon 0
int commandeXbee(String aff, int val = -1, int retour = 1)
{
  Serial.print(aff);
  if(val >= 0) Serial.print(val);
  if(retour) Serial.println();
  if(!XbeeOK()) return 0; // Si la réponse n'est pas OK, on renvoie 0
  return 1;
}

// Fonction permettant de configurer le Xbee
// Prend en paramètre l'adresse source (ATMY), l'adresse de destination (ATDL, octet de poids faible) et l'identifiant du réseau (ATID)
// Renvoie 1 si la configuration a bien été effectué sinon 0
int configXbee(int atmy, int atdl, int atid)
{
  if(!commandeXbee("+++", -1, 0)) return 0; // On indique que l'on veut configurer le Xbee
  if(!commandeXbee("ATMY", atmy)) return 0; // Adresse source
  if(!commandeXbee("ATDL", atdl)) return 0; // Adresse de destination, octet de poids faible
  if(!commandeXbee("ATDH", 0)) return 0; // Adresse de destination, octet de poids fort (fixé)
```

```
  if(!commandeXbee("ATID", atid)) return 0; // Identifiant du réseau
  if(!commandeXbee("ATCN")) return 0; // Fin de configuration

  return 1;
}

// Fonction permettant de renvoyant l'angle associer à un axe et lu sur le port série
// Prend en paramètre l'axe sous la forme d'un caractère
// Retourne l'angle si pas de problème sinon -1
int lecture(char axe)
{
  char serialChar;
  int serialInt = 0;

  do
  {
    while(!Serial.available()){};
    serialChar = Serial.read();
  } while(serialChar != axe);
  while(!Serial.available()){};
  serialChar=Serial.read();
  if(serialChar!='<') return -1;
  while(serialChar != '>' )
  {
    if(serialChar >= '0' && serialChar <= '9') serialInt = (serialInt * 10) + (serialChar - '0');
    while(!Serial.available()){};
    serialChar=Serial.read();
  }
  return serialInt;
}


void setup(void) {
  Serial.begin(9600);

  delay(1000);
  pitchServo.attach(9);
  rollServo.attach(10);
  yawServo.attach(11);

  // Configuration du Xbee
  while(!configXbee(10, 11, 3005)); // ATMY, ATDL, ATID

  yawServo.write(0);
  delay(500);
  yawServo.write(180);
  delay(500);
  yawServo.write(90);
  delay(500);
}

void loop(void) {
```

```
  timer = millis();

  int pitch = lecture('x');
  int roll = lecture('y');
  int yaw = lecture('z');

  if(pitch >= 0 && pitch <= 180) pitchServo.write(pitch);
  if(roll >= 0 && roll <= 180) rollServo.write(roll);
  if(yaw >= 0 && yaw <= 180) yawServo.write(yaw);

  delay(10);
}
```

Annexe 3: Code Camera

```
#include <inttypes.h>
#include <compat/twi.h>
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <stdio.h>
#include <stdlib.h>
#include <avr/pgmspace.h>

#define Y PINA
#define UV PINC
#define VSYNC ((PINB>>0)&0x01)
#define HREF ((PINB>>1)&0x01)
#define PCLK ((PINB>>2)&0x01)

#define AGND1 40
#define AGND2 41
#define AGND3 42
#define CGND 43

#define VCC1 44
#define VCC2 45
#define RST 39


#define F_CPU 16000000UL //16 Mhz
#define SCL_CLOCK  200000UL //200 Khz
#define CAMW 0xC0
#define CAMR 0xC1


void i2c_init(void)
{
  /* initialize TWI clock: 100 kHz clock, TWPS = 0 => prescaler = 1 */
  TWSR = 0;    /* no prescaler */
  TWBR = ((F_CPU/SCL_CLOCK)-16)/2;  /* must be > 10 for stable operation */
  Serial.println(TWBR);
}/* i2c_init */

unsigned char i2c_start(unsigned char address)
{
        uint8_t   twst;
        // send START condition
        TWCR = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);
        // wait until transmission completed
        while(!(TWCR & (1<<TWINT)));
        // check value of TWI Status Register. Mask prescaler bits.
        twst = TW_STATUS & 0xF8;
        if ( (twst != TW_START) && (twst != TW_REP_START)) return 1;
        // send device address
        TWDR = address;
        TWCR = (1<<TWINT) | (1<<TWEN);
        // wail until transmission completed and ACK/NACK has been received
        while(!(TWCR & (1<<TWINT)));
        // check value of TWI Status Register. Mask prescaler bits.
        twst = TW_STATUS & 0xF8;
        if ( (twst != TW_MT_SLA_ACK) && (twst != TW_MR_SLA_ACK) ) return 1;
        return 0;
```

```
}/* i2c_start */


void i2c_start_wait(unsigned char address)
{
        uint8_t   twst;
        while ( 1 )
        {
        // send START condition
        TWCR = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN);
        // wait until transmission completed
        while(!(TWCR & (1<<TWINT)));
        // check value of TWI Status Register. Mask prescaler bits.
        twst = TW_STATUS & 0xF8;
        if ( (twst != TW_START) && (twst != TW_REP_START)) continue;
        // send device address
        TWDR = address;
        TWCR = (1<<TWINT) | (1<<TWEN);
        // wail until transmission completed
        while(!(TWCR & (1<<TWINT)));
        // check value of TWI Status Register. Mask prescaler bits.
        twst = TW_STATUS & 0xF8;
        if ( (twst == TW_MT_SLA_NACK )||(twst ==TW_MR_DATA_NACK) )
        {
        /* device busy, send stop condition to terminate write operation */
        TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWSTO);
        // wait until stop condition is executed and bus released
        while(TWCR & (1<<TWSTO));
        continue;
        }
        //if( twst != TW_MT_SLA_ACK) return 1;
        break;
        }
}/* i2c_start_wait */

unsigned char i2c_rep_start(unsigned char address)
{
        return i2c_start( address );
}/* i2c_rep_start */


void i2c_stop(void)
{
        /* send stop condition */
        TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWSTO);
        // wait until stop condition is executed and bus released
        while(TWCR & (1<<TWSTO));
}/* i2c_stop */


unsigned char i2c_write( unsigned char data )
{
        uint8_t   twst;
        // send data to the previously addressed device
        TWDR = data;
        TWCR = (1<<TWINT) | (1<<TWEN);
        // wait until transmission completed
        while(!(TWCR & (1<<TWINT)));
        // check value of TWI Status Register. Mask prescaler bits
```

```
        twst = TW_STATUS & 0xF8;
        if( twst != TW_MT_DATA_ACK) return 1;
        return 0;
}/* i2c_write */

unsigned char i2c_readAck(void)
{
        TWCR = (1<<TWINT) | (1<<TWEN) | (1<<TWEA);
        while(!(TWCR & (1<<TWINT)));
        return TWDR;
}/* i2c_readAck */


unsigned char i2c_readNak(void)
{
        TWCR = (1<<TWINT) | (1<<TWEN);
        while(!(TWCR & (1<<TWINT)));
        return TWDR;
}/* i2c_readNak */

unsigned char camera_read(unsigned char regNum)
{
        unsigned char f;
        unsigned char data;
        // send start condition and SLA+W
        f = i2c_start(CAMW);
        // tell camera what register and send stop condition
        f = i2c_write(regNum);
        i2c_stop();
        // send start condition and SLA+R
        f = i2c_start(CAMR);
        // read byte and send NAK
        data = i2c_readNak();
        // stop
        i2c_stop();
        return data;
}

unsigned char camera_write(unsigned char regNum, unsigned char data)
{
        unsigned char f;
        // send start condition and SLA+W
        f = i2c_start(CAMW);
        if (f) return f;
        // tell camera what register
        f = i2c_write(regNum);
        if (f) return f;
        // write data and stop
        f = i2c_write(data);
        if (f) return f;
        i2c_stop();
        return 0;
}

unsigned char camera_init(void)
{
        unsigned char f;
        f = camera_write(0x12,0x80); // soft reset
        if (f) return f;
```

```
        _delay_ms(2);
        f = camera_write(0x14,0x20); // QCIF frame size (176x144)
        if (f) return f;
        _delay_ms(2);
        f = camera_write(0x12,0x24); // (default setting) AGC enable, YCrCb mode, no AWB 0x24
        if (f) return f;
        _delay_ms(2);
        f = camera_write(0x13,0x01); // (default setting) 16 bit format (see datasheet)
        if (f) return f;
        _delay_ms(2);
        f = camera_write(0x11,0x0a); // reduce fps
        if (f) return f;
        _delay_ms(2);
        f = camera_write(0x29,0x00); // set to camera master mode
        if (f) return f;
        _delay_ms(2);
        return 0;
}


void takePicture()
{
        byte data_Y[176];
        byte data_U[176];
        byte data_V[176];
        int ordre=0;
        int indice=0;
        byte R,G,B;
        //int t1=millis();
        //while(VSYNC);
        //while(!VSYNC);
        // first rising edge here
        //while(VSYNC);
        while(!VSYNC);
        // second rising edge here
        while(VSYNC);
        for (int line=0; line<144; line++)
        {
        while (!HREF); // wait for HREF to go high
        for (int pix=0; pix<176; pix++)
        {
        while(!PCLK);

        data_Y[pix]=Y;

        if(ordre=0)
        {
        data_U[pix]=UV;
        ordre=1;
        }
        else
        {
        data_V[pix]=UV;ls

        ordre=0;
        }

        while (PCLK);
        }
```

```
          while(HREF);
          for(int i=0;i<176;i++)
          {
          R=data_Y[i] + 1,13983*data_V[i];
          G=data_Y[i] - 0,39465*data_U[i] - 0,58060*data_V[i];
          B=data_Y[i] + 2,03211*data_Y[i];
          Serial.write(R);
          Serial.write(G);
          Serial.write(B);
          }
  }
}


void setup()
{
  Serial.begin(115200);
  Serial.println("bienvenue");
  unsigned char f;
  pinMode(AGND1,OUTPUT);
  pinMode(AGND2,OUTPUT);
  pinMode(AGND3,OUTPUT);
  pinMode(CGND,OUTPUT);
  pinMode(VCC1,OUTPUT);
  pinMode(VCC2,OUTPUT);
  pinMode(RST,OUTPUT);

  digitalWrite(AGND1,LOW);
  digitalWrite(AGND2,LOW);

  digitalWrite(AGND3,LOW);
  digitalWrite(CGND,LOW);
  digitalWrite(VCC1,HIGH);
  digitalWrite(VCC2,HIGH);
  digitalWrite(VCC2,HIGH);

  digitalWrite(RST, HIGH);
  delay(300);
  digitalWrite(RST, LOW);

  i2c_init();
  f=camera_init();
  Serial.println(f);
}

void loop()
{
  takePicture();
  while(1);

}
```