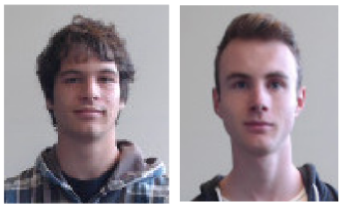
GROUPE N° 4
Tâche N° 2



Régné 28% Jagline 72%

Intégration STM32F411

1. Etat initial de la tâche :

(Jagline) Au démarrage du semestre nous ne disposons d'aucun code de base en stm32. La seule source que nous avons est le code Arduino du semestre précédent (ne fonctionnant pas vraiment).

2. Objectifs :

- Compréhension et intégration des codes pour comparer le fonctionnement à l'arduino

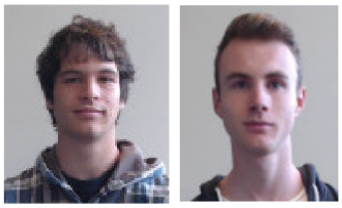
- Amélioration du code, optimisation

3. Démarche de travail/plannification/problèmes rencontrés

Problèmes rencontrés	Solution supposée / apportée
Code Arduino ne fonctionnant pas	Adaptation et optimisation avant de démarrer la Stm32
Pas de code de base	Adaptation du code Arduino obtenu
Nombreux problèmes avec la DMA, l'ADC	Changement des timers, choix de ne plus utiliser le potentiomètre
Réalisation de la PWM	Changement de timer car celui utilisé rentrait en conflit avec les l'ADC
Problème de transmission des données dans le sens PC -> carte	

4. Présentation des résultats obtenus

GROUPE N° 4
Tâche N° 2



Régné 28% Jagline 72%

Intégration STM32F411

Voici la configuration des pins:

Accéléromètre	PA0	ADC1_INO	DMA2_Stream0
IN4 hasheur	PA9	GPIO_OUTPUT	
IN3 hasheur	PA8	GPIO_OUTPUT	
Electro-aimant	PA11	GPIO_OUTPUT	
KAS	PA12	GPIO_EXTI12	
A_Vout	PB6	TIM4_CH1	Encoder Mode
B_Vout	PB7	TIM4_CH2	Encoder Mode
Timer fonctionnement moteur		TIM5	1Hz de base
Timer ADC		TIM9	200Hz
Timer asservissement		TIM10	50Hz
PWM hacheur	PB9	TIM11_CH1	PWM Generation CH1

L'horloge générale de la carte fonctionne à 72MHz.

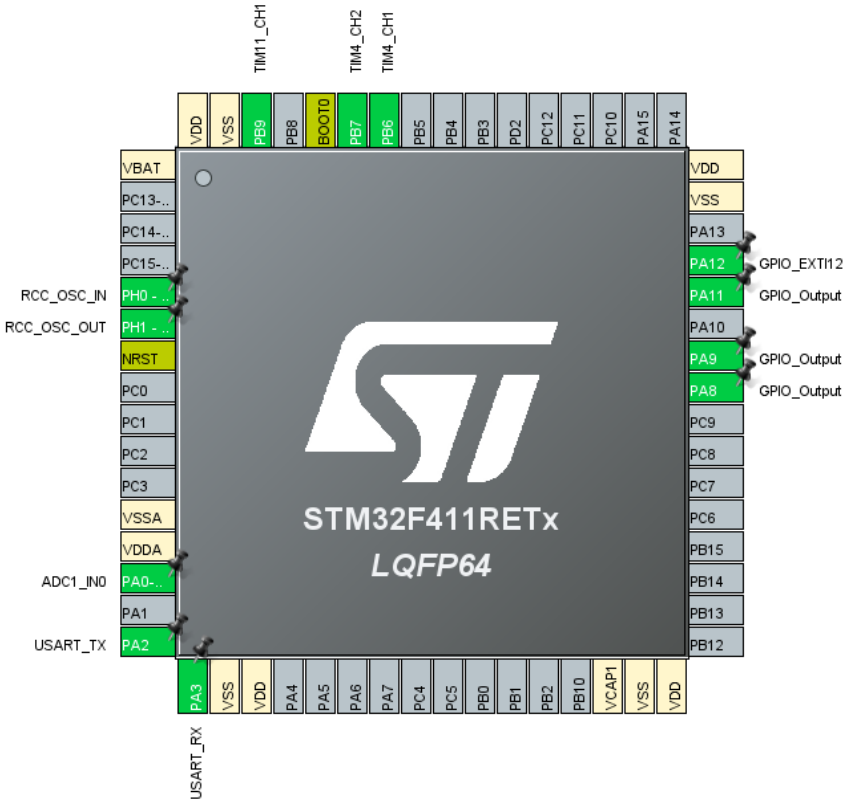
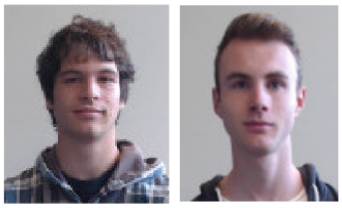


Figure 1: Pins utilisées sur la STM32F411RE

GROUPE N° 4
Tâche N° 2



Intégration STM32F411

Régné 28% Jagline 72%

Nous avons commencé par tenter de traduire le code Arduino qui était terminé pour que celui-ci fonctionne sous STM32. Ceci n'a pas été une bonne idée car nous avons rencontré de trop nombreux problèmes auxquels nous ne nous attendions pas. Leur résolution à donc été très fastidieuse. Nous avons donc repris chaque fonction une par une en commençant par l'asservissement du moteur. De plus, sous Arduino nous fonctionnions avec une librairie externe pour gérer l'asservissement.

Comptage tours moteurs:

Nous avions mis en place une première solution de comptage de tours moteurs avec une génération d'interruptions lors d'un front du moteur. Cette solution fonctionnait jusqu'à ce qu'on mette en place l'écoute puis écriture sur le port série selon des commandes, cela faisait beaucoup trop d'interruptions à prendre en compte en boucle en plus de l'écriture et le programme plantait.

Nous avons donc dû changer la méthode de comptage de ticks du moteur et avons pour cela utilisé le mode encoder du timer 4. En effet, il n'y a plus d' interruptions générées constamment, c'est le timer qui prend en charge les ticks moteurs reçus par les pins PB6 et PB7. Nous avons alors plus qu'à récupérer sa valeur avec:

```
int tick=TIM4->CNT;
```

Asservissement:

Pour cet asservissement nous avons mis en place un timer pour échantillonner la fonction. Nous avons un timer fonctionnant en interruption à une fréquence de 50Hz. Sa fonction de callback appelle donc notre fonction d'asservissement:

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef* htim){

    if (htim-> Instance == TIM10){
        asservissement();
    }
```

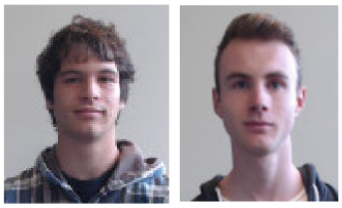
Pour calculer les paramètres du timer nous avons effectué le calcul:

Preload=$(T_{out} * F_{clk})/prescaler$

Nous avons Fclk=72 000 000Hz, Tout=1/50=0.02s et nous fixons prescaler=72Hz. Soit Auto-reload=19 999 (20 000-1 car nous comptons à partir de 0).

Nous avons ajouté une interruption sur un pin pour enregistrer les changements d'états de l'encodeur et nous avons finalement utilisé un dernier timer pour générer une PWM qui permet le contrôle en tension du moteur.

GROUPE N° 4
Tâche N° 2



Régné 28% Jagline 72%

Intégration STM32F411

PWM Moteur

Pour commander le moteur nous avons mis en place une génération de PWM (modulation de largeur d'impulsion) sur le Timer 11 consistant en générer une succession d'états hauts ou bas permettant de synthétiser une valeur moyenne de tension simulant un signal analogique.

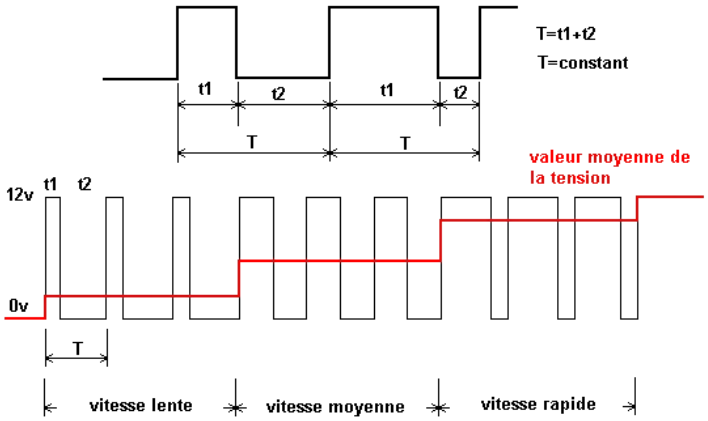


fig2: principe du PWM

Figure 2: Principe de la PWM

Cette commande est envoyée au moteur contrôlant sa vitesse par le biais du timer 11.

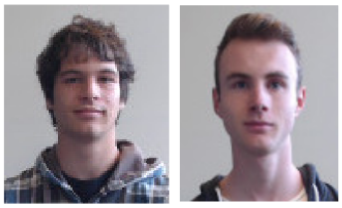
En effet, nous allons écrire la valeur de commande directement dans le registre de capture/compare du timer ce qui permet de contrôler le rapport cyclique du signal généré.

```
//envoie dans le moteur
TIM11->CCR1=(int)(cmd_tour_seconde*((TIM11->ARR))/tourMax);
```

La génération de PWM a été assez compliquée à mettre en place indirectement car générer une PWM seule fonctionnait mais la coupler avec un ADC en lecture de potentiomètre ne faisait plus rien fonctionner correctement. Ce n'est qu'après de nombreuses heures que nous avons vu que c'était une incompatibilité entre l'ADC et les timers utilisés comme nous allons le préciser plus tard pour la lecture de l'accéléromètre.

Nous avons finalement abandonné la lecture d'un potentiomètre physique pour laisser son contrôle à l'IHM.

GROUPE N° 4
Tâche N° 2



Intégration STM32F411

Régné 28% Jagline 72%

Lecture des valeurs de l'accéléromètre:

Pour lire les valeurs de l'accéléromètre et pouvoir plus tard les envoyer sur le port série nous avons mis en place un ADC en DMA.

Cela permet de convertir une valeur analogique en valeur binaire digital utilisable par la carte. Passer par le mode DMA (Direct Access Memory) (l'ADC peut aussi fonctionner en mode Interruption ou Polling) permet de ne pas passer par le CPU mais d'envoyer les données reçues directement en mémoire, ce qui ne surcharge pas le processeur :

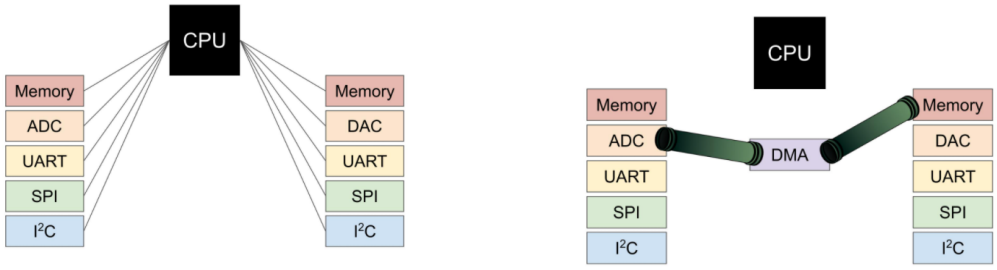


Figure 3: fonctionnement DMA

L'ADC décode en boucle les valeurs de l'accéléromètre et les envoie en mémoire directement lorsqu'il termine une conversion en appelant sa callback:

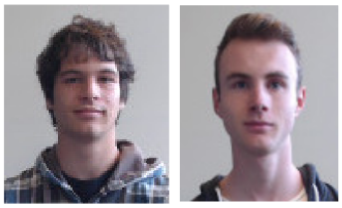
```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
    accelerometerValue = adcResultsDMA[0];
    temps+=1000/300; //300Hz de frequence, envoi en ms
    if (ecriture) {
        snprintf(txBufSend, txBufSendLEN, "%d,%d\r\n", accelerometerValue,(int)temps);
        HAL_UART_Transmit(&huart2, (uint8_t *)txBufSend, strlen(txBufSend), HAL_MAX_DELAY);
    }
}
```

Nous avons rencontré de nombreux problèmes avec l'ADC DMA et les timers. En effet, il ne fonctionnait jamais comme il fallait malgré les très nombreux tutoriels regardés et tests effectués.

Nous avons passé des heures à tenter de comprendre et résoudre les problèmes jusqu'à ce qu'un collègue nous indique avoir eu plusieurs problèmes également avec les timers. Nous avons conclu que les problèmes viennent principalement de conflits entre certains Timer, l'ADC et leurs utilisations, cependant aucunes informations de la part du logiciel nous ont été données et cela à rendu la résolution du problème très complexe.

Et effectivement en changeant nos timers 2, 3 et 3 par les 9, 10 et 11 pour l'ADC, l'asservissement et la PWM moteur, nous avons enfin eu une conversion ADC fonctionnelle.

GROUPE N° 4
Tâche N° 2



Régné 28% Jagline 72%

Intégration STM32F411

Nous avons donc pu confirmer la lecture de l'accéléromètre, la transmission des données et son échantillonnage fixe.

La communication avec l'IHM:

L'étape finale est de permettre de paramétrer le code directement à partir de l'IHM réalisé par Quentin, pour cela nous avons dû mettre en place un protocole de communication avec un message bien précis qui nous permet de savoir ce que la carte doit recevoir et comment l'analyser. En effet pour pouvoir recevoir le message et l'analyser sur la carte SMT32, nous devions fixer un taille en caractères.

Mode	Séparateur	Temps de fonctionnement			Séparateur	Choix des vitesses à réaliser en Oscillation forcée (ignoré en Oscillation libre)																
2	**,**	**0**	**1**	**5**	**,**	**1**	**-**	**0**	**-**	**0**	**-**	**1**	**-**	**0**	**-**	**1**	**-**	**0**	**-**	**0**	**-**	**1**	**-**	**0**

Figure 4, Représentation du message série

Le message fait donc 25 caractères quel que soit le mode de fonctionnement demandé par l'IHM. La carte écoute régulièrement le port série ce qui permet de couper le code ou de changer de mode si l'IHM le demande . Une fois reçu, un fonction (SplitDatas) décompose le message en trois morceaux (repéré grâce aux ,).

Le premier morceau correspond au mode de fonctionnement demandé par l'IHM, les valeurs vont de 0 à 3 avec 0 représentant l'alternance d'activation de l'électro-aimant, 1 le mode en oscillation libre, 2 le mode en oscillation forcée, 3/autre l'arrêt du code.

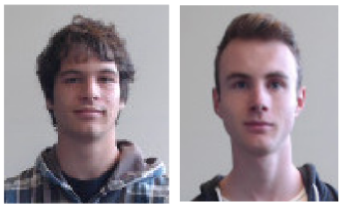
Le deuxième morceau concerne le temps d'activation du moteur suivant pour les différentes vitesses demandées pour le mode 2 en oscillation forcée.

Le troisième morceau permet au code concernant le mode 2 de déterminer quelles sont les vitesses à réaliser. Les 1 indiquant que cette vitesse doit être générée et les 0 indiquant qu'elle doit être ignorée. Les vitesses sont enregistrées dans un tableau (vitesseBase) dans le code STM32 et ce tableau est multiplié par le tableau de 1 et de 0.

```
float vitesseBase[10]={0.25,0.5,0.75,1.,1.25,1.5,1.75,2.,2.25,2.5};
```

Nous avons mis en place 2 fonctions de traitement des données reçues: SplitDatas() et Traitement() qui sont appelées lorsque le port série reçoit une donnée:

GROUPE N° 4
Tâche N° 2



Régné 28% Jagline 72%

Intégration STM32F411

```
//sépare les données reçues en données utilisables
//elles sont reçues selon le format : mode,temps,vitesse1-vitesse2-vitesse3-vitesse4-vitesse5-vitesse6-vitesse7-
//exemple pour lancer le moteur en oscillations forcées une vitesse sur deux pendant 5 s:
//2,005,0-1-0-1-0-1-0-1-0-1
void SplitDatas(){
    char *token =strtok(txBufReceive,","); //sépare selon ,
    int i=0;
    while (token){
        strcpy(commande[i],token);
        i++;
        token=strtok(NULL,",");

    }

    char *token2 =strtok(commande[2],"-"); //sépare selon -
    int j=0;
    while (token2){
        strcpy(vitesseTemp[j],token2);
        j++;
        token2=strtok(NULL,"-");
    }

    mode=commande[0][0]-'0';

    int tm[3]; //concatène les 3 caractères du temps reçu en un entier
    for (int i=0; i<3;i++){
        tm[i]=commande[1][i]-'0';
    }
    tempsMoteur=tm[0]*100+tm[1]*10+tm[2];

    for (int i=0;i<10;i++){
        vitessesRecues[i]=vitesseTemp[i][0]-'0';
    }


    for (int i=0;i<10;i++){ //envoie les vitesses commandées dans un tableau
        vitesse[i]=vitessesRecues[i]*vitesseBase[i];
    }
    consigne_moteur_nombre_tours_par_seconde=vitesse[rang];
}
```

SplitDatas() sert à séparer le message reçu selon les morceaux vus précédemment.

Pour cela la fonction strtok() permet de séparer une chaîne de caractères en plusieurs selon un séparateur, ici “,” pour les 3 blocs principaux mode, temps et vitesses.

Nous refaisons ensuite une séparation du 3ème bloc vitesses avec “-” pour récupérer chaque consigne de vitesse individuellement et les mettre dans un tableau.

Prenons l’exemple précédent, nous nous retrouvons donc avec:

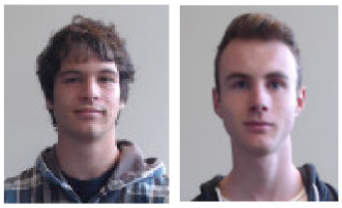
int mode=2;

int tempsMoteur=015;

float vitesse[10] ={1,0,0,1,0,1,0,0,1,0};

Ensuite nous appelons la fonction de traitement des données.

GROUPE N° 4
Tâche N° 2



Régné 28% Jagline 72%

Intégration STM32F411

```
//analyse et traite les données reçues suite à leur séparation
void Traitement(){
    ecriture=false;
    //electroaimant
    if (mode==0){
        if (imode==1){
            HAL_GPIO_WritePin(GPIOA,GPIO_PIN_11, GPIO_PIN_SET); //activer electroaimant
            imode=-1;
        }
        else if (imode==-1){
            HAL_GPIO_WritePin(GPIOA,GPIO_PIN_11, GPIO_PIN_RESET); //liberer electroaimant
            imode=1;
        }
    }
    //oscillation libre
    else if (mode==1){
        HAL_GPIO_WritePin(GPIOA,GPIO_PIN_11, GPIO_PIN_RESET); //liberer electroaimant
        consigne_moteur_nombre_tours_par_seconde=0.0;
        somme_erreur=0.0;
        ecriture=true;
    }
    //oscillation forcee
    else if (mode==2){
        TIM5->ARR=(TIM5->ARR+1)*tempsMoteur-1; //lancer le timer selon le temps demandé par l'IHM
        if (HAL_TIM_Base_Start_IT(&htim5)!=HAL_OK){
            Error_Handler();
        }
        ecriture=true;

    }
}
```

Traitement analyse le mode reçu et agit en conséquence.

- Si c'est le mode 0, il va activer ou désactiver l'électroaimant.
- Si c'est le mode 1 d'oscillations libres, il va libérer l'électroaimant et envoyer les données de l'accéléromètre en continu.
- Si c'est le mode 2 d'oscillations forcées, il va récupérer le tempsMoteur et l'envoyer au timer 5 qui génère des interruptions toutes les secondes de base. En les multipliant on retrouve donc des interruptions selon le temps indiqué dans la trame:

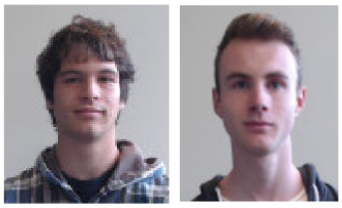
```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef* htim){

    if (htim->Instance == TIM5){
        ecriture=false;
        if (rang<10){
            while(vitesse[rang]==0){
                rang++;
            }
            ecriture=true;
            consigne_moteur_nombre_tours_par_seconde=vitesse[rang];
            somme_erreur=0.;
            rang++;
        }
        else{
            ecriture=false;
            somme_erreur=0.;
            consigne_moteur_nombre_tours_par_seconde=0;
        }
    }
```

Lorsqu'une interruption est générée, on regarde si on est censé tourner le moteur ou non dans le tableau traité précédemment et à quelle vitesse et on envoie les données sur le port série jusqu'à la prochaine interruption.

- Si c'est un autre chiffre, il va arrêter l'envoi de données et stopper le moteur.

GROUPE N° 4
Tâche N° 2



Intégration STM32F411

Régné 28% Jagline 72%

5. Bilan des résultats par rapport aux objectifs fixés

Nous avons donc répondu à l'objectif principal qui était de retrouver le fonctionnement Arduino sur la carte STM32F411re.

Cependant, ça n'a pas été simple du tout. Nous avons tout le temps rencontré des problèmes, rien n'a bien fonctionné du premier coup. Et nous avons dû passer énormément d'heures à débugger, corriger, trouver d'où ça venait etc.

Que ce soit la PWM qui ne fonctionnait que seule, l'ADC qui avait de très nombreux problèmes de compatibilité avec les différents timers, tous les timers qu'il a fallu changer, modifier le comptage de tours moteurs qui faisait planter le code, gérer la lecture et l'écriture en continu en faisant attention que le code ne se bloque pas dans une boucle d'écriture par exemple…

Nous avons malgré tout atteint notre objectif supplémentaire qui était de communiquer des paramètres de l'IHM vers la carte STM pour ainsi contrôler le fonctionnement du système grâce à l'IHM, notamment la commande de l'électroaimant et le mode oscillations forcées permettant de faire tourner le moteur selon les vitesses cochées et le temps choisi sur l'IHM qui fonctionnent très bien.

6. Perspectives d'évolution

Les évolutions portent principalement sur la résolution des problèmes de communication série entre l'IHM et la carte. Nous avons principalement un problème sur l'écoute régulière du port série.

L'asservissement du moteur peut probablement être amélioré par une détermination plus précise des coefficients du PID.

La prise en compte de KAS doit être mise en fonctionnement. Une amélioration des possibilités de contrôle du système via l'IHM peut être envisagée.