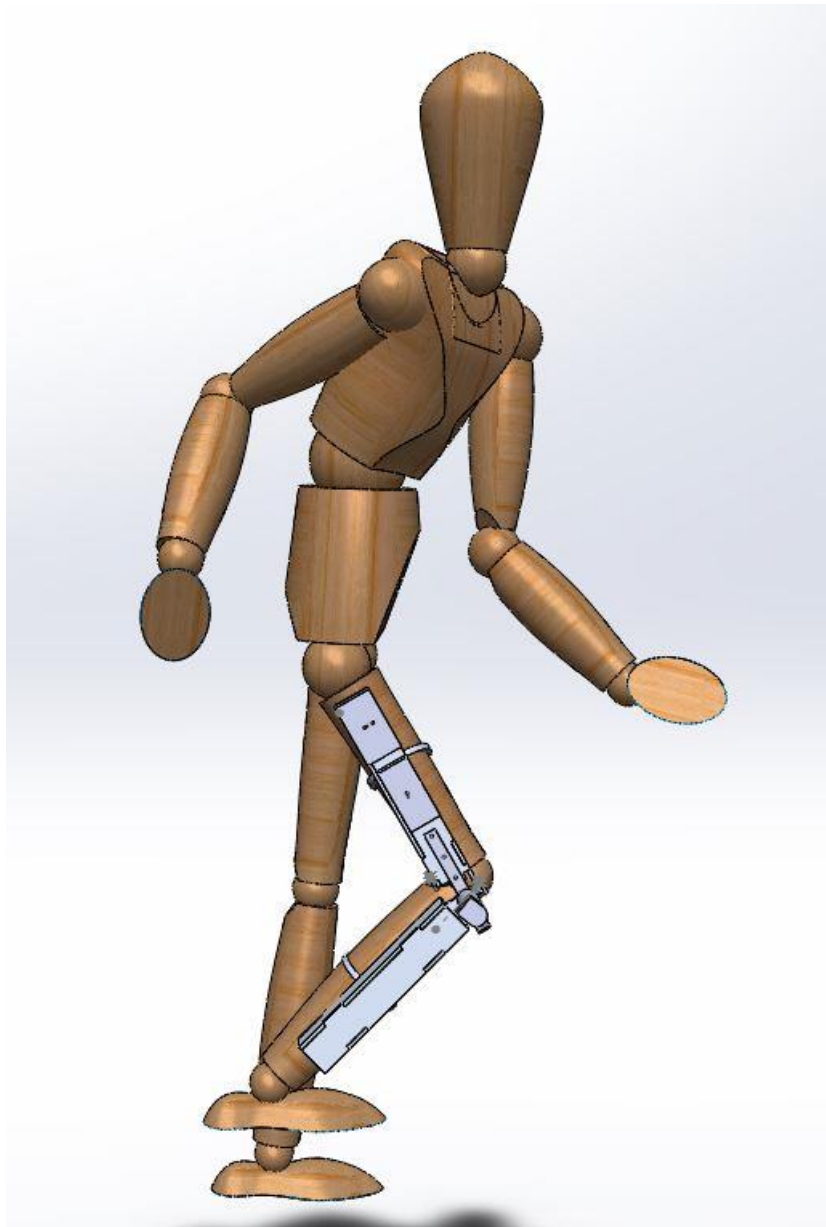


Projet Genou Héraclès



ESME Sudria

Projets Ingé2 2018-2019

Timothée Fréville, Daria de Tinguy, Louis Lolivier

Sujet du projet : Projet Genou Héraclès

Encadrant : Alex Caldas

COMPTE RENDU :

Table des matières

Projet Genou Héraclès	1
Introduction	3
Cahier Des Charges	6
I. Mécanique	9
A) Moteur	9
B) Structure	10
II. Circuit Electrique	15
A) Alimentation 12V	16
B) Alimentation 5V	16
C) Alimentation 3,3V	17
D) Alimentation du capteur EMG	18
III. Commande moteur	19
A) Driver	19
B) PWM	22
IV. Capteurs	27
A) Capteur EMG	27
B) Convertisseur analogique-numérique (ADC)	29
C) Capteur de Fin de Course	35
D) L'accéléromètre	37
V. Calcul du mouvement	40
ANNEXES	42
Code :	43

Introduction

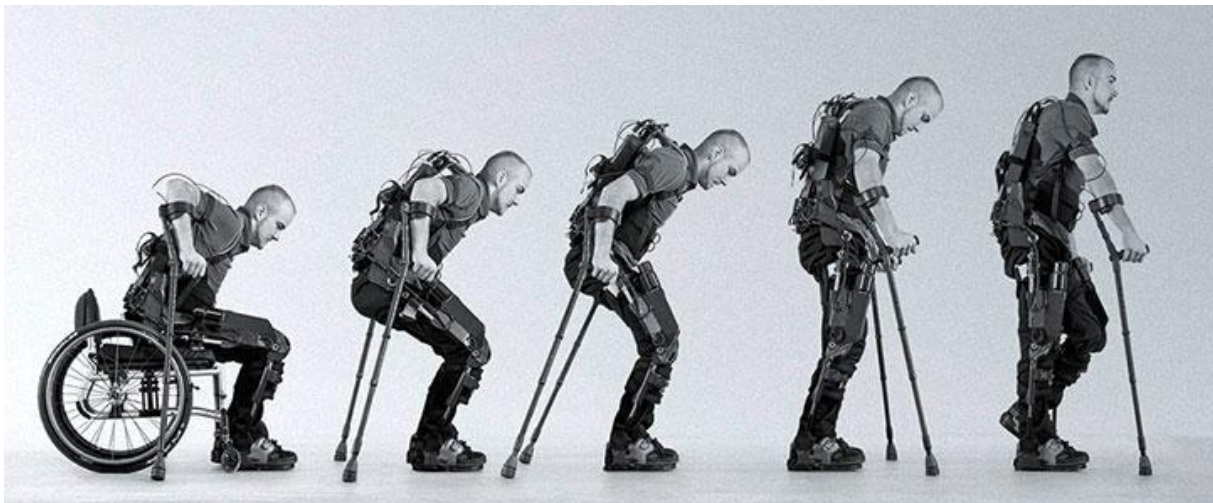


FIGURE 1 : EXEMPLE D'EXOSQUELETTE PERMETTANT A UNE PERSONNE PARALYSE DE MARCHER A NOUVEAU

L'idée même d'une prothèse mécanique pour remplacer un membre n'est pas si récente que cela. D'un certain point de vu les armures du moyen-âge étaient déjà des améliorations de nos capacités physiques. Tous les outils utilisés par l'humanité sont des extensions de ces membres donc l'idée de vouloir s'affranchir de certaines contraintes physiques est aussi vieille que l'invention du premier outil par l'Homme.



FIGURE 2 : ARMURE DE CHEVALIER DU MOYEN-AGE

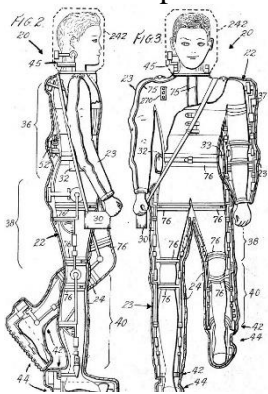


FIGURE 3: PLAN DE L'EXOSQUELETTE DE KULTSAR

En 1964 l'américain Emery Kultsar design l'un des premiers exosquelettes. Son objectif est de protéger les travailleurs des conditions extrêmes comme le feu, les explosions, les chutes de pierre... cette machine devait permettre en théorie de pouvoir ramener son porteur même inconscient dans un endroit dénué de danger. Le Brevet de cette machine a été déposé dans les années 60 mais elle était complètement irréalisable pour l'époque.

Comme on peut le voir dans certains de nos films actuels tel que Blade Runner ou dans les livres d'anticipation comme ceux d'Issac Asimov. Quelle soit prothèse ou augmentation humaine l'idée d'un corps augmenté a traversé les siècles. Aujourd'hui plus que jamais les thématiques d'augmentation et de transhumanisme apportées par notre technologie sont au cœur de nos sociétés. C'est dans ce contexte que notre projet d'exosquelette s'inscrit.

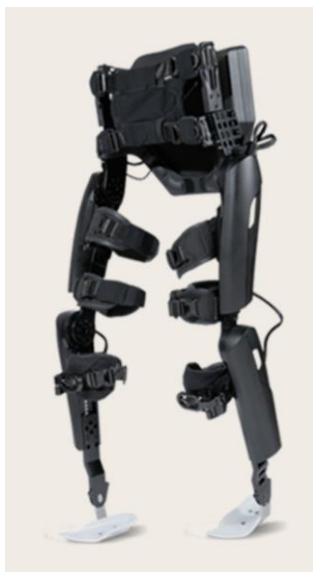


FIGURE 4: AFFICHE DU FILM BLADE RUNNER

Le projet Héraclès est né de l'idée de réaliser une version améliorée du projet Hercule qui nous a été présenté lors des journées portes ouvertes et des salons par l'ESME avant même notre admission. Ce projet étant très complexe, il nous a été conseillé de nous concentrer uniquement sur une seule des trois articulations d'une jambe. Nous avons donc décidé d'orienter notre projet vers la réalisation d'un exo-genou. Une dernière question subsistait, le choix de la commande du système. Dans un premier temps nous avons pensé à une commande mécanique sous la forme d'une télécommande ou d'une détection d'un mouvement de la jambe. Puis nous nous sommes finalement orientés vers un capteur électromyogramme qui permet de détecter plus rapidement si l'utilisateur veut se déplacer.

Avant d'en arriver jusque-là, nous nous sommes renseignés sur les projets similaires qui ont déjà été réalisés. Un certain nombre d'entre eux ont pour but d'assister les personnes à mobilité réduite ce qui ne correspondait pas à ce que nous avions l'intention de faire mais dont certaines fonctionnalités se sont révélées intéressantes pour nous.

Deux projets de ce type nous ont intéressés, le premier, REWALK est un projet industriel pensé pour des individus ne pouvant plus du tout marcher et initie la marche en détectant l'inclinaison du haut du corps. Le manque d'information à propos de cet exosquelette nous a orienté vers un autre projet qui a pour but d'aider les personnes à mobilité réduite : ALICE. Le gros avantage de ce projet est qu'il est open source, cela nous a permis de nous donner une idée plus précise des méthodes qui pourraient être utilisées pour réaliser notre projet.



FIGURES 4 ET 5. DE GAUCHE A DROITE, LE PROJET REWALK ET LE PROJET ALICE



FIGURE 6 ET 7. DE GAUCHE A DROITE, LE PROJET HERCULE ET LE PROJET XOS 2

Cahier Des Charges

Contexte :

Dans le cadre de ce qui fut un projet inter-majeur, nous pensons développer un exosquelette nommé Héraclès.

Héraclès serait une paire de jambes d'exosquelette comparables à celles du projet Hercule. Les jambes doivent suivre le mouvement de l'utilisateur et alléger ses charges. L'exosquelette complet doit pouvoir soutenir une charge de 100kg. Dans l'optique de ce projet nous allons construire un genou de l'exosquelette.

Problématique :

Le genou d'Héraclès doit donc pouvoir soutenir une charge de 4kg, détecter le mouvement du muscle de la cuisse de l'utilisateur et ainsi suivre/copier les mouvements naturels de la jambe (pression minimum sur la jambe humaine).

Durant le fonctionnement :

Fonction de service	Fonctions techniques	Critères	Niveaux	Caractéristiques	Flexibilité
Capacité de mouvement	Rapidité de détection du mouvement	Faible latence de détection du mouvement	3x Electrodes musculaire	Détection d'une contraction musculaire amplifié	300mV de bruit max
	Ergonomie	Accompagnement du mouvement	Moteur au genou	12V 7.948Nm 18.02tr/min	Quelques ms Quelques mm
	Renfort		4kg		500g

Assurer les charges		Capacité de charge supplémentaire sans efforts additionnel		Compris dans les caractéristiques moteur	
	Support	Poids ressenti sur les membres postérieurs	1kg		0.5KG
Alimentation	Autonomie	Durée de fonctionnement à vide	1h	12V 7A 60Ah	30min
	Réutilisabilité	Rechargeable		Batterie Lithium	300fois
Protéger l'utilisateur	Protéger contre des erreurs de programmes	Bouton stop *	Arrêt mécanique (coupe la liaison contrôleur/moteur)	Bouton poussoir	Latence nulle
	Protéger contre le dysfonctionnement	Angles conformes aux articulations humaines		Bloquer aux angles maximums du genou : 0°-120°	5° près
Utilisation des Commandes	Ergonomie	Simplicité d'accès aux commandes	Arrêts accessibles avec une longueur de bras		20cm près
Être confortable	Nuisance sonore	Bruit acceptable durant le fonctionnement	48dB	Vitesse limitée à 18tr/min Fréquence supérieure à 20kHz	2db
	Position agréable	. Rigueur des supports . Forme des supports	Sangles à scratch		

* Point d'amélioration : Le moteur garde la dernière position en mémoire et retourne au repos progressivement

Schéma synoptique :

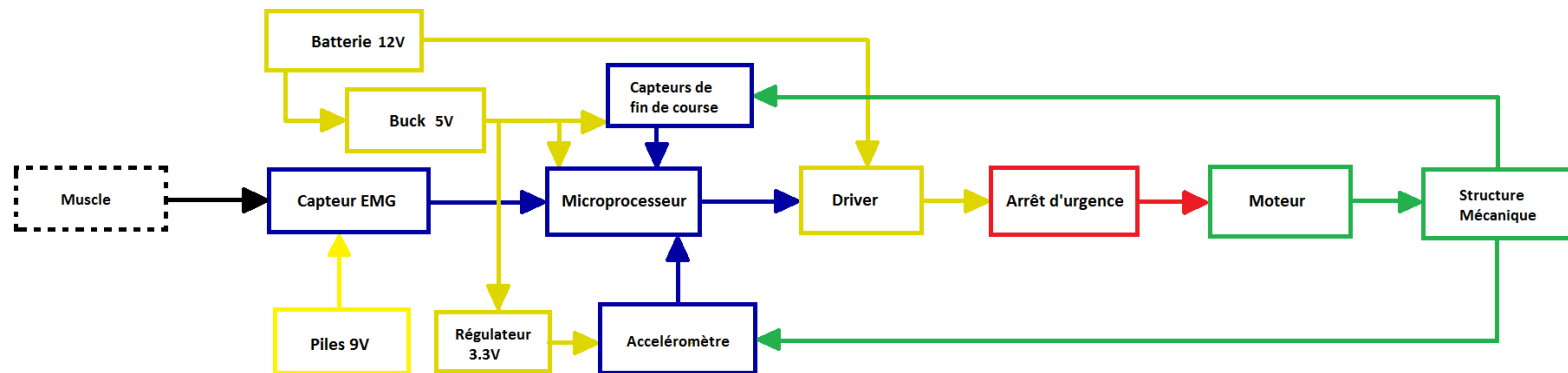


FIGURE 8 : SCHEMA SYNOPTIQUE DE LA PARTIE GENOUX D'HERACLES

Schéma fonctionnel :

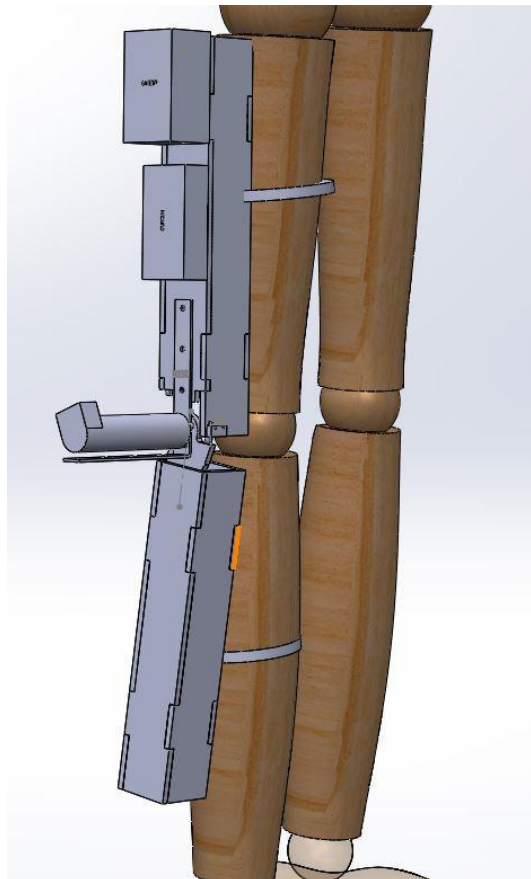


FIGURE 9 : SCHEMA FONCTIONNEL DE LA PARTIE GENOUX D'HERACLES

I. Mécanique

A) Moteur

Nous avons beaucoup revu cette partie-ci aux vues des contraintes (monétaires) du projet. [L'annexe mécanique](#) comprend notre premier cahier des charges, le moteur, la batterie et le réducteur que nous comptons utiliser avec le premier SolidWorks de l'ensemble. Tous ces éléments ont été revus pour s'adapter au moteur référence : « Maxon DC motor 2332.966-56.236-200 » fourni par l'école avec un réducteur de 288 :1.

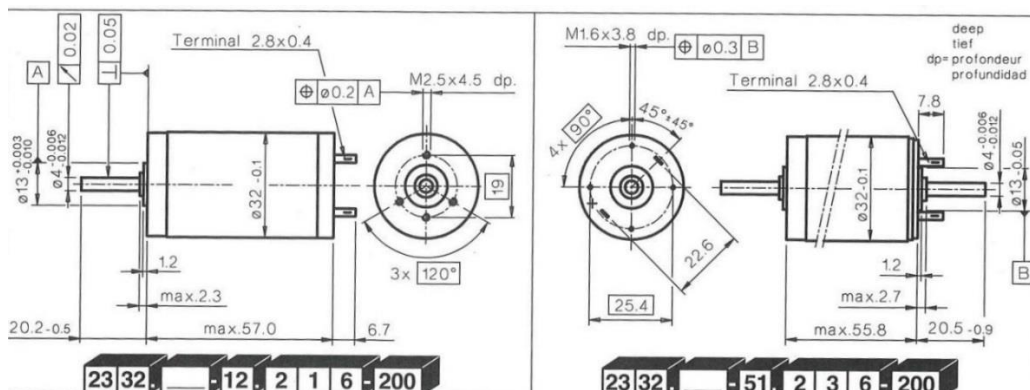


FIGURE 10: SCHEMA DU MOTEUR

Caractéristiques moteur	N° de bobinage (N° de commande)	966
1 Puissance conseillée	W	15
2 Tension nominale	Volt	12,00
3 Vitesse à vide	tr/min	4750
4 Couple de démarrage	mNm	86,9
5 Pente vitesse/couple	tr/min/mNm	57,3
6 Courant à vide	mA	34,4
7 Courant de démarrage	mA	3780
8 Résistance aux bornes	Ohm	3,18
9 Vitesse limite	tr/min	9200
10 Courant permanent max.	mA	1200
11 Couple permanent max.	mNm	27,60
12 Puissance max. fournie à la tens. nom.	mW	10200
13 Rendement max.	%	77,0
14 Constante de couple	mNm/A	23,0
15 Constante de vitesse	tr/min/V	415
16 Constante de temps mécanique	ms	14,6
17 Inertie du rotor	gcm ²	24,3
18 Inductivité	mH	0,53
19 Résistance therm. carcasse/ambient	K/W	12,50
20 Résistance therm. rotor/carcasse	K/W	1,90

FIGURE 11 : CARACTERISTIQUES DU MOTEUR

Le moteur a ainsi un couple de 7.948Nm avec le réducteur intégré, une vitesse de 18.02tr/min et une tension nominale de 12V.

Le moteur peut soutenir jusqu'à 4 Kg à bout d'un bras de 40cm.

B) Structure

Nous pensions tout d'abord à une armature en métal pour l'exosquelette. Nous n'avons pas trouvé de métal usinable et adaptable à notre usage. Nous avons donc choisi un squelette en bois de 6mm d'épaisseur pour de premiers essais.

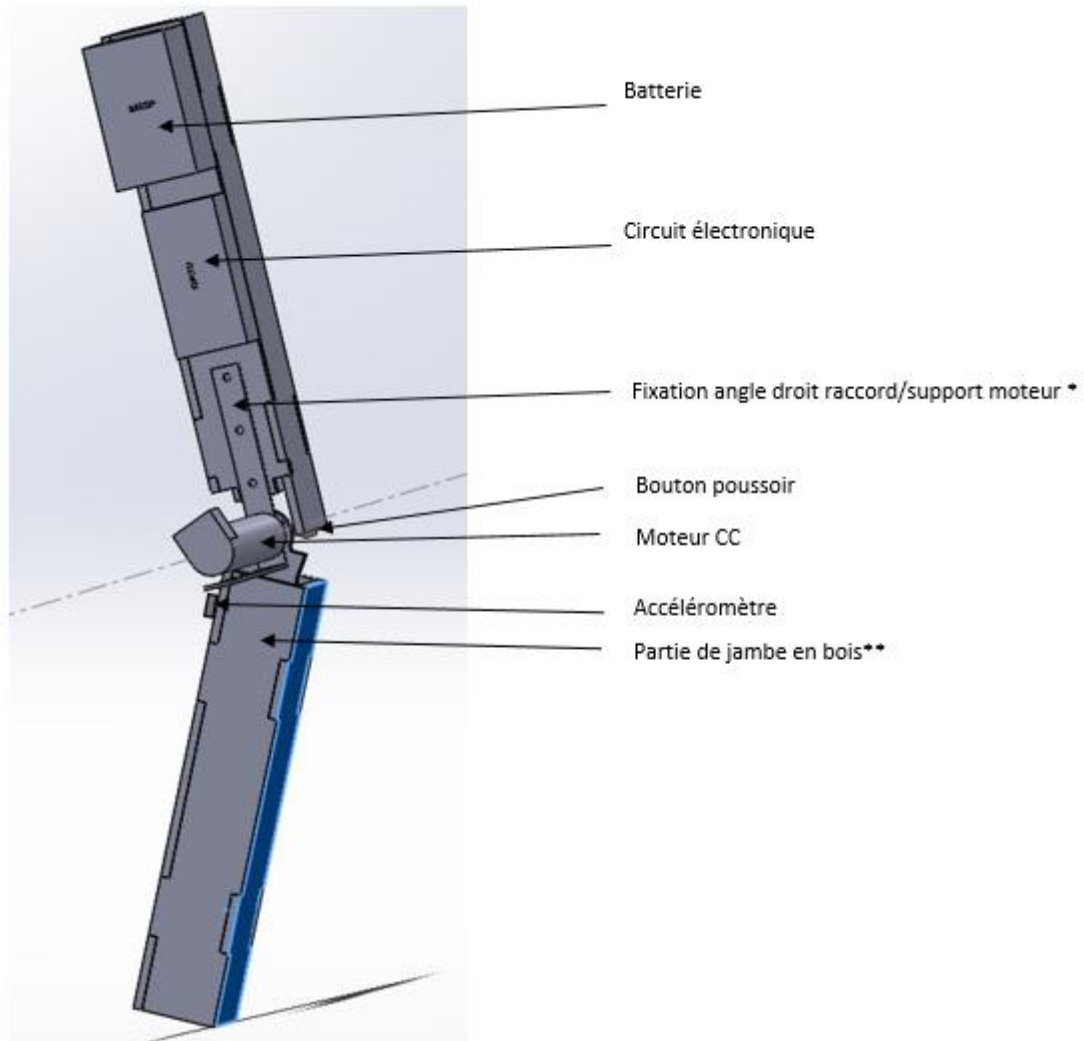


FIGURE 12 : SOLIDWORKS HERACLES

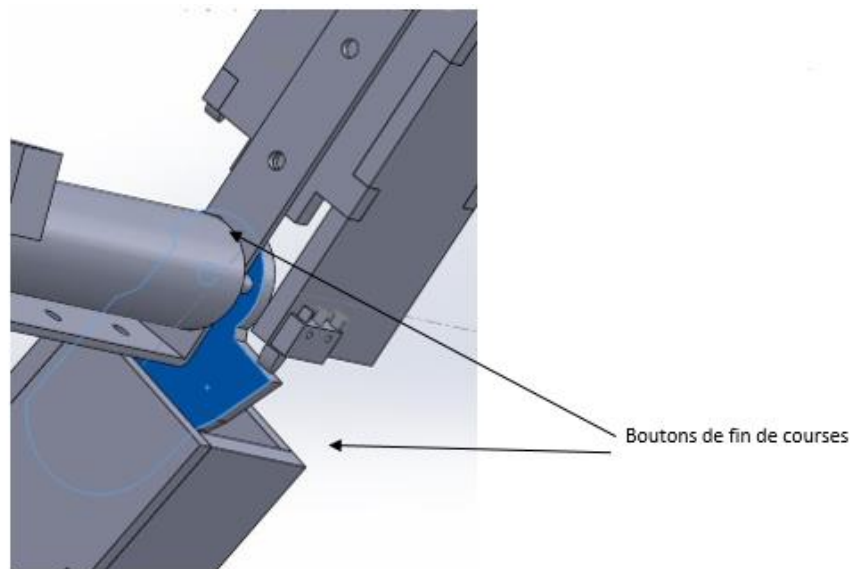


FIGURE 13 : SOLIDWORKS HERACLES ZOOM SUR LES CAPTEURS DE FIN DE COURSES

Le moteur est perpendiculaire à la jambe et dépasse sur 10cm, le rendant plus susceptible de recevoir un choc. L'idéal serait de placer le moteur en parallèle à la jambe et raccorder avec des engrenages ou juste un tube à angle droit. Nous avons choisi de le laisser perpendiculaire dans un premier temps.

Le support sert donc à la fois à faire un raccord entre la partie supérieure « fixe » de la jambe (par rapport au moteur du genou) et la partie inférieure mobile de la jambe et à la fois à prévenir le désaxage du moteur par rapport à son axe en cas de choc.

Les parties hautes et basses de la jambe ainsi que le raccord pivot ont été usinées dans du bois de 6mm d'épaisseur. La structure est en 3D pour assurer un minimum de robustesse à l'ensemble. La partie haute des jambes mesure 35cm, la partie basse 30cm. L'idéal serait que la structure s'adapte à la taille des jambes de chaque individu. La partie haute comprend également une structure qui sert à limiter l'ampleur du mouvement exécutable.

Un genou humain peut se plier au maximum jusqu'à 160°. Lors d'une course on peut aller jusqu'à 110° et lors d'une marche on ne dépasse pas les 90° d'angle.

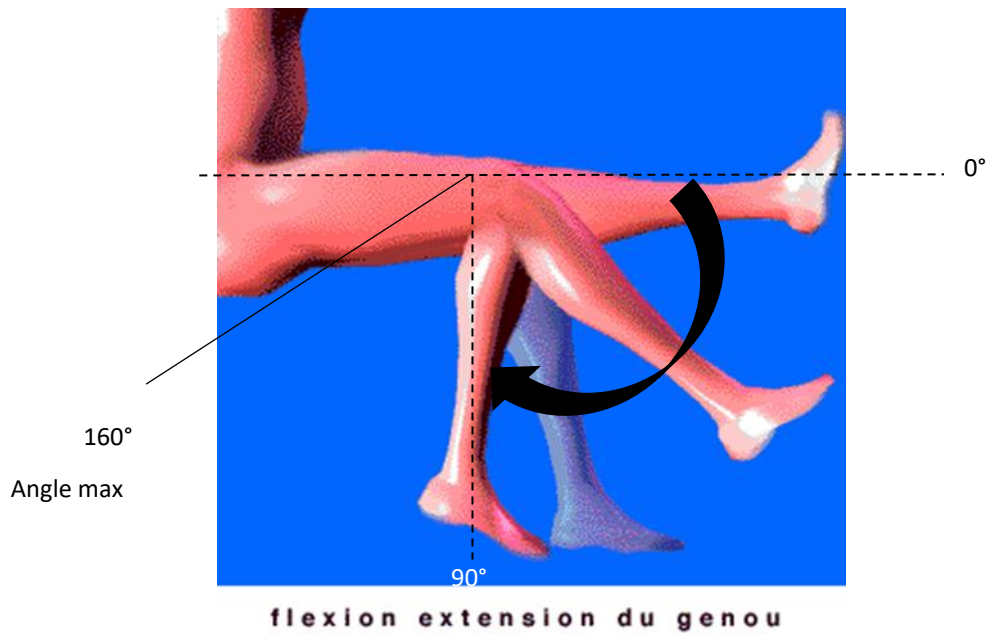


FIGURE14: JAMBE DEPLIEE ET REPLIEE AU MAXIMUM.

Nous avons donc décidé d'une structure qui force le mouvement du moteur à rester dans des angles acceptables pour la jambe.

La jambe peut ainsi effectuer des mouvements usuels en restant dans ses limites : debout, marche. Les éléments mécaniques bloquent donc les mouvements à -4° (en prenant en compte qu'une jambe tendue n'est pas exactement droite) jusqu'à 119° .

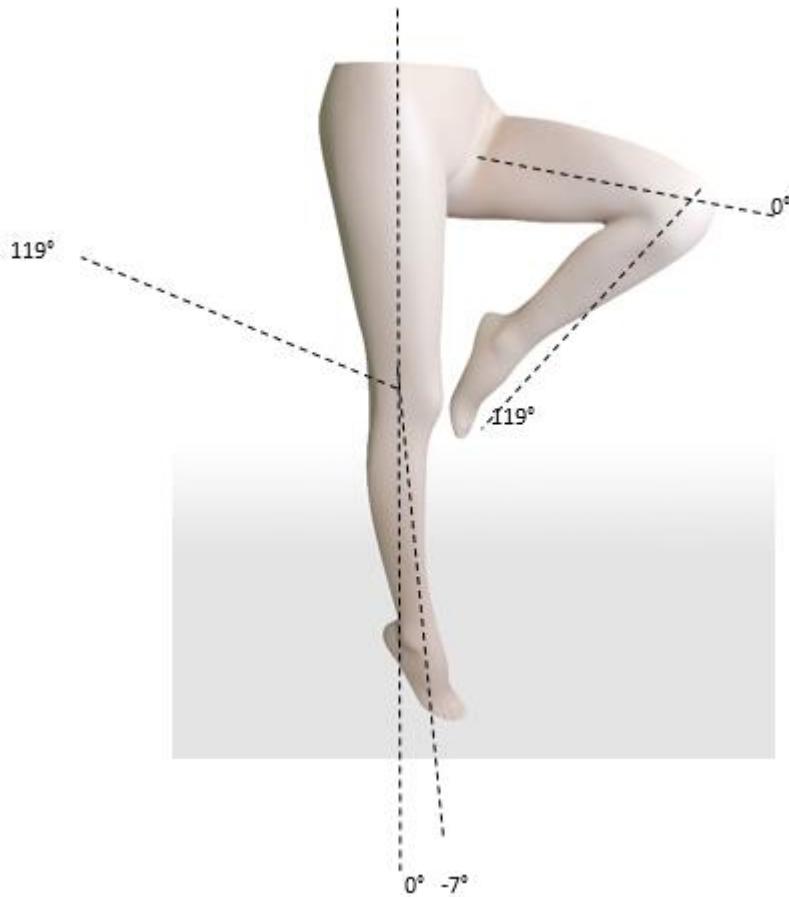


FIGURE 15: LES POSITIONS MAXIMALES AVEC HERACLES

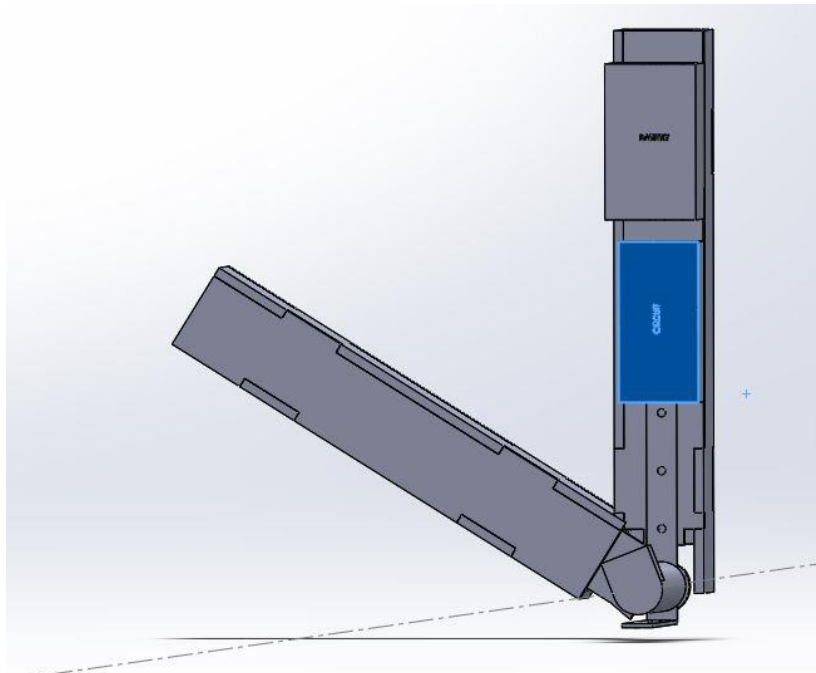


FIGURE 16 : SOLIDWORK HERACLES _ANGLE DE 119°

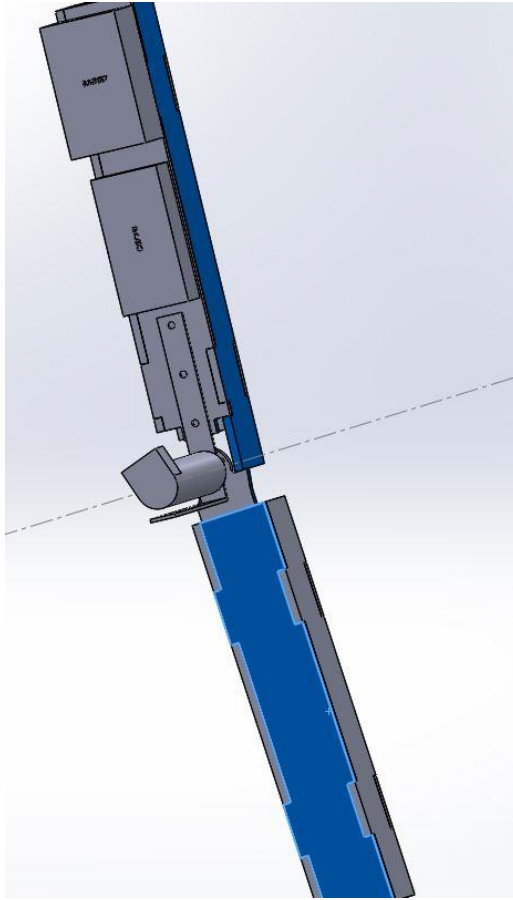


FIGURE 17: SOLIDWORK HERACLES _ANGLE DE -7°

II. Circuit Electrique

Pour ce projet, trois types d'alimentations ont été nécessaires, une de 12 Volts, une de 5 Volts et une de 3,3 Volts. Nous avons donc choisi d'alimenter notre projet par une batterie 12V et d'utiliser un Buck qui transforme la tension de 12 Volts en 5 Volts. Ainsi, nous obtenons une tension 12 Volts que l'on envoi dans le driver afin d'alimenter le moteur et une tension de 5 Volts qui alimente le microcontrôleur. Enfin nous avons ajouté un régulateur linéaire (LM1117T) pour obtenir une tension de 3,3 Volts qui nous permet d'alimenter l'accéléromètre.

A) Alimentation 12V

A l'exception du capteur EMG, tout notre projet est alimenté par une batterie 12 Volts. Cette tension sera utilisée par le moteur mais aussi transformée en 5 Volts pour alimenter le microcontrôleur puis en 3,3 Volts pour alimenter l'accéléromètre.

Etant donné qu'il s'agit de l'alimentation principale du système, on a ajouté un interrupteur en sortie de batterie qui permet de contrôler la mise en fonctionnement et l'arrêt du système. De cette manière, avec un seul interrupteur on peut éteindre toutes les alimentations du système (à l'exception, encore, de celle du capteur EMG).

Pour le branchement sur la carte, nous avons dû prévoir des fils plus épais que ceux utilisés pour le 5 Volts car le moteur demande un courant assez élevé (pouvant aller jusqu'à 3.8 Ampères). Aussi, pour faciliter le câblage, nous avons utilisé les rails sur le côté de la carte qui permet d'amener l'alimentation en 12 Volts où elle est nécessaire.

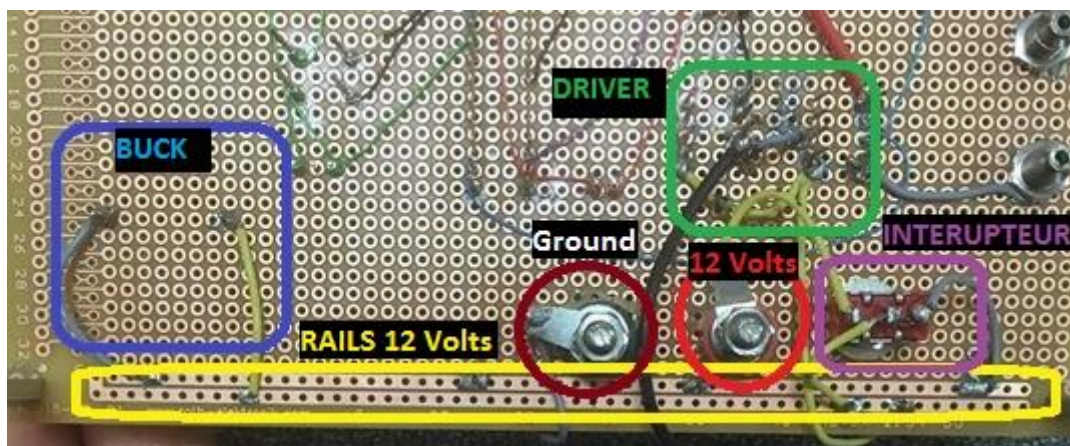


FIGURE 18 : PHOTO DU CABLAGE DE L'ALIMENTATION 12 VOLTS

B) Alimentation 5V

La plus grosse partie de notre système électronique, et particulièrement le microcontrôleur, est alimentée avec du 5 volts. Afin d'obtenir cette tension, nous utilisons un Buck qui transforme notre alimentation 12 Volts en 5 volts. La mise en place de ce Buck est plutôt simple, il suffit de brancher les 2 masses des deux côtés (qui sont reliées à l'intérieur du Buck) et de brancher le 12 Volts en entrée ainsi que récupérer le 5 Volts en sortie.

Pour indiquer que l'alimentation 5 Volts est active, on a mis en place une LED rouge en parallèle de la sortie du Buck. Une résistance pour limiter le courant dans la LED a été prévu et dimensionné comme suit.

$R = U/I$ avec $U = 5$ Volts et I_{max} , le courant maximum accepté par la LED = 0.020 Ampères soit $R = 5/0.020 = 250$ Ohms. On obtient le schéma de câblage suivant :

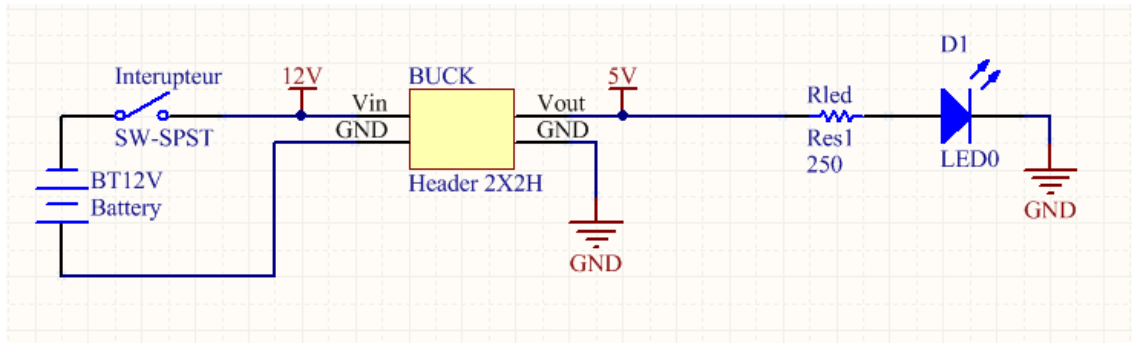


FIGURE 19 : SCHEMA DU CABLAGE DU BUCK

De la même manière que pour l'alimentation 12 Volts nous utilisons l'autre paire de rails de la carte. Cela permet d'alimenter plus facilement le microcontrôleur, la LED, les boutons de fin de course et l'alimentation 3,3 Volts.

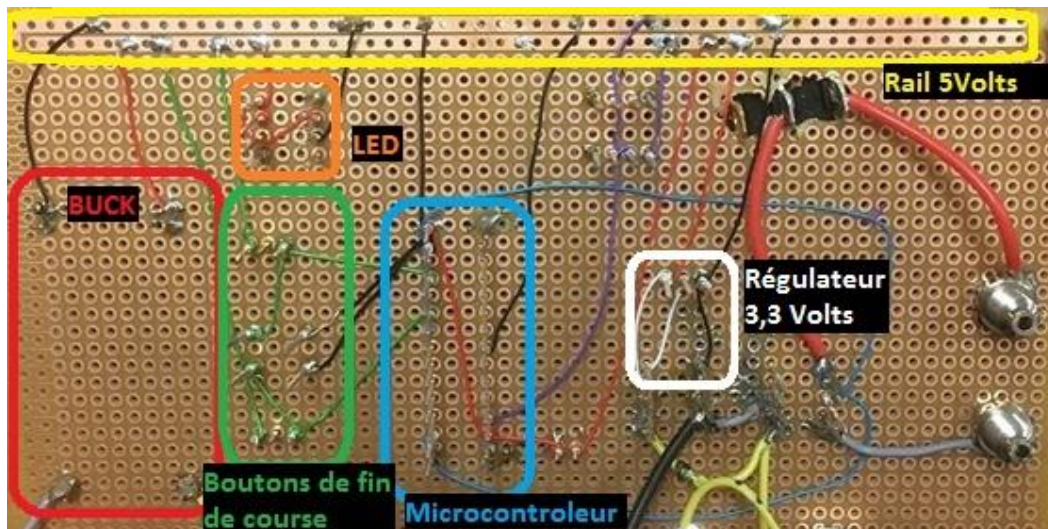
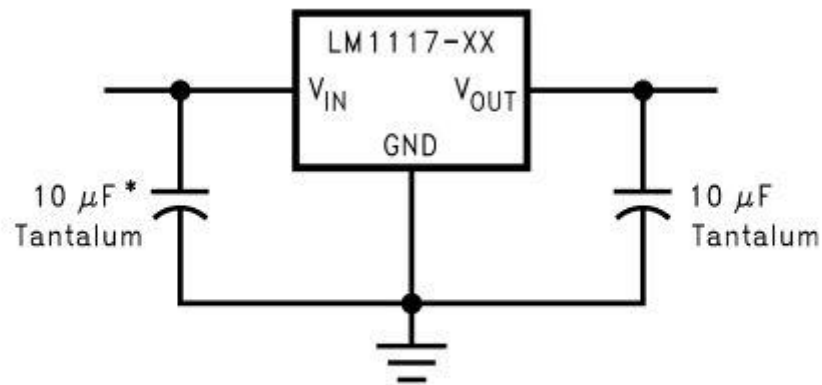


FIGURE 20 : PHOTO DU CABLAGE DE L'ALIMENTATION 5 VOLTS

C) Alimentation 3,3V

L'accéléromètre que nous utilisons (ADXL 3xx) demande une alimentation 3,3 Volts, pour obtenir celle-ci, nous utilisons un régulateur linéaire LM1117T-3.3 qui prend une alimentation 5 Volts en entrée et renvoi du 3,3 Volts en sortie. Comme pour la plupart des régulateurs linéaires, deux condensateurs sont nécessaires. Ici il s'agit de condensateurs 10 microfarads branchés comme suit :



* Required if the regulator is located far from the power supply filter.

FIGURE 21 : SCHEMA DE CABLAGE DU LM1117-XX SELON LA DATASHEET

On fera attention à l'ordre des pins du régulateurs :

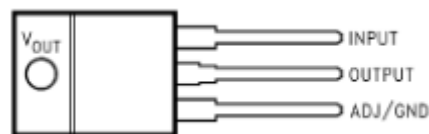


FIGURE 22 : CONFIGURATION DES PINS DU LM1117-3.3 SELON LA DATASHEET

D) Alimentation du capteur EMG

Le capteur EMG nécessite une alimentation spécifique soit du 9 Volts et du -9 Volts. Pour l'obtenir, nous utilisons deux piles 9 Volts branchées de la manière suivante :

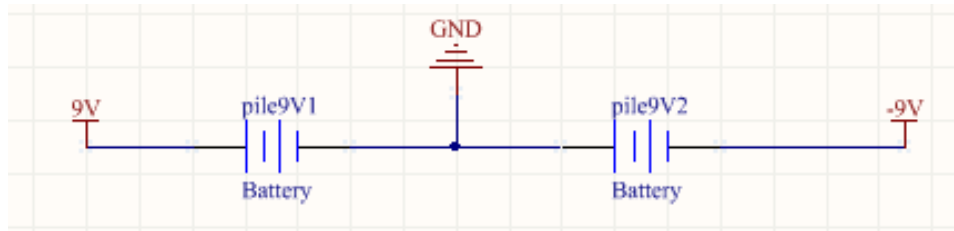


FIGURE 23 : SCHEMA DE CABLAGE DE L'ALIMENTATION DU CAPTEUR

III. Commande moteur

A) Driver

Résumons la problématique actuelle. Nous devons créer un circuit électrique qui fera la jonction entre le Microprocesseur et le moteur tout en prenant soin de bien séparer la partie commande de la partie puissance (histoire que notre Microprocesseur ne se prenne pas un courant de 100mA en entrée). Commençons simplement avec un pont en H.

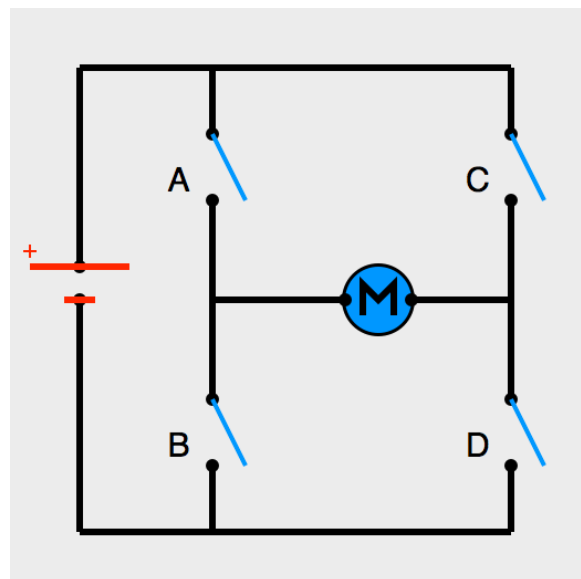


FIGURE 24 : SCHEMA D'UN PONT EN H

Le principe est plutôt simple :

En fermant **A** et **D** on fait tourner le moteur dans un sens

En fermant **B** et **C** on fait tourner le moteur dans le sens opposé

Maintenant mettons, à la place des interrupteurs, des transistors dont la base est commandée par le microprocesseur et des diodes de reflux de courant pour éviter que l'entraînement du moteur ne crée du courant qui pourrait endommager le microprocesseur. On a donc le schéma suivant :

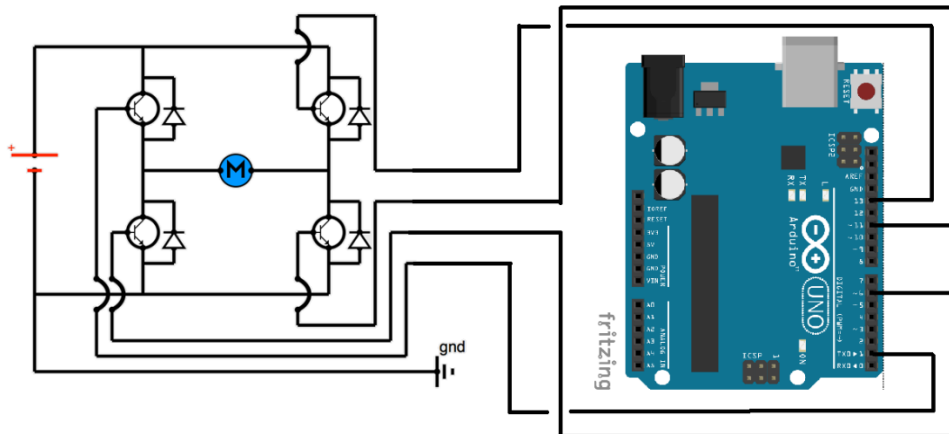
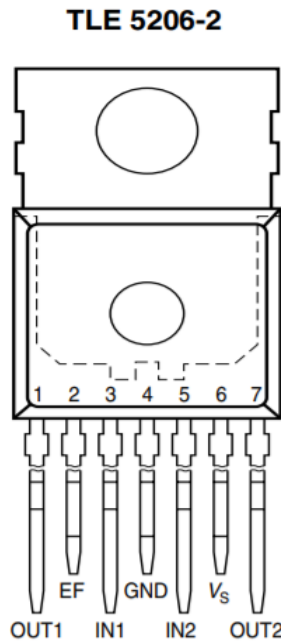


FIGURE 25 : SCHEMA DE CABLAGE D'UN PONT EN H BRANCHE A UNE CARTE

La base de chaque transistor est connectée à une sortie analogique du PIC ce qui permettrait de contrôler l'afflux de courant du moteur et ainsi sa vitesse. Ainsi on sépare la partie commande de la partie puissance. On pourrait faire nous-même ce pont en H mais des composants tout en 1 existent déjà. Ils sont beaucoup plus efficaces et acceptent des pics de courant plus importants donc nous allons opter pour cette solution. La 1ere chose à faire est de regarder la datasheet du moteur pour choisir un driver en accord avec le fonctionnement du moteur. La datasheet du moteur prévoit un courant de 34 mA en nominal (avec un pic de 3.7A au démarrage) pour du 12V. nous avons donc choisi pour driver le TLE52062GAUMA1.

TLE52062GAUMA1 :



Le TLE 5206-2 est basiquement un pont en H. composé de 7 pins

1. **OUT1** : Connecté au + du moteur
2. **EF** : Error Flag
3. **IN1** : Connecté à la Pin ...
4. **GRD** : Connecté à la masse du circuit
5. **IN2** : Connecté à la Pin ...
6. **Vs** : Connecté au 12V
7. **OUT2** : Connecté au – du moteur

FIGURE 26 : SCHEMA DU TLE 5206-2

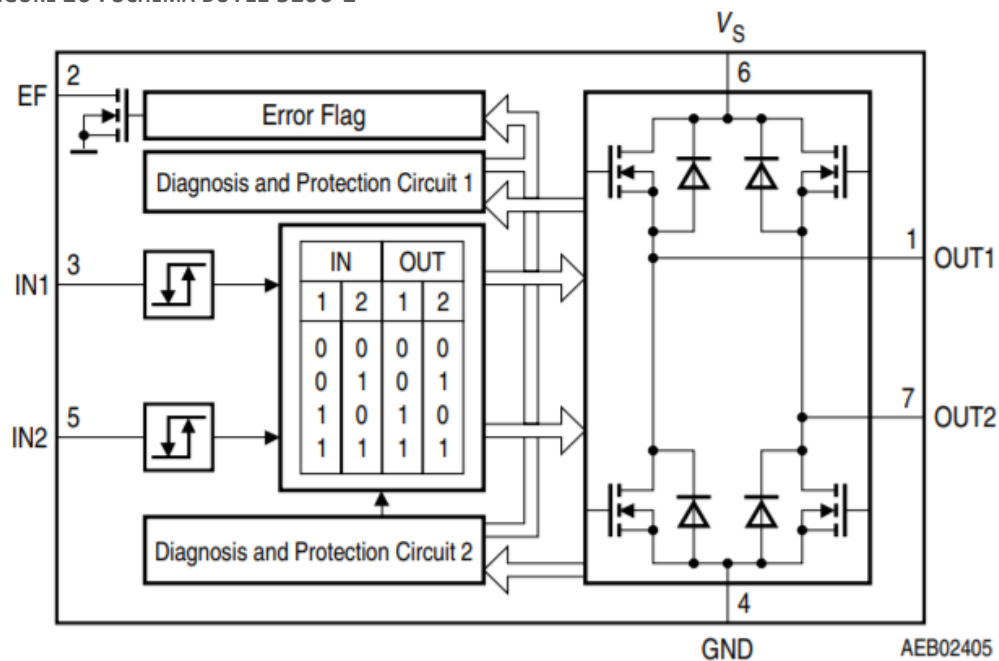


FIGURE 27 : DIAGRAMME BLOC DU DRIVER

Sens Horaire :

- **IN1** doit être en High
- **IN2** envoi une PWM

Sens anti-Horaire :

- **IN2** doit être en High
- **IN1** envoie une PWM



FIGURE 28 : TLE 5206-2

B) PWM

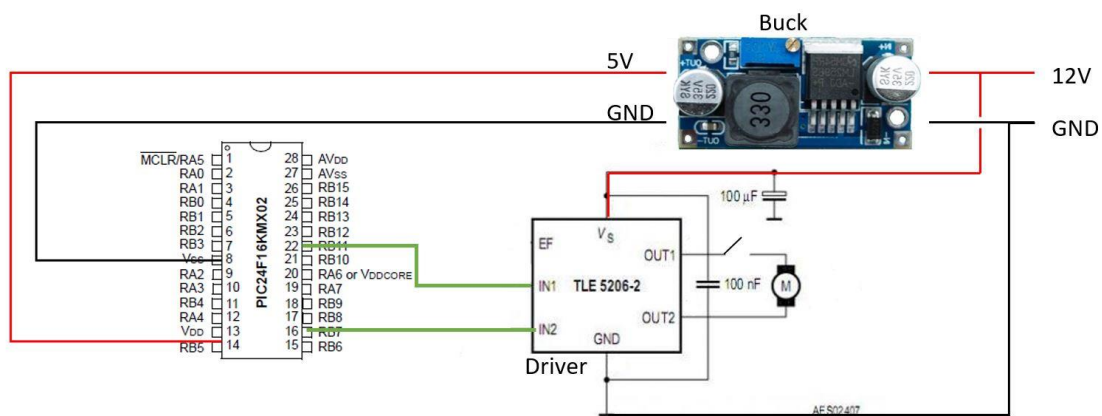


FIGURE 29 : CABLAGE DE LA PWM AU DRIVER ET AU MOTEUR

Nous avons vu avec le Driver que l'on pouvait contrôler le sens du moteur. Cela dit nous ne contrôlons pas encore sa vitesse. Le meilleur moyen de contrôler la vitesse d'un moteur courant continu est de contrôler l'afflux de courant qui traverse le moteur. Pour cela on va utiliser un hacheur de courant directement disponible depuis le microcontrôleur la PWM.

Basiquement la PWM est un signal carré répété à intervalles réguliers. Il est caractérisé par une période, un rapport cyclique :

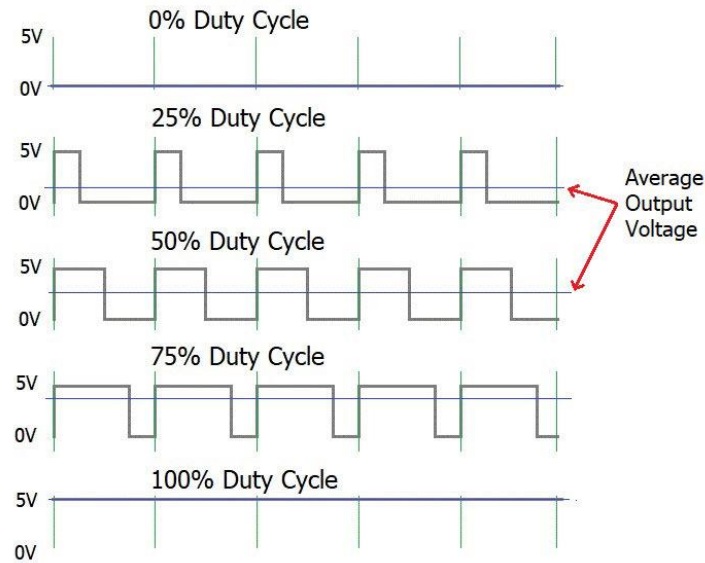


FIGURE 30 : DIAGRAMME DE LA PWM EN ENTRE DU DRIVER

- Pour coder la PWM il faut d'abord comprendre comment le microcontrôleur crée cette PWM. Le PIC va créer une tension d'une certaine pente alpha qui recommencera selon une certaine période. Le PIC possède 3 registres qui nous intéressent.

Ra : valeur où le signal chute

Rb : valeur où le signal augmente

PRL : valeur de la période globale

- En admettant $A < B$ et Période = B il nous suffit de faire varier A entre 0 et B pour obtenir un rapport cyclique de 0% à 100%

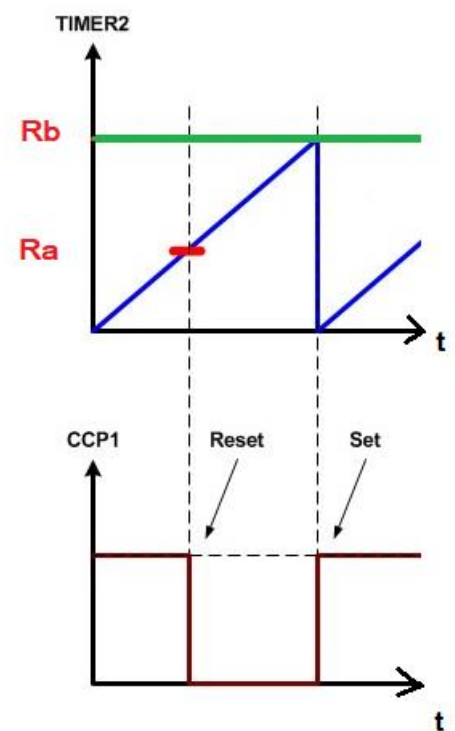


FIGURE 31 : PARAMETRE DE LA PWM

Commençons par initialiser la PWM en créant la fonction [init_PWM](#) :

Le but d'initialiser la PWM est de la configurer dès la mise en service du microcontrôleur de manière à ensuite pouvoir l'utiliser comme on le veut au cours du programme.

Surtout ne pas oublier d'inclure la bibliothèque associée à la PWM.

```
#include "pwm.h"

void init_PWM(void)
{
    // Set MCCR operating mode
    CCP1CON1Lbits.CCSEL = 0;           // Set MCCR operating mode (OC mode)
    CCP1CON1Lbits.MOD = 0b0101;        // Set mode (Buffered Dual-Compare/PWM mode)

    //Configure MCCR Timebase
    CCP1CON1Lbits.TMR32 = 0;           // Set timebase width (16-bit)
    CCP1CON1Lbits.TMRSYNC = 0;         // Set timebase synchronization (Synchronized)
    CCP1CON1Lbits.CLKSEL = 0b0000;     // Set the clock source (Tcy)
    CCP1CON1Lbits.TMRPS = 0b00;        // Set the clock pre-scaler (1:1)
    CCP1CON1Hbits.TRIGEN = 0;          // Set Sync/Triggered mode (Synchronous)
    CCP1CON1Hbits.SYNC = 0b000000;     // Select Sync/Trigger source (Self-sync)

    //Configure MCCR output for PWM signal
    CCP1CON2Hbits.OCAEN = 1;           // Enable desired output signals (OC1A)
    CCP1CON3Hbits.OUTM = 0b000;        // Set advanced output modes (Standard output)
    CCP1CON3Hbits.POLACE = 0;          // Configure output polarity (Active High)
    CCP1TMR1 = 0x0000;                 // Initialize timer prior to enable module.
    CCP1PRL = 0x00AA;                  // Configure timebase period
    CCP1RA = 0x00AA;                   // Set the rising edge compare value
    CCP1RB = 0x00AA;                   // Set the falling edge compare value 1023 values
    CCP1CON1Lbits.CCPON = 1;           // enable PWM1
}
```

Comme on peut le voir sur le code

Ra correspond à [CCP1RA](#)

Rb correspond à [CCP1RB](#)

PRL correspond à [CCP1PRL](#)

CCP1CON2Hbits.OCAEN est le pin de sortie du signal de la PWM (pin 16)

Initialisons la 2nd PWM maintenant. Niveau code se sera pareil seul le pin de sortie sera différent.

```
void init_PWM2(void)
{
    // Set MCCR operating mode
    CCP2CON1Lbits.CCSEL = 0;           // Set MCCR operating mode (OC mode)
    CCP2CON1Lbits.MOD = 0b0101;        // Set mode (Buffered Dual-Compare/PWM mode)
    //Configure MCCR Timebase
    CCP2CON1Lbits.TMR32 = 0;           // Set timebase width (16-bit)
    CCP2CON1Lbits.TMRSYNC = 0;         // Set timebase synchronization (Synchronized)
    CCP2CON1Lbits.CLKSEL = 0b000;       // Set the clock source (Tcy)
    CCP2CON1Lbits.TMRPS = 0b00;        // Set the clock pre-scaler (1:1)
    CCP2CON1Hbits.TRIGEN = 0;          // Set Sync/Triggered mode (Synchronous)
    CCP2CON1Hbits.SYNC = 0b000000;     // Select Sync/Trigger source (Self-sync)
    //Configure MCCR output for PWM signal
    CCP2CON2Hbits.OCAEN = 1;           // Enable desired output signals (OC2A)
    CCP2CON3Hbits.OUTM = 0b000;        // Set advanced output modes (Standard output)
    CCP2CON3Hbits.POLACE = 0;          // Configure output polarity (Active High)
    CCP2TMRCL = 0x0000;               // Initialize timer prior to enable module.
    CCP2PRL = 0x00AA;                 // Configure timebase period //1023=3ff
    CCP2RA = 0x00AA;                  // Set the rising edge compare value
    CCP2RB = 0x00AA;                  // Set the falling edge compare value 1023 values
    CCP2CON1Lbits.CCPON = 1;          // enable PWM1
}
```

Maintenant on peut modifier en temps réel la valeur de Ra de manière à faire varier la PWM de 0% à 100%. Au niveau de la période de la PWM avec `CCP1PRL` elle doit être assez petite de manière à ce que la fréquence soit dans les 10Khz pour que le moteur puisse tourner sans difficulté. Vu qu'on le veut également relativement silencieux on doit au moins doubler cette fréquence pour dépasser le seuil des fréquences audibles. Pour cela on va set la registre `PRL = 0x00AA` puis on augmentera encore la fréquence dans d'autres fonctions que nous verrons plus en détail plus tard.

Avant de passer aux fonctions qui vont gérer les valeurs des registres des PWM on va d'abord définir quelques registres importants pour ne pas se perdre. Ainsi en début de code du fichier

[pwm.c](#)

```
//commençons par définir quelque variables
#define etat_pwm_1 CCP1CON1Lbits.CCPON
#define etat_pwm_2 CCP2CON1Lbits.CCPON
#define vitesse_pwm_1 CCP1RA
#define vitesse_pwm_2 CCP2RA
```

Maintenant que cela a été fait et que l'on sait faire fonctionner une PWM on va créer une fonction de direction pour pouvoir faire tourner le moteur dans un sens ou dans l'autre sans avoir à changer tout un tas de registres. Pour cela on va retourner sur le [fonctionnement du Driver](#).

Petit rappel du fonctionnement :

Sens Horaire :

- **IN1** doit être en High
- **IN2** envoie une PWM

Sens anti-Horaire :

- **IN2** doit être en High
- **IN1** envoie une PWM

Plutôt que de déconnecter la PWM de la Pin choisie pour ensuite la configurer comme un bit logique en état haut. L'idée va être de laisser la PWM active mais de mettre un rapport cyclique de 100%. Ainsi il n'y aura pas de front descendant et c'est comme si le pin était constamment en 5V. commençons à écrire la fonction comme cela.

```
void PWM_activated(unsigned short direction, unsigned short vitesse)
{
    if(direction == 0)
    {
        etat_pwm_1 = 1;           // On active la PWM1
        vitesse_pwm_1 = vitesse;  //on allume la PWM1 avec la fréquence "freq_pwm"

        vitesse_pwm_2 = 0x0000;
        etat_pwm_2 = 1;           // On met à 5V constant la PWM2

    }
    else if(direction == 1)
    {
        vitesse_pwm_1 = 0x0000;
        etat_pwm_1 = 1;           // on met à 5V constant la PWM1

        etat_pwm_2 = 1;           // On active la PWM2
        vitesse_pwm_2 = vitesse;  //on allume la PWM2 avec la fréquence "freq_pwm"

    }
    else
    {
        //on éteint les 2 PWM
        etat_pwm_1 = 0;           // On desactive la PWM1
        etat_pwm_2 = 0;           // On desactive la PWM2
    }
}
```

IN1 = PWM 1

IN2 = High

IN1 = High

IN2 = PWM 2

IN1 = Low

IN2 = Low

- La fonction ne prend en 1^{er} argument qu'un chiffre entre 0 et 2 :
0 = sens horaire
1 = sens anti-horaire
2 = les 2 PWM sont éteints
- Et en 2eme argument la vitesse voulue.

Comme convenue pour faire fonctionner le moteur il nous faut une fréquence assez haute pour que le moteur puisse fonctionner. La PWM est directement reliée à l'horloge du microcontrôleur.

IV. Capteurs

A) Capteur EMG

Le capteur Musculaire : est utilisé pour mesurer l'activité des muscles par l'intermédiaire d'un potentiel électrique, connu sous le nom d'électromyographie (EMG), et est traditionnellement utilisé pour la recherche médicale et le diagnostic des maladies neuromusculaires. Cependant, avec l'avènement des microcontrôleurs et circuits intégrés de plus en plus petits et encore plus puissants, les circuits et capteurs EMG ont trouvés leur chemin dans les prothèses, la robotique et d'autres systèmes de contrôle.

Définitions les besoins actuels : on veut récupérer le signal d'un muscle qui d'après Wikipédia et d'autres sites envoi entre +/- 1.5mV, le reconditionner et le diriger vers un Microprocesseur pour ensuite pouvoir l'utiliser. Ce qui nous donne le schéma suivant :

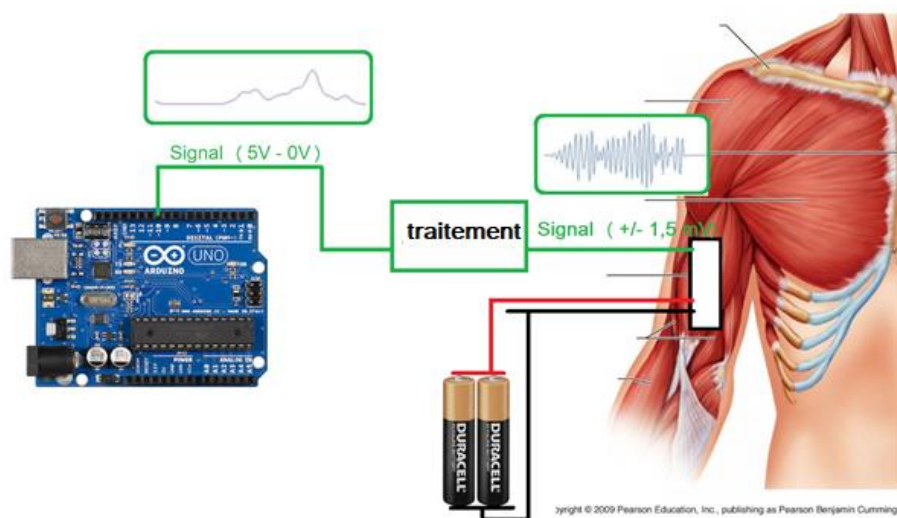


FIGURE 32 : SCHEMA EXPLICATIF DU FONCTIONNEMENT DU CAPTEUR EMG

Pour le capteur musculaire on a 2 choix qui s'offrent à nous :

- 1) Soit on prend un capteur EMG en kit qui possède le capteur et le circuit électronique pour augmenter le gain et réduire le bruit de manière à ce que le signal soit lisible par une carte électronique. Ce qui délivre à la fin une tension entre 0V et 5V.
 - **Avantages** : on n'a pas à se soucier de faire un montage AOP pour avoir un signal lisible
 - **Inconvénient** : le produit coûte plus cher (70€)
 -
- 2) Soit on le fait nous-même. En achetant juste un capteur Myographique et composant le reste du montage. Le montage des AOP devra dans un 1^{er} temps redresser le signal, puis filtrer l'enveloppe avec un passe bas comme cela :

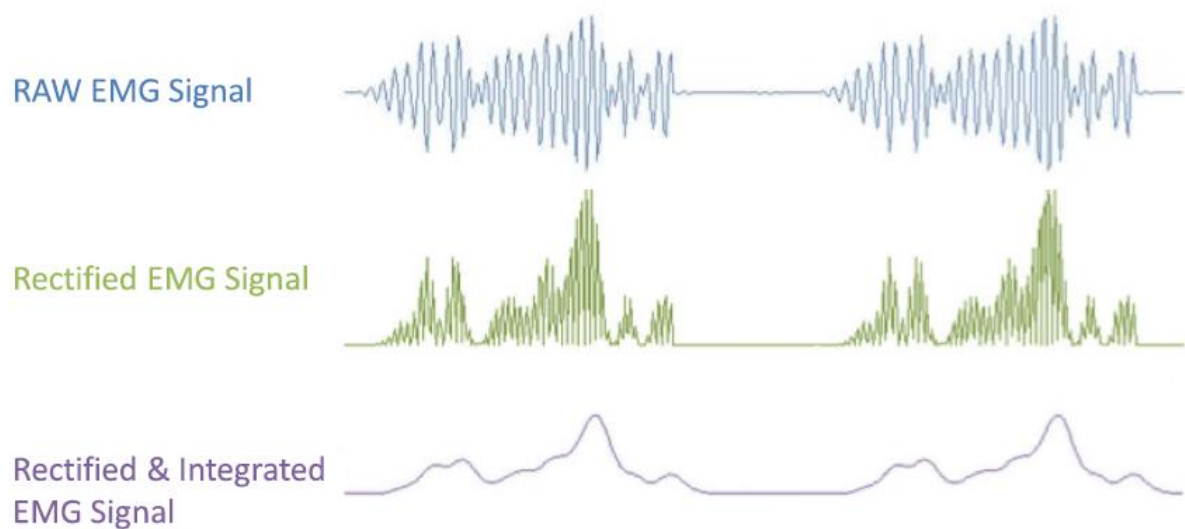


FIGURE 33 : SIGNAL LU, RECTIFIE ET RENVOYE PAR LE CAPTEUR EMG

- **Avantages** : on utilise nos connaissances en AOP pour faire du traitement du signal + on réduit drastiquement les couts du projet (10€)
- **Inconvénient** : le dimensionnement des AOP pourrait nous faire perdre un temps précieux.

Conclusion :

Après avoir effectué quelques tests avec le capteur EMG fourni par l'école nous nous sommes rendu compte de l'imprécision de ce dernier. Le capteur EMG actuel nous donne du 1V constant et, dans le meilleur des cas, en accentuant le mouvement, du 3V.

Ce qui nous laisse peu de marge de manœuvre pour coder une PWM en fonction de celui-ci. Qui plus est le bruit est assez important, un petit choc dans les câbles suffit à déclencher un saut de 3V.

Pour simuler un EMG parfait on a donc opté pour un potentiomètre. De manière à pouvoir gérer un signal ayant une plage allant de 0 à 5V. Ce signal simule donc un EMG idéal.

L'EMG et le potentiomètre fonctionnent de la manière suivante :

Si le muscle est stimulé, un signal est envoyé à l'ADC. Si le muscle n'effectue pas de mouvement, le signal est nul.

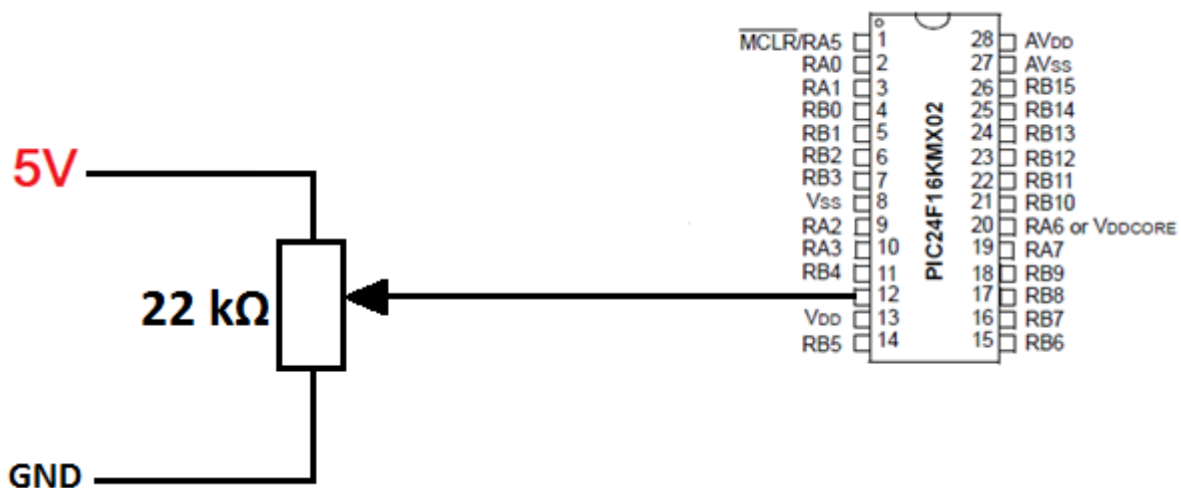


FIGURE 34 : SCHEMA DE CABLAGE DU POTENTIOMETRE

B) Convertisseur analogique-numérique (ADC)

Explication concernant l'ADC

Sur une pic24F16KM202 :

L'ADC prend un voltage de référence (nous avons choisi de prendre les pins AVDD et AVSS branchés à une source de 5V) et le compare à la tension d'entrée venant des capteurs. Il compare cette valeur à celle de la tension de référence et le caractérise par une valeur allant de 0 à $2^{nb\text{rs de bits de conversion}} - 1$. Nous avons choisi un convertisseur analogique/digital de 10 bits (soit 1023 valeurs possibles, soit 4,9mV de sensibilité pour des signaux allant de 0V à 5v).

Donc toutes les tensions des capteurs possibles sont échelonnées en entiers compris entre 0 et 1023.

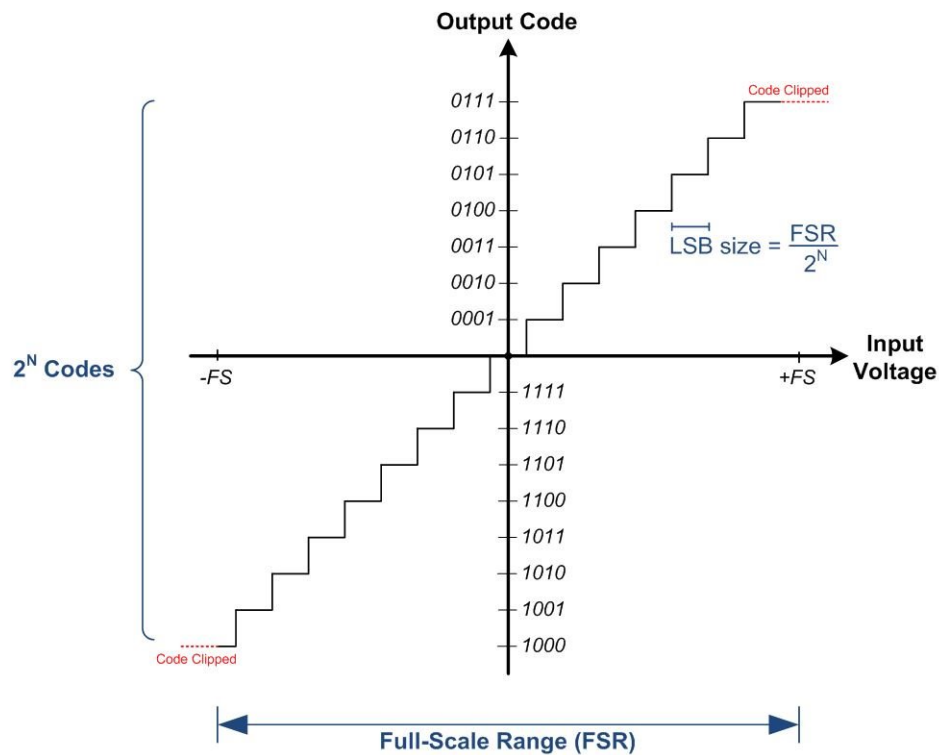


FIGURE 35 : EXEMPLE DE GRADATION DES VALEURS D'ENTREES REPARTIES ENTRE LES VALEURS MIN ET MAX POSSIBLES AVEC N=4

On peut également choisir un convertisseur 12bits qui augmenterait la sensibilité, mais pour l'utilisation que nous en faisons ce n'est pas nécessaire, au contraire on risque d'augmenter la possible présence de bruit.

Code d'initialisation de l'ADC :

```
void init_ADC(void)
{
    AD1CON1bits.ADSIDL= 0;           // don't STOP IN IDLE MODE
    AD1CON1bits.ASAM= 0;             // not auto-start
    AD1CON1bits.MODE12= 0;           // fct en 10bit, 12bits pas utile
    AD1CON1bits.FORM = 0b00;         // format bit: int
    AD1CON1bits.SSRC = 0b000;        // manual convert trigger mode
    AD1CON2 = 0;                     // Configure A/D voltage reference
                                    // Vr+ and Vr- from AVdd and AVss (PVCFG<1:0>=00, NVCFG=0)
                                    // Interrupt after every sample
    AD1CON3 = 0x0002;                // Sample time: Tad = 3Tcy
                                    // A/D conversion clock as Tcy
    //AD1CON4 = 0;                    // Allocates 1 word of buffer to each analog input
    AD1CHS= 0x0010;                  // --> the input is the positive input 0x10=16
    AD1CSSL = 0;                     // No inputs are scanned.
}
```

Grossièrement :

-l'AD1CON1 gère le mode de fonctionnement de l'ADC :

ADC ON/OFF (ADON)

- La data entrante enregistrée sur 10 ou 12bits (MODE12)
- La tâche à effectuer lors de l'échantillonnage (SAMP)
- La source d'horloge de l'échantillonnage (SSRC)

NB : Il y a deux modes d'utilisations recommandés pour cette commande

0b 0000 → le bit du SAMP doit être mis à zéro dans le software (voir [Read ADC](#) plus bas)

0b 0111 → le bit du SAMP est mis à zéro après un nombre de TAD (A/D conversion clock) suffisant pour effectuer le sampling, la conversion du signal est automatique.

(Le code est à adapter selon notre choix)

Sinon on peut régler cette horloge sur la fréquence de n'importe quel Pin d'entré.

- Le fonctionnement automatique de l'échantillonnage ou non (ASAM)

-l'AD1CON2 gère le mode de fonctionnement de l'échantillonnage :

- Le voltage de référence (PVCFG et NVCFG0)
- Le mode d'enregistrement des datas entrantes, où et comment (BUFREGEN, BUFS, BUFM)

-l'ADC1CON3 gère le mode de fonctionnement de la conversion de l'échantillon :

- L'horloge de fonctionnement (ADRC)

- La fréquence de conversion (ADCS)

Code de fonctionnement de l'ADC :

```
unsigned short ADC_read(void)
{
    AD1CHSbits.CHOSA = 16;           // Configure input channels on input AN16 pin12 (1000)

    AD1CON1bits.ADON = 1;           // A/D CONVERTER IS OPERATING
    AD1CON1bits.SAMP = 1;           // Start sampling the input
    __delay_ms(6);                   // delay assigned empirically: tad min of 12cycles -->tad=3TCY so 31TAD, not optimal
    AD1CON1bits.SAMP = 0;           // stop sampling and Start converting
    while(!AD1CON1bits.DONE){};     // while conversion not completed
    unsigned short ADC_VALUE= ADC1BUF0; //the value sampled and converted is stocked
    AD1CON1bits.ADON = 0;           // A/D CONVERTER IS NOT OPERATING
    return ADC_VALUE;
}
```

Le *delay* doit être choisi en prenant en compte la Clock de l'ADC pour optimiser le temps au maximum. Il ne doit en aucun cas être inférieur au temps de fonctionnement.

Pour un 10-bits conversion, il faut 12 TAD (périodes d'horloges) pour une conversion ADC.

Le TAD est le temps nécessaire à l'ADC pour convertir 1 bit.

Le TCY est la période d'un cycle d'instruction. Il est lié à l'horloge du microcontrôleur multiplié par 4 (TCY= TOSC *4) (nb : TOSC= 1/FOSC).

Fosc est la fréquence de l'horloge, Tosc est la période de l'horloge.

period, TCY. For correct A/D conversions, the A/D conversion clock (TAD) must be selected to ensure a minimum TAD time, as specified by the device family data sheet.

Equation 51-1: A/D Conversion Clock Period

$$TAD = TCY (ADCSx + 1)$$

$$ADCSx = \frac{TAD}{TCY} - 1$$

FIGURE 38: PASSAGE DE LA DATASHEET 12-BIT A/D CONVERTER WITH THRESHOLD DETECT

NB: Tous les pins ne sont pas utilisables pour l'ADC, ils doivent permettre une entrée analogique (comprendre la mention ANxx)

Lorsqu'on veut relever les valeurs envoyées à l'ADC (avec SSRC = 0b 0000) on doit :

- Donner le Channel analogique d'où vient l'information.
- Activer l'ADC
- Lancer l'échantillonnage
- Attendre un peu pour être sûr que l'opération a bien été effectuée (utile uniquement si on est en mode manuel)
- On convertit
- Tant que la conversion n'est pas terminée on reste dans la boucle
- Ensuite on peut récupérer la valeur stockée temporairement dans le Buffer
- On Désactive l'ADC

```
unsigned short ADC_read_accY(void)
{
    unsigned short Y_acc;
    AD1CHSbits.CH0SA = 18;           // Configure input channels on input AN18 pin15 (1010)
    AD1CON1bits.ADON = 1;           // A/D CONVERTER IS OPERATING
    AD1CON1bits.SAMP = 1;           // Start sampling the input
    __delay_ms(6);

    AD1CON1bits.SAMP = 0;           // stop sampling and Start converting

    while(!AD1CON1bits.DONE) {};    // while conversion not completed
    Y_acc= ADC1BUF0;

    AD1CON1bits.ADON = 0;           // A/D CONVERTER IS NOT OPERATING
    return Y_acc;
}
```

Ensuite nous avons voulu limiter les erreurs liées au bruit en faisant une moyenne des valeurs d'entrées.

```
unsigned short Average_Value_ADC(void)
{
    unsigned short value1, value2, value3, value_average;

    value1=ADC_read();
    value2=ADC_read();               //3 values are read from the EMG
    value3=ADC_read();
    value_average = (value1+value2+value3)/3; // we do an average of those values to smooth out problems linked to noise

    return value_average;
}
```

Nous avons choisi de prendre 3 valeurs (ça aurait tout aussi bien pu être 4 ou 5, ce fut choisi empiriquement) et d'en faire une moyenne avant de l'envoyer au reste du programme. Les delays présent dans le [ADC_read](#) n'impactent pas le mouvement fluide de l'exosquelette.

C) Capteur de Fin de Course

Maintenant que l'ADC a été codé il va nous falloir un moyen de dire à notre microcontrôleur qu'il arrive au maximum du mouvement autorisé par la jambe. Pour cela pleins de solutions sont possibles. Un accéléromètre, un capteur de distance etc. Cela dit pour des raisons de simplicité et d'efficacité nous avons décidé de mettre des capteurs de fin de course. Avant de passer à la partie code on va commencer par expliquer la partie câblage.

Les pins choisis seront les pins RB12 et RB14 respectivement les pin 23 et 25 de la Datasheet. Seuls les pins avec l'appellation INTx peuvent être des pins d'interruption.

On a décidé de relier le bouton poussoir au rail +5V ainsi quand le bouton sera enclenché le pin choisi sera en état high. Et donc en état low tant que le bouton ne sera pas sollicité. Voici comment se présente le câblage :

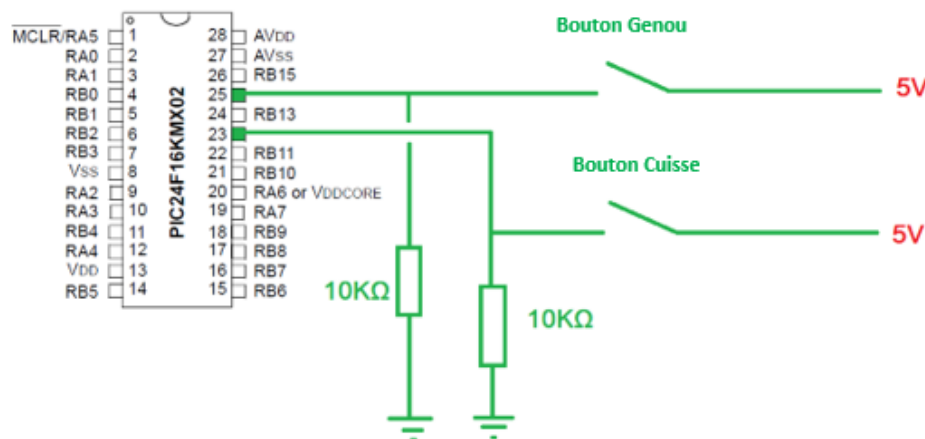


FIGURE 40 SCHEMA DE CABLAGE DES CAPTEURS DE FIN DE COURSES

Pour éviter un état de haute impédance Z aux bornes des pins on choisit de relier des résistances de Pull-up depuis les pins désirés vers la masse. Ainsi quand les boutons ne sont pas sollicités l'état des pins est en low.

Dans un premier temps on va aller dans le fichier [user.c](#) pour configurer les pins d'interruption. Une fonction d'interruption est utilisée pour stopper un programme en cours. Les interruptions ont des priorités entre elles. Notre programme n'a que deux interrupts de même priorité donc on peut donner à cette fonction n'importe quelle valeur de priorité.

Dans la fonction [InitApp\(\)](#) on va écrire :

```
TRISBbits.TRISB12 = 1;      // Set RB12 as Input
ANSBbits.ANSB12 = 0;      // Set A0 as digital IO
INTCON2bits.INT2EP = 0;    // Rising edge
IFS1bits.INT2IF = 0;      // Clear INT0 Interrupt
IEC1bits.INT2IE = 1;      // enable INT0 Interrupt
IPC7bits.INT2IP2 = 1;      // Priority
IPC7bits.INT2IP1 = 1;
IPC7bits.INT2IP0 = 0;
```

Bouton Cuisse

```
TRISBbits.TRISB14 = 1;      // Set RB14 as Input
ANSBbits.ANSB14 = 0;      // Set A0 as digital IO
INTCON2bits.INT1EP = 0;    // Rising edge
IFS1bits.INT1IF = 0;      // Clear INT0 Interrupt
IEC1bits.INT1IE = 1;      // enable INT0 Interrupt
IPC5bits.INT1IP2 = 1;      // Priority
IPC5bits.INT1IP1 = 1;
IPC5bits.INT1IP0 = 1;
```

Bouton Genou

Comme noté dans le code on va :

- TRISBbits.TRISB12 = 1; Configurer les pins en input
- ANSBbits.ANSB12 = 0; Les mettre en mode « digital Reading »
- INTCON2bits.INT2EP = 0; Dire que l'interruption se fera sur front montant
- IFS1bits.INT2IF = 0; Ecraser les valeurs précédentes du registre
- IEC1bits.INT2IE = 1; Activer l'interrupteur
- IPC7bits.INT2IP2 = 1;
- IPC7bits.INT2IP1 = 1;
- IPC7bits.INT2IP0 = 0; Et définir les priorités

Et ce pareil pour le bouton du genou.

Maintenant allons écrire dans le fichier [ininterrupt.c](#).

Au début :

```
//definition des variable que l'on vas souvent utiliser
#define Bouton_fin_cuisse IFS1bits.INT1IF
#define Bouton_fin_genou IFS1bits.INT2IF
```


Définissons ensemble le reste de la fonction d'interruption.

Dans cette fonction on fera prendre à la variable « pos » des valeurs différentes qui nous serviront plus tard pour coder le mouvement de la jambe et la faire s'arrêter en fonction de la valeur de cette variable. En 2nd ligne on va remettre le flag d'erreur à 0

```
extern unsigned short pos;
void __attribute__((interrupt, auto_psv)) _INT2Interrupt(void)
{
    pos=0;
    Bouton_fin_genou=0;
}

void __attribute__((interrupt, auto_psv)) _INT1Interrupt(void)
{
    pos=2;
    Bouton_fin_cuisse=0;
}
```

D) L'accéléromètre

Accéléromètre utilisé en tant qu'inclinomètre dans notre projet : ADXL3XX.

L'accéléromètre détecte l'orientation de la surface sur laquelle il est apposé. Il capte l'inclinaison par rapport à la gravité terrestre.

Il s'alimente en 3.3V, quelques microampères et chaque axe envoie sa position en volt.

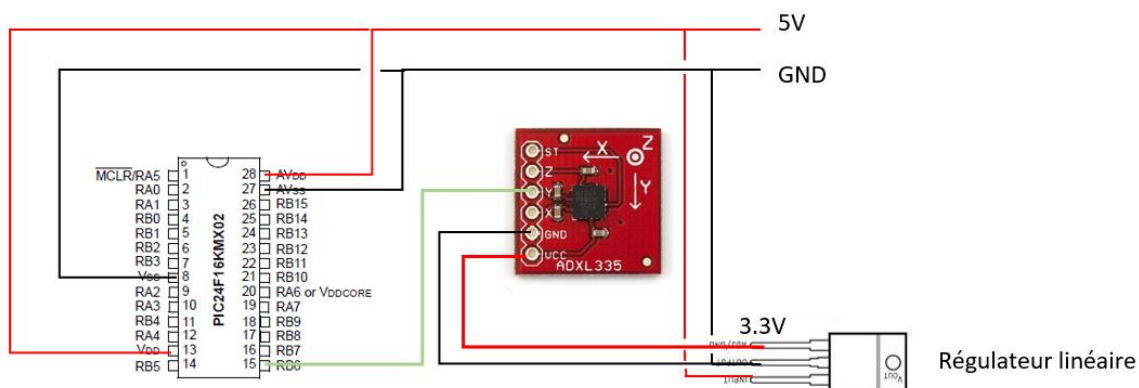


FIGURE 41: SCHEMA DE CABLAGE DE L'ACCELEROMETRE

Cet accéléromètre rend les données en Volt, la plage de chaque axe est comprise entre 0 et 3.3V. On a donc besoin de l'ADC pour analyser les données entrantes.

Pour bien déterminer les valeurs que l'on traitera il faut positionner l'accéléromètre comme on le souhaite en position verticale et ne plus le bouger de la position choisie sur la mécanique. On veut les valeurs fidèles à elles-mêmes lorsque l'on répète l'opération. C'est donc nécessaire de positionner l'ADC au même endroit avec la même inclinaison à chaque fois, sinon les valeurs rendues ne seront plus adaptées aux valeurs relevées pour le code et ça fausserait la réponse du système.

Pour cette raison nous n'avons pas donné de rôle primordial à l'accéléromètre, pour éviter qu'en cas de désaxage ou de bug le système soit mis à mal par les valeurs de cet appareil.

Tout d'abord pour déterminer le meilleur axe et la meilleure position à donner à l'accéléromètre nous avons fait des tests. Relevé des graduations de valeurs en prenant soin qu'elles ne passent pas d'une valeur faible à une valeur haute sans transition (quand l'accéléromètre a fait un tour sur son axe, il peut passer d'une valeur 0 à 360, par exemple, sur un décalage d'un degré).

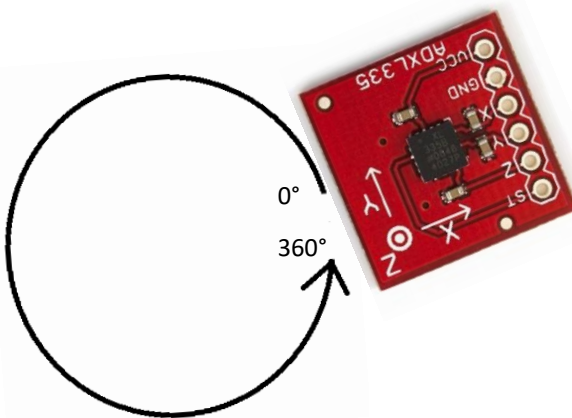


FIGURE 42: LE DECALAGE D'UN DEGRE PEUT PROVOQUER UNE DIFFERENCE DE VALEUR CONSEQUENTE

```
// Y=325 /X=177 --> -5°
//Yacc=343 / Xacc=170--> jambe droit 0° à 5 unités prêt
// Y=281/ X=180 --> jambe à 45 environ
// Y=274/ X= 170 --> 90
//Y=278 / X=177 --> 100
//Y=276 /X=181 -->110
//Y=292 X=178 --> 145
```

FIGURE 43: EXEMPLE DE RELEVÉ DE VALEURS POUR DECIDER DE L'AXE ET DE SA POSITION SUR LA JAMBE

Avec des tests nous remarquons que seul un axe

Nous avons décidé d'utiliser l'accéléromètre pour faire ralentir le moteur quand il arrive à proximité des boutons de fin de courses pour ne pas forcer sur la mécanique.

Voici des valeurs choisies pour un test (susceptible de changer) :

```
#define Pos_max_Droite 288
#define Pos_Droite_acceptable 295

#define Pos_plie_ralentir 370
#define Pos_droite_ralentir 298

#define Vitesse_max_ralentir 300
#define coef_reduc 0.3
```

On a la position maximum de la jambe tendue (Pos_max_Droite) et les positions à partir desquelles on commence à ralentir la vitesse du moteur si elle est trop élevée.

On a ainsi en dessous l'impulsion minimum à partir de laquelle la vitesse sera ralentie et le coefficient utilisé pour diminuer l'impulsion (30% choisi empiriquement).

Le code de l'accéléromètre :

```
float slow = 1.0;

void acc_use( unsigned short Y_acc, unsigned short ADC_Value)
{
    /* if( Y_acc > Pos_max_Droite){ // If we decide to stop the motor before touching the end course button
    }*/

    if (Y_acc > Pos_plie_ralentir || Y_acc < Pos_droite_ralentir) // If we get close to the maximum movement authorised
    {
        if (ADC_Value > Vitesse_max_ralentir) // Check if the speed is consequent
        {
            slow = coef_reduc; // we change this Flag (used in the function compute_movement))
        }
        else if (ADC_Value <= Vitesse_max_ralentir) // Else the speed is deemed slow enough
        {
            slow = 1.0; // The Flag is set as 1 (no change on the value)
        }
    }
    else // If there is nothing to declare
    {
        slow = 1.0; //Nothing to change
    }
}
```

Bloc bleu : Nous avons mis un flag « SLOW » à modifier lorsque l'on veut diminuer la vitesse, il sera envoyé à la fonction qui régule le mouvement selon la valeur de l'EMG. (Voir la fonction [Compute_movement](#)).

Ensuite on prend la valeur de l'ADC et la valeur de l'accéléromètre en entrée

Bloc rouge : si les valeurs de l'accéléromètre sont en dessous ou au-dessus des valeurs « acceptables » alors on vérifie ensuite que les valeurs de l'ADC de l'EMG sont en dessous ou au-dessus de la vitesse minimum acceptée.

Si la valeur envoyée par l'EMG est trop élevée alors on modifie le flag « SLOW » en lui donnant la valeur du coefficient réducteur. Si la valeur de l'EMG est assez faible, la vitesse restera tel quelle.

Enfin si les valeurs de l'accéléromètre renvoi que la jambe n'est proche d'aucun capteur de fin de course, il n'y a pas lieu d'agir sur la rapidité du mouvement.

L'idéal serait que l'ADC capte bien ces valeurs et évite même qu'on ne touche à ses boutons de fin de course (qui existent pour la sécurité de l'utilisateur) en initiant un arrêt ou un mouvement dans l'autre sens selon où l'on est et si l'on envoie un signal ou non. Mais vu que la plage de valeurs de l'accéléromètre est susceptible de se modifier entre deux essais (parce que nous n'avons pas fixé l'accéléromètre de façon permanente sur la jambe et que la mécanique n'est pas absolument stable dans sa version bêta) ce ne sera pas le cas ici. Le risque pour l'utilisateur est trop grand.

V. Calcul du mouvement

Une fois que l'on a récupéré les valeurs que renvoient les capteurs et que l'on a configuré le fonctionnement du moteur, il ne nous reste plus qu'à déterminer la direction et la vitesse que l'on donne au moteur.

Notre stratégie consiste à activer le moteur lorsqu'un signal est lu en sortie de l'EMG et si le genou n'est pas plié (détecté par un capteur de fin de course). Ensuite tant qu'il y a un signal et que le genou n'arrive pas à sa position maximale, on le laisse monter. Cependant, si le signal devient nul ou que la position maximale est atteinte, alors on inverse la direction du moteur jusqu'à ce que le genou atteigne la position de base (jambe droite) ou qu'un signal réapparaisse.

La vitesse est déterminée à partir de l'entrée du capteur EMG, plus la valeur lue est élevée, plus le moteur sera rapide. Pour cela, on envoie à la fonction « `pwm_activated` » (présentée dans la partie pwm) directement la valeur lue par le capteur. Comme ses valeurs sont comprises entre 0 et 1023, alors on donne 1023 pour valeur à l'horloge de la PWM. Ainsi, la vitesse obtenue est proportionnelle à la valeur renvoyée par l'EMG. Seulement, avec ces valeurs la fréquence vaut 4k HZ ce qui génère un bruit lors du fonctionnement du moteur, on divise donc toutes les valeurs par 6 pour obtenir une fréquence de 24k Hz, inaudible à l'oreille humaine.

Pour la vitesse de retour le fonctionnement diffère car le signal est alors nul. Pour résoudre ce problème, on prend la valeur maximum lue à l'aller puis on l'enregistre et on la renvoie pour le retour.

En plus de cela, on détermine, à l'aide de l'accéléromètre, si la jambe se trouve proche des zones maximum et minimum du mouvement et on réduit de 30% la vitesse dans ces zones.

Enfin, les positions maximum et minimum sont déterminées par les capteurs de fin de course qui change la variable globale dans une fonction d'interruption.

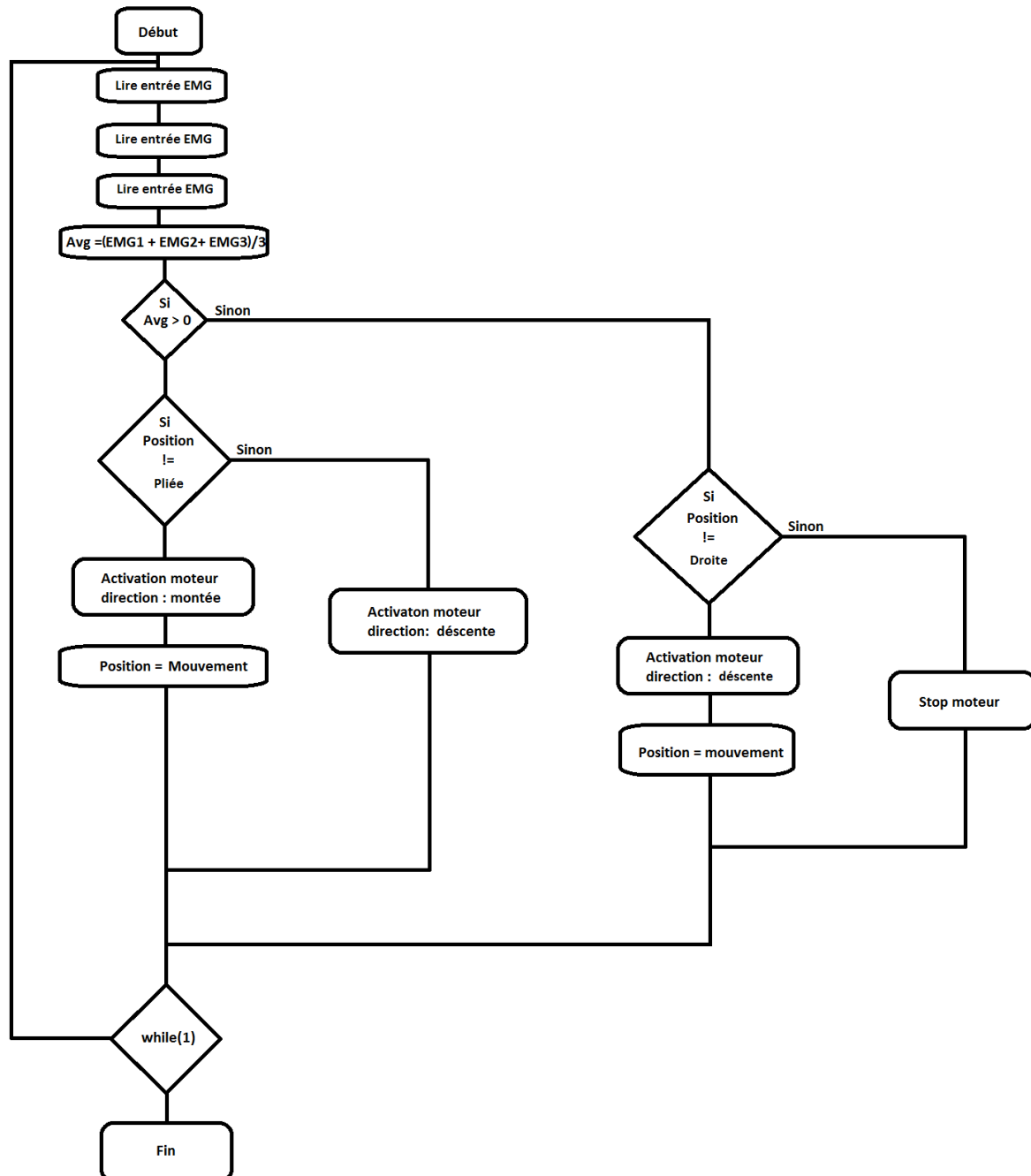


FIGURE 44 : SCHEMA DU CODE DU CALCUL DE MOUVEMENT

ANNEXES

Code :

Sommaire du code :

Main.c	43
Accelerometre.h	45
Accelerometer.c	46
Pwm.h	48
PWM.c	48
Interrupt.h	52
Interrupts.c	52
Adc.h	58
ADC.c	58
System.h	62
system.c	63
User.h	65
User.c	65
Traps.c	67
Configuration_bits.c	71

Main.c

```

/*****
***/

/* Files to Include                                     */

/*****
***/

/* Device header file */

```



```
#if defined(__XC16__)
    #include <xc.h>
#elif defined(__C30__)
    #if defined(__PIC24E__)
        #include <p24Exxxx.h>
    #elif defined (__PIC24F__)||defined (__PIC24FK__)
        #include <p24Fxxxx.h>
    #elif defined(__PIC24H__)
        #include <p24Hxxxx.h>
    #endif
#endif

#include <stdint.h>      /* Includes uint16_t definition      */
#include <stdbool.h>     /* Includes true/false definition    */

#include "system.h"      /* System funct/params, like osc/peripheral config */
#include "user.h"        /* User funct/params, such as InitApp */
#include "adc.h"
#include "interrupts.h"
#include "pwm.h"
#include "accelerometer.h"
#include <libpic30.h>

/*****
***/

/* Global Variable Declaration */

/*****
***/

unsigned short pos = 0; // Position variable Initialisation : (0 = knee straight/ 1 = movement /
2 = knee bend)
```

```

/*****
***/

/* Main Program */

/*****
***/

```

```

int main(void)
{

    unsigned short ADC_VALUE = 0, Y_acc = 0; // Adc variables Initialisation
    unsigned short freq_pwm_reverse = 500; // Speed of come back Initialisation(500 by
    default for mid speed)

    InitApp();           // PORTS Initialisation
    init_PWM();          // PWM Initialisation
    init_PWM2();         // PWM2 Initialisation
    init_ADC();          // ADC Initialisation


    while(1)
    {
        ADC_VALUE = Average_Value_ADC(); // Take an average value of the EMG captor
        Y_acc = ADC_read_accY();         // Take the value return by the accelerometer
        acc_use(Y_acc, ADC_VALUE);       // Check if it's needed to slow the motor
        freq_pwm_reverse = compute_movement(ADC_VALUE,freq_pwm_reverse); //
        Compute the direction and the speed that will be put to the motor
    }
    return 0;
}

```

Accelerometre.h

```

/*****

```

```

/* Files to Include */

```

```

*****/

```

```

/* Device header file */

```

```

#if defined(__XC16__)

```

```

    #include <xc.h>

```

```

#elif defined(__C30__)

```

```

    #if defined(__PIC24E__)

```

```

        #include <p24Exxxx.h>

```

```

    #elif defined (__PIC24F__)||defined (__PIC24FK__)

```

```

        #include <p24Fxxxh.h>

```

```

    #elif defined(__PIC24H__)

```

```

        #include <p24Hxxxh.h>

```

```

    #endif

```

```

#endif /* ACCELEROMETER_H */

```

```

void acc_use( unsigned short Y_acc,unsigned short ADC_Value);

```

Accelerometer.c

```

#include <xc.h>

```

```

#include<stdint.h> //uint32

```

```

#include<stdbool.h> //boolean

```

```

#include "adc.h"

```

```

#include "interrupts.h"

```

```

#include "pwm.h"

```

```

#include "user.h"

```

```

#include "system.h"

```

```

#include "accelerometer.h"

```

```

#define Pos_max_Droite 288

```

```

#define Pos_Droite_acceptable 295

```

```
#define Pos_plie_rallentir 370
#define Pos_droite_rallentir 298

#define Vittese_max_rallentir 300
#define coef_reduc 0.3

float slow = 1.0;

void acc_use( unsigned short Y_acc, unsigned short ADC_Value)
{

    /* if( Y_acc > Pos_max_Droite){ // If we decide to stop the motor before touching the end course button
    }*/

    if (Y_acc > Pos_plie_rallentir || Y_acc < Pos_droite_rallentir) // If we get close to the maximum movement
    autorised
    {
        if (ADC_Value > Vittese_max_rallentir)                // Check if the speed is consequent
        {
            slow = coef_reduc;                                // we change this Flag (used in the function
compute_movemen))
        }
        else if (ADC_Value <= Vittese_max_rallentir)          // Else the speed is deemed slow enough
        {
            slow = 1.0;                                        // The Flag is set as 1 (no change on the value)
        }
    }
    else                                                        // If there is nothing to declare
    {
        slow = 1.0;                                           //Nothing to change
    }
}
```

Pwm.h

```

/*****
/* Files to Include                                     */
*****/

/* Device header file */
#if defined(__XC16__)
    #include <xc.h>
#elif defined(__C30__)
    #if defined(__PIC24E__)
        #include <p24Exxxx.h>
    #elif defined (__PIC24F__)||defined (__PIC24FK__)
        #include <p24Fxxx.h>
    #elif defined(__PIC24H__)
        #include <p24Hxxx.h>
    #endif
#endif

void init_PWM(void);
void init_PWM2(void);
void PWM_activated(unsigned short direction, unsigned short vitesse);

```

PWM.c

```

/*
* File: pwm.c
* Author: Louis
*
* Created on 11 avril 2019, 17:10
*/

#include "pwm.h"

```

```
void init_PWM(void)
{

    // Set MCCR operating mode
    CCP1CON1Lbits.CCSEL = 0;        // Set MCCR operating mode (OC mode)
    CCP1CON1Lbits.MOD = 0b0101;     // Set mode (Buffered Dual-Compare/PWM mode)

    //Configure MCCR Timebase
    CCP1CON1Lbits.TMR32 = 0;        // Set timebase width (16-bit)
    CCP1CON1Lbits.TMRSYNC = 0;      // Set timebase synchronization (Synchronized)
    CCP1CON1Lbits.CLKSEL = 0b000;   // Set the clock source (Tcy)
    CCP1CON1Lbits.TMRPS = 0b00;     // Set the clock pre-scaler (1:1)
    CCP1CON1Hbits.TRIGEN = 0;       // Set Sync/Triggered mode (Synchronous)
    CCP1CON1Hbits.SYNC = 0b000000;  // Select Sync/Trigger source (Self-sync)

    //Configure MCCR output for PWM signal
    CCP1CON2Hbits.OCAEN = 1;        // Enable desired output signals (OC1A)
    CCP1CON3Hbits.OUTM = 0b000;     // Set advanced output modes (Standard output)
    CCP1CON3Hbits.POLACE = 0;       // Configure output polarity (Active High)
    CCP1TMRCL = 0x0000;            // Initialize timer prior to enable module.
    CCP1PRL = 0xFFFF;             // Configure timebase period
    CCP1RA = 0x0000;              // Set the rising edge compare value
    CCP1RB = 0xFFFF;             // Set the falling edge compare value 1023 values
    CCP1CON1Lbits.CCPON = 1;       // enable PWM1

}
```

```
void init_PWM2(void)
{

    // Set MCCR operating mode
    CCP2CON1Lbits.CCSEL = 0;        // Set MCCR operating mode (OC mode)
    CCP2CON1Lbits.MOD = 0b0101;     // Set mode (Buffered Dual-Compare/PWM mode)

    //Configure MCCR Timebase
    CCP2CON1Lbits.TMR32 = 0;        // Set timebase width (16-bit)
    CCP2CON1Lbits.TMRSYNC = 0;      // Set timebase synchronization (Synchronized)
```

```

CCP2CON1Lbits.CLKSEL = 0b000;    // Set the clock source (Tcy)
CCP2CON1Lbits.TMRPS = 0b00;      // Set the clock pre-scaler (1:1)
CCP2CON1Hbits.TRIGEN = 0;        // Set Sync/Triggered mode (Synchronous)
CCP2CON1Hbits.SYNC = 0b00000;    // Select Sync/Trigger source (Self-sync)
//Configure MCCP output for PWM signal
CCP2CON2Hbits.OCAEN = 1;         // Enable desired output signals (OC2A)
CCP2CON3Hbits.OUTM = 0b000;      // Set advanced output modes (Standard output)
CCP2CON3Hbits.POLACE = 0;        // Configure output polarity (Active High)
CCP2TMRL = 0x0000;              // Initialize timer prior to enable module.
CCP2PRL = 0xFFFF;               // Configure timebase period //1023=3ff
CCP2RA = 0x0000;                 // Set the rising edge compare value
CCP2RB = 0xFFFF;                // Set the falling edge compare value 1023 values
CCP2CON1Lbits.CCPON = 1;         // enable PWM1
}

void activate_pwm(unsigned short direction)
{
    if(direction == 0)
    {
        //on allume la PWM1 avec la fréquence "freq_pwm"
        CCP1CON1Lbits.CCPON = 1;    // On active la PWM1

        //on met la PWM2 a 5V constant
        CCP2PRL = 0x00AA;           //On réinitialise PRL (timebase period)
        CCP2RA = 0x0000;
        CCP2RB = 0x00AA;           // On donne à RA et à RB des valeurs min et max pour obtenir un signal
        constant 5V
        CCP2CON1Lbits.CCPON = 1;    // On active la PWM2
    }
    else if(direction == 1)
    {
        //on met la PWM1 a 5V constant
        CCP1PRL = 0x00AA;           //On réinitialise PRL (timebase period)
        CCP1RA = 0x0000;
        CCP1RB = 0x00AA;           // On donne à RA et à RB des valeurs min et max pour obtenir un signal
        constant 5V
        CCP1CON1Lbits.CCPON = 1;    // On active la PWM1
    }
}

```



```

//on allume la PWM2 avec la fréquence "freq_pwm"
CCP2CON1Lbits.CCPON = 1;          // On active la PWM2
}
else
{
    //on éteint les 2 PWM
    CCP2CON1Lbits.CCPON = 0;        // On desactive la PWM1
    CCP1CON1Lbits.CCPON = 0;        // On desactive la PWM2
}
}

void set_freq_pwm1(unsigned short value_ADC)
{
    CCP1PRL = 0x00AA;               //On réinitialise PRL (timebase period)
    CCP1RA = value_ADC/6; // On définit le rising time en fonction de la valeur lue par le capteur (divisé par 6
pour obtenir une fréquence supérieur à 20kHz)
    CCP1RB = 0x00AA;               // On définit le fall time
}

void set_freq_pwm2(unsigned short value_ADC)
{
    CCP2PRL = 0x00AA;               // On réinitialise PRL (timebase period)
    CCP2RA = value_ADC/6; // On définit le rising time en fonction de la valeur lue par le capteur (divisé par 6
pour obtenir une fréquence supérieur à 20kHz)
    CCP2RB = 0x00AA;               // On définit le fall time
}

```

Interrupt.h

```
void init_timer(void);

//void __attribute__((interrupt,auto_psv)) _T1Interrupt(void);
void __attribute__((interrupt,auto_psv)) _INT2Interrupt(void);
void __attribute__((interrupt,auto_psv)) _INT1Interrupt(void);
```

Interrupts.c

```

/*****
/* Files to Include */
*****/

//definition des variable que l'on vas souvent utiliser
#define Bouton_fin_cuisse IFS1bits.INT1IF
#define Bouton_fin_genou IFS1bits.INT2IF

/* Device header file */
#if defined(__XC16__)
    #include <xc.h>
#elif defined(__C30__)
    #if defined(__PIC24E__)
        #include <p24Exxxx.h>
    #elif defined (__PIC24F__)||defined (__PIC24FK__)
        #include <p24Fxxx.h>
    #elif defined(__PIC24H__)
        #include <p24Hxxx.h>
    #endif
#endif
#endif
```

```
#include <stdint.h>    /* Includes uint16_t definition */
#include <stdbool.h>    /* Includes true/false definition */
#include "pwm.h"

/*****

/* Interrupt Vector Options */

*****/

/*

*/

/* Refer to the C30 (MPLAB C Compiler for PIC24F MCUs and dsPIC33F DSCs) User */
/* Guide for an up to date list of the available interrupt options. */
/* Alternately these names can be pulled from the device linker scripts. */
/*

*/

/* PIC24F Primary Interrupt Vector Names: */
/*

*/

/* _INT0Interrupt    _IC4Interrupt */
/* _IC1Interrupt    _IC5Interrupt */
/* _OC1Interrupt    _IC6Interrupt */
/* _T1Interrupt    _OC5Interrupt */
/* _Interrupt4    _OC6Interrupt */
/* _IC2Interrupt    _OC7Interrupt */
/* _OC2Interrupt    _OC8Interrupt */
/* _T2Interrupt    _PMPInterrupt */
/* _T3Interrupt    _SI2C2Interrupt */
/* _SPI1ErrInterrupt    _MI2C2Interrupt */
/* _SPI1Interrupt    _INT3Interrupt */
/* _U1RXInterrupt    _INT4Interrupt */
/* _U1TXInterrupt    _RTCCInterrupt */
/* _ADC1Interrupt    _U1ErrInterrupt */
/* _SI2C1Interrupt    _U2ErrInterrupt */
/* _MI2C1Interrupt    _CRCInterrupt */
/* _CompInterrupt    _LVDInterrupt */
/* _CNInterrupt    _CTMUInterrupt */
/* _INT1Interrupt    _U3ErrInterrupt */
/* _IC7Interrupt    _U3RXInterrupt */
/* _IC8Interrupt    _U3TXInterrupt */
/* _OC3Interrupt    _SI2C3Interrupt */
```

```

/* _OC4Interrupt    _MI2C3Interrupt                */
/* _T4Interrupt     _U4ErrInterrupt                */
/* _T5Interrupt     _U4RXInterrupt                */
/* _INT2Interrupt   _U4TXInterrupt                */
/* _U2RXInterrupt   _SPI3ErrInterrupt              */
/* _U2TXInterrupt   _SPI3Interrupt                 */
/* _SPI2ErrInterrupt _OC9Interrupt                 */
/* _SPI2Interrupt   _IC9Interrupt                  */
/* _IC3Interrupt                      */
/*                                */
/* PIC24H Primary Interrupt Vector Names:          */
/*                                */
/* _INT0Interrupt   _SPI2Interrupt                */
/* _IC1Interrupt    _C1RxRdyInterrupt              */
/* _OC1Interrupt    _C1Interrupt                   */
/* _T1Interrupt     _DMA3Interrupt                 */
/* _DMA0Interrupt   _IC3Interrupt                  */
/* _IC2Interrupt    _IC4Interrupt                  */
/* _OC2Interrupt    _IC5Interrupt                  */
/* _T2Interrupt     _IC6Interrupt                  */
/* _T3Interrupt     _OC5Interrupt                  */
/* _SPI1ErrInterrupt _OC6Interrupt                 */
/* _SPI1Interrupt   _OC7Interrupt                  */
/* _U1RXInterrupt   _OC8Interrupt                 */
/* _U1TXInterrupt   _DMA4Interrupt                 */
/* _ADC1Interrupt   _T6Interrupt                   */
/* _DMA1Interrupt   _T7Interrupt                   */
/* _SI2C1Interrupt  _SI2C2Interrupt                */
/* _MI2C1Interrupt  _MI2C2Interrupt                */
/* _CNInterrupt     _T8Interrupt                   */
/* _INT1Interrupt   _T9Interrupt                   */
/* _ADC2Interrupt   _INT3Interrupt                 */
/* _IC7Interrupt    _INT4Interrupt                 */
/* _IC8Interrupt    _C2RxRdyInterrupt              */
/* _DMA2Interrupt   _C2Interrupt                   */
/* _OC3Interrupt    _DCIErrInterrupt               */
/* _OC4Interrupt    _DCIInterrupt                  */

```

```

/* _T4Interrupt    _U1ErrInterrupt          */
/* _T5Interrupt    _U2ErrInterrupt          */
/* _INT2Interrupt  _DMA6Interrupt           */
/* _U2RXInterrupt  _DMA7Interrupt           */
/* _U2TXInterrupt  _C1TxReqInterrupt        */
/* _SPI2ErrInterrupt _C2TxReqInterrupt       */
/*                                     */
/* PIC24E Primary Interrupt Vector Names:   */
/*                                     */
/* __INT0Interrupt  __C1RxRdyInterrupt  __U3TXInterrupt          */
/* __IC1Interrupt   __C1Interrupt        __USB1Interrupt         */
/* __OC1Interrupt   __DMA3Interrupt       __U4ErrInterrupt        */
/* __T1Interrupt    __IC3Interrupt        __U4RXInterrupt         */
/* __DMA0Interrupt  __IC4Interrupt        __U4TXInterrupt         */
/* __IC2Interrupt   __IC5Interrupt        __SPI3ErrInterrupt      */
/* __OC2Interrupt   __IC6Interrupt        __SPI3Interrupt         */
/* __T2Interrupt    __OC5Interrupt        __OC9Interrupt          */
/* __T3Interrupt    __OC6Interrupt        __IC9Interrupt          */
/* __SPI1ErrInterrupt __OC7Interrupt       __DMA8Interrupt        */
/* __SPI1Interrupt  __OC8Interrupt        __DMA9Interrupt         */
/* __U1RXInterrupt  __PMPInterrupt        __DMA10Interrupt        */
/* __U1TXInterrupt  __DMA4Interrupt       __DMA11Interrupt        */
/* __AD1Interrupt   __T6Interrupt         __SPI4ErrInterrupt      */
/* __DMA1Interrupt  __T7Interrupt         __SPI4Interrupt         */
/* __NVMInterrupt   __SI2C2Interrupt      __OC10Interrupt         */
/* __SI2C1Interrupt __MI2C2Interrupt      __IC10Interrupt         */
/* __MI2C1Interrupt __T8Interrupt         __OC11Interrupt         */
/* __CM1Interrupt   __T9Interrupt         __IC11Interrupt         */
/* __CNInterrupt    __INT3Interrupt       __OC12Interrupt         */
/* __INT1Interrupt  __INT4Interrupt       __IC12Interrupt         */
/* __AD2Interrupt   __C2RxRdyInterrupt    __DMA12Interrupt        */
/* __IC7Interrupt   __C2Interrupt         __DMA13Interrupt        */
/* __IC8Interrupt   __DMA5Interrupt       __DMA14Interrupt        */
/* __DMA2Interrupt  __RTCCInterrupt       __OC13Interrupt         */
/* __OC3Interrupt   __U1ErrInterrupt      __IC13Interrupt         */
/* __OC4Interrupt   __U2ErrInterrupt      __OC14Interrupt         */
/* __T4Interrupt    __CRCInterrupt        __IC14Interrupt         */

```

```

/* __T5Interrupt    __DMA6Interrupt    __OC15Interrupt    */
/* __INT2Interrupt  __DMA7Interrupt    __IC15Interrupt    */
/* __U2RXInterrupt  __C1TxReqInterrupt  __OC16Interrupt    */
/* __U2TXInterrupt  __C2TxReqInterrupt  __IC16Interrupt    */
/* __SPI2ErrInterrupt __U3ErrInterrupt    __ICDIInterrupt    */
/* __SPI2Interrupt  __U3RXInterrupt      */
/*
/*
/* For alternate interrupt vector naming, simply add 'Alt' between the prim. */
/* interrupt vector name '_' and the first character of the primary interrupt */
/* vector name. There are no Alternate or 'Alt' vectors for the 24E family. */
/*
/* For example, the vector name __ADC2Interrupt becomes __AltADC2Interrupt in
/* the alternate vector table.
/*
/* Example Syntax:
/*
/* void __attribute__((interrupt,auto_psv)) <Vector Name>(void)
/* {
/*     <Clear Interrupt Flag>
/* }
/*
/* For more comprehensive interrupt examples refer to the C30 (MPLAB C
/* Compiler for PIC24 MCUs and dsPIC DSCs) User Guide in the
/* <compiler installation directory>/doc directory for the latest compiler
/* release.
/*
/*****
/* Interrupt Routines
/*****

/* TODO Add interrupt routine code here. */
// datasheet: ?Timers? (DS39704)
void init_timer(void)
{
    T1CON = 0x00;           //stops 16bits timer1, uses the Secondary Oscillator (SOSC) as the clock
                             source, Prescale 1:1, Internal clock (FOSC/2)

```

```

TMR1= 0x00;                //clear content of the timer1 register
PR1 = 0x0FFFF;             //the period register with the
IPC0bits.T1IP = 0x01;       //interrupt priority control 0 --> Interrupt is Priority 1
IFS0bits.T1IF =0;           //interrupt flag status 0 -Timer1 Interrupt Flag Status bitb 0= Interrupt
request has not occurred
IEC0bits.T1IE = 1;          // INTERRUPT ENABLE CONTROL REGISTER 0 -Timer1 Interrupt
Enable bit 1= Interrupt request is enabled

T1CONbits.TON = 0;          //Start Timer1 with prescaler settings at 1:1 etc
}

/*
volatile unsigned short counter;
void __attribute__((interrupt,auto_psv)) _T1Interrupt(void)
{
    // Interrupt Service Routine code goes here
    //Poll the T0IF flag to see if TMR0 has overflowed

    ++counter;               //if T0IF = 1 increment the counter variable by 1
    IFS0bits.T1IF = 0;        //Clear the T0IF flag so that the next overflow can be detected ( Reset
Timer1 interrupt flag and Return from ISR)

}
*/

extern unsigned short pos;
void __attribute__((interrupt,auto_psv))_INT2Interrupt(void)
{
    pos=0;
    Bouton_fin_genou=0;
}

void __attribute__((interrupt,auto_psv))_INT1Interrupt(void)
{
    pos=2;
    Bouton_fin_cuisse=0;
}

```


Adc.h

```

/*****
/* Files to Include
*****/

/* Device header file */
#if defined(__XC16__)
    #include <xc.h>
#elif defined(__C30__)
    #if defined(__PIC24E__)
        #include <p24Exxxx.h>
    #elif defined (__PIC24F__)||defined (__PIC24FK__)
        #include <p24Fxxx.h>
    #elif defined(__PIC24H__)
        #include <p24Hxxx.h>
    #endif
#endif
#endif

```

```

void init_ADC(void);
void Delay (unsigned short temp);
unsigned short ADC_read(void);
unsigned short ADC_read_accY(void);

unsigned short Average_Value_ADC(void);
unsigned short compare(unsigned short now,unsigned short max);
unsigned short ADC_seuil_changement(unsigned short ADC_Value);
unsigned short compute_movement(unsigned short value_read,unsigned short freq_pwm_reverse);

```

ADC.c

```

#include <xc.h>
#include<stdint.h> //uint32
#include<stdbool.h> //boolean

```

```
#include "adc.h"
#include "interrupts.h"
#include "pwm.h"
#include "user.h"
#include "system.h"
#include "accelerometer.h"

#include<libpic30.h>

#define value_min 50
#define leg_ascend 0
#define leg_down 1
#define stop 2

#define pos_straight 0
#define pos_mov 1
#define pos_bend 2

extern unsigned short pos;
extern float slow;

void init_ADC(void)
{

    AD1CON1bits.ADSIDL= 0;           // Don't STOP IN IDLE MODE
    AD1CON1bits.ASAM= 0;             // Not auto-start
    AD1CON1bits.MODE12= 0;           // Fct en 10bit, 12bits pas utile
    AD1CON1bits.FORM = 0b00;         // Format bit: int
    AD1CON1bits.SSRC = 0b000;        // manual convert trigger mode
    AD1CON2 = 0;                     // Configure A/D voltage reference
                                    // Vr+ and Vr- from AVdd and AVss(PVCFG<1:0>=00, NVCFG=0),
                                    // Inputs are not scanned,
                                    // Interrupt after every sample
```

```

AD1CON3 = 0x0002;          // Sample time: Tad = 3Tcy
                            // A/D conversion clock as Tcy

AD1CHS= 0x0010;           // --> the input is the positive input 0x10=16
AD1CSSL = 0;              // No inputs are scanned.
}

unsigned short ADC_read(void)
{
    AD1CHSbits.CH0SA = 16;    // Configure input channels on input AN16 pin12 (1000)
    AD1CON1bits.ADON = 1;     // A/D CONVERTER IS OPERATING
    AD1CON1bits.SAMP = 1;     // Start sampling the input
    __delay_ms(6);           // Delay assigned empirically: tad min de 12cycles --> tad= 3TCY so 31TAD

    AD1CON1bits.SAMP = 0;     // Stop sampling and Start converting

    while(!AD1CON1bits.DONE){}; // While conversion not complet
    unsigned short ADC_VALUE= ADC1BUF0; // The value sampled and converted is recolted

    AD1CON1bits.ADON = 0;     // A/D CONVERTER IS NOT OPERATING
    return ADC_VALUE;        // We return the value read
}

unsigned short ADC_read_accY(void)
{
    unsigned short Y_acc;
    AD1CHSbits.CH0SA = 18;    // Configure input channels on input AN18 pin15 (1010)
    AD1CON1bits.ADON = 1;     // A/D CONVERTER IS OPERATING
    AD1CON1bits.SAMP = 1;     // Start sampling the input
    __delay_ms(6);

    AD1CON1bits.SAMP = 0;     // Stop sampling and Start converting

    while(!AD1CON1bits.DONE){}; // While conversion not complet
    Y_acc= ADC1BUF0;         // We return the value read

    AD1CON1bits.ADON = 0;     // A/D CONVERTER IS NOT OPERATING

```

```

return Y_acc;
}

unsigned short Average_Value_ADC(void)
{
    unsigned short value1, value2, value3, value_average;

    value1=ADC_read();
    value2=ADC_read();           // 3 values are read from the EMG
    value3=ADC_read();

    value_average = (value1+value2+value3)/3; // We do an average of those value to smooth out problems
    linked to noise

    return value_average;        // Return the average
}

unsigned short compute_movement(unsigned short value_read,unsigned short freq_pwm_reverse)
{
    unsigned short freq_pwm;      // We define the value of the PWM's frequency (so the motor's speed)
    when the knee bends and this will be associated to the RA of the PWM

    freq_pwm = value_read;        // We give to the frequency of the PWM the value read by the ADC

    // If the value read is equal or above the value_min (so if the signal is consequent enough to move)
    if (value_read >= value_min)
    {
        freq_pwm_reverse = compare(freq_pwm, freq_pwm_reverse); // We check that freq_pwm_reverse contain
        the max value

        if(pos != pos_bend)      // If the leg isn't on position 2 (doesn't touch the end course button
        with the knee bent)
        {
            PWM_activated(leg_ascend, freq_pwm/slow); // We activate the pwm in the direction "knee bent"
            pos = pos_mov;      // We set the position's variable to 1 (in movement)
        }
    }
}

```

```

        else if(pos == pos_bend)                // If the leg is on position 2 (touch the endcourse button with the
knee bent)
        {
            PWM_activated(leg_down, freq_pwm_reverse/slow); // we activate the pwm in the sens "get straight"
            and set the frequency of the pwm1 to the max frequency of the previous movement
        }
    }

    // If the value read is under the value_min (so if the signal isn't consequent enough to move)
    else if(value_read < value_min)
    {

        if(pos != pos_straight)                // If the position isn't on 0 (knee isn't straight)
        {
            PWM_activated(leg_down, freq_pwm_reverse/slow); // We activate the pwm in the direction "knee
going straight"
            pos = pos_mov;                      // We set the position's variable to 1 (in movement)
        }
        else if(pos == pos_straight)            // If the position is on 0 (knee is straight)
        {
            PWM_activated(stop, 0);             // We stop both pwm (and so the motor)
            freq_pwm_reverse = 500;             // We reset the come back speed
        }
    }
    return freq_pwm_reverse;                   // We return the come back speed
}

```

```

unsigned short compare(unsigned short now,unsigned short max) // Return the bigger value
{
    if (now > max)
        return now;
    else
        return max;
}

```

System.h

```

/*****

```

```

/* System Level #define Macros */
/*****

/* TODO Define system operating frequency */

/* Microcontroller MIPs (FCY) */
#define SYS_FREQ      8000000L
#define FCY           SYS_FREQ/2

/*****

/* System Function Prototypes */
/*****

/* Custom oscillator configuration funtions, reset source evaluation
functions, and other non-peripheral microcontroller initialization functions
go here. */
#include<stdint.h> //uint32
#include<stdbool.h> //boolean
#include <libpic30.h>
void ConfigureOscillator(void); /* Handles clock switching/osc initialization */

                                     system.c
/*****

/* Files to Include */
/*****

/* Device header file */
#if defined(__XC16__)
    #include <xc.h>
#elif defined(__C30__)
    #if defined(__PIC24E__)
        #include <p24Exxxx.h>
    #elif defined (__PIC24F__)||defined (__PIC24FK__)
        #include <p24Fxxx.h>
    #elif defined(__PIC24H__)
        #include <p24Hxxx.h>

```

```

#endif

#endif

#include <stdint.h>    /* For uint32_t definition */
#include <stdbool.h>    /* For true/false definition */

#include "system.h"    /* variables/params used by system.c */

/*****

/* System Level Functions                                */
/*
/* Custom oscillator configuration funtions, reset source evaluation */
/* functions, and other non-peripheral microcontroller initialization */
/* functions get placed in system.c                        */
/*
*****/

/* Refer to the device Family Reference Manual Oscillator section for
information about available oscillator configurations. Typically
this would involve configuring the oscillator tuning register or clock
switching using the compiler's __builtin_write_OSCCON functions.
Refer to the C Compiler for PIC24 MCUs and dsPIC DSCs User Guide in the
compiler installation directory /doc folder for documentation on the
__builtin functions. */

/* TODO Add clock switching code if appropriate. An example stub is below. */
void ConfigureOscillator(void)
{

#if 0

/* Disable Watch Dog Timer */
RCONbits.SWDTEN = 0;

/* When clock switch occurs switch to Prim Osc (HS, XT, EC)with PLL */
__builtin_write_OSCCONH(0x03); /* Set OSCCONH for clock switch */
__builtin_write_OSCCONL(0x01); /* Start clock switching */

```



```
while(OSCCONbits.COSC != 0b011);

/* Wait for Clock switch to occur */
/* Wait for PLL to lock, if PLL is used */
/* while(OSCCONbits.LOCK != 1); */

#endif

}
```

User.h

```

/*****
    */
    /* User Level #define Macros
    */
    */
    */

/* TODO Application specific user parameters used in user.c may go here */

/*****
    */
    /* User Function Prototypes
    */
    */
    */

/* TODO User level functions prototypes (i.e. InitApp) go here */

void InitApp(void);    /* I/O and Peripheral Initialization */

```

User.c

```

/*****
    */
    /* Files to Include
    */
    */

/* Device header file */
#if defined(__XC16__)
    #include <xc.h>
#elif defined(__C30__)
    #if defined(__PIC24E__)
        #include <p24Exxxx.h>
    #elif defined (__PIC24F__)||defined (__PIC24FK__)
        #include <p24Fxxx.h>
    #endif

```

```
#elif defined(__PIC24H__)
    #include <p24Hxxxx.h>
#endif
#endif

#include <stdint.h>    /* For uint32_t definition */
#include <stdbool.h>    /* For true/false definition */

#include "user.h"      /* variables/params used by user.c */

/*****

/* User Functions                                     */
*****/

/* <Initialize variables in user.h and insert code for user algorithms.> */

/* TODO Initialize User Ports/Peripherals/Project here */

void InitApp(void)
{
    /* Setup analog functionality and port direction */

    TRISBbits.TRISB12 = 1;    // Set RB12 as Input
    ANSBbits.ANSB12 = 0;     // Set A0 as digital IO
    INTCON2bits.INT2EP = 0;   // Rising edge
    IFS1bits.INT2IF = 0;      // Clear INT0 Interrupt
    IEC1bits.INT2IE = 1;      // enable INT0 Interrupt
    IPC7bits.INT2IP2 = 1;     // Priority
    IPC7bits.INT2IP1 = 1;
    IPC7bits.INT2IP0 = 0;

    TRISBbits.TRISB14 = 1;    // Set RB14 as Input
    ANSBbits.ANSB14 = 0;     // Set A0 as digital IO
    INTCON2bits.INT1EP = 0;   // Rising edge
    IFS1bits.INT1IF = 0;      // Clear INT0 Interrupt
    IEC1bits.INT1IE = 1;      // enable INT0 Interrupt
    IPC5bits.INT1IP2 = 1;     // Priority
```

```

IPC5bits.INT1IP1 = 1;
IPC5bits.INT1IP0 = 1;


ANSAbits.ANSA0 = 0;    // Set A0 as digital IO
TRISAbits.TRISA0 = 0;  // Set A0 as output
TRISBbits.TRISB7 = 0;  // Set B7 as output
TRISBbits.TRISB11 = 0; // Set B11 as output


//ADC
//EMG
TRISAbits.TRISA4 = 1;  // Set RA4 as input --> pin 12 an16
ANSAbits.ANSA4 = 1;    // Set RA4as analog IO


//accelerometer
TRISBbits.TRISB6 = 1;  // Set RB6 as input --> pin 15 an18
ANSBbits.ANSB6 = 1;    // Set RB6 as analog IO
TRISBbits.TRISB8 = 1;  // Set RB8 as input --> pin 17 an20
ANSBbits.ANSB8 = 1;    // Set RB8 as analog IO
TRISBbits.TRISB9 = 1;  // Set RA4 as input -->pin 18 an21
ANSBbits.ANSB9 = 1;    // Set RA4as analog IO


/* Initialize peripherals */
}


```

Traps.c

```

/*****
/* Files to Include
*/
*****/

/* Device header file */
#if defined(__XC16__)
#include <xc.h>
#elif defined(__C30__)
#if defined(__PIC24E__)

```

```

#include <p24Exxxx.h>

#elif defined (__PIC24F__)||defined (__PIC24FK__)

#include <p24Fxxx.h>

#elif defined(__PIC24H__)

#include <p24Hxxx.h>

#endif

#endif

#include <stdint.h>    /* Includes uint16_t definition */
#include <stdbool.h>    /* Includes true/false definition */

/*****
/* Trap Function Prototypes */
*****/

/* <Other function prototypes for debugging trap code may be inserted here> */

/* Use if INTCON2 ALTIVT=1 */
void __attribute__((interrupt,no_auto_psv)) _OscillatorFail(void);
void __attribute__((interrupt,no_auto_psv)) _AddressError(void);
void __attribute__((interrupt,no_auto_psv)) _StackError(void);
void __attribute__((interrupt,no_auto_psv)) _MathError(void);

#if defined(__PIC24F__)||defined(__PIC24H__)

/* Use if INTCON2 ALTIVT=0 */
void __attribute__((interrupt,no_auto_psv)) _AltOscillatorFail(void);
void __attribute__((interrupt,no_auto_psv)) _AltAddressError(void);
void __attribute__((interrupt,no_auto_psv)) _AltStackError(void);
void __attribute__((interrupt,no_auto_psv)) _AltMathError(void);

#endif

/* Default interrupt handler */
void __attribute__((interrupt,no_auto_psv)) _DefaultInterrupt(void);

#if defined(__PIC24E__)

```

```

/* These are additional traps in the 24E family. Refer to the PIC24E
migration guide. There are no Alternate Vectors in the 24E family. */
void __attribute__((interrupt,no_auto_psv)) _HardTrapError(void);
void __attribute__((interrupt,no_auto_psv)) _DMACError(void);
void __attribute__((interrupt,no_auto_psv)) _SoftTrapError(void);

#endif

/*****
/* Trap Handling */
/*
/* These trap routines simply ensure that the device continuously loops */
/* within each routine. Users who actually experience one of these traps */
/* can add code to handle the error. Some basic examples for trap code, */
/* including assembly routines that process trap sources, are available at */
/* www.microchip.com/codeexamples */
*****/

/* Primary (non-alternate) address error trap function declarations */
void __attribute__((interrupt,no_auto_psv)) _OscillatorFail(void)
{
    INTCON1bits.OSCFAIL = 0;    /* Clear the trap flag */
    while(1);
}

void __attribute__((interrupt,no_auto_psv)) _AddressError(void)
{
    INTCON1bits.ADDRERR = 0;    /* Clear the trap flag */
    while (1);
}

void __attribute__((interrupt,no_auto_psv)) _StackError(void)
{
    INTCON1bits.STKERR = 0;    /* Clear the trap flag */
    while (1);
}

```

```

void __attribute__((interrupt,no_auto_psv)) _MathError(void)
{
    INTCON1bits.MATHERR = 0;    /* Clear the trap flag */
    while (1);
}

#ifdef __PIC24F__ || defined(__PIC24H__)

/* Alternate address error trap function declarations */
void __attribute__((interrupt,no_auto_psv)) _AltOscillatorFail(void)
{
    INTCON1bits.OSCFAIL = 0;    /* Clear the trap flag */
    while (1);
}

void __attribute__((interrupt,no_auto_psv)) _AltAddressError(void)
{
    INTCON1bits.ADDRERR = 0;    /* Clear the trap flag */
    while (1);
}

void __attribute__((interrupt,no_auto_psv)) _AltStackError(void)
{
    INTCON1bits.STKERR = 0;    /* Clear the trap flag */
    while (1);
}

void __attribute__((interrupt,no_auto_psv)) _AltMathError(void)
{
    INTCON1bits.MATHERR = 0;    /* Clear the trap flag */
    while (1);
}

#endif

/*****
/* Default Interrupt Handler */

```

```

/*                                     */

/* This executes when an interrupt occurs for an interrupt source with an */
/* improperly defined or undefined interrupt handling routine. */
/*****/

void __attribute__((interrupt,no_auto_psv)) _DefaultInterrupt(void)
{
    while(1);
}

#if defined(__PIC24E__)

/* These traps are new to the PIC24E family. Refer to the device Interrupt
chapter of the FRM to understand trap priority. */
void __attribute__((interrupt,no_auto_psv)) _HardTrapError(void)
{
    while(1);
}
void __attribute__((interrupt,no_auto_psv)) _DMACError(void)
{
    while(1);
}
void __attribute__((interrupt,no_auto_psv)) _SoftTrapError(void)
{
    while(1);
}

#endif

```

Configuration bits.c

```

/*****/

/* Files to Include */
/*****/

/* Device header file */
#if defined(__XC16__)
    #include <xc.h>

```

```
#elif defined(__C30__)
    #if defined(__PIC24E__)
        #include <p24Exxxx.h>
    #elif defined (__PIC24F__)||defined (__PIC24FK__)
        #include <p24Fxxx.h>
    #elif defined(__PIC24H__)
        #include <p24Hxxx.h>
    #endif
#endif

/*****

/* Configuration Bits */
/*
/*
/* This is not all available configuration bits for all PIC24 devices. */
/* Refer to the PIC24 device specific .h file in the compiler */
/* support\PIC24x\h (x=F,H,E) directory for complete options specific to the */
/* selected device. For additional information about what the hardware */
/* configurations mean in terms of device operation, refer to the device */
/* datasheet 'Special Features' chapter. */
/*
/*
/* A feature of MPLAB X is the 'Generate Source Code to Output' utility in */
/* the Configuration Bits window. Under Window > PIC Memory Views > */
/* Configuration Bits, a user controllable configuration bits window is */
/* available to Generate Configuration Bits source code which the user can */
/* paste into this project. */
*****/

/* TODO Fill in your configuration bits from the config bits generator here. */

// PIC24FV16KM202 Configuration Bit Settings

// 'C' source line config statements

// FBS
#pragma config BWRP = OFF           // Boot Segment Write Protect (Disabled)
#pragma config BSS = OFF            // Boot segment Protect (No boot program flash segment)
```



```
// FGS
#pragma config GWRP = OFF          // General Segment Write Protect (General segment may be written)
#pragma config GCP = OFF          // General Segment Code Protect (No Protection)

// FOSCSEL
#pragma config FNOSC = FRC         // Oscillator Select (Fast RC Oscillator (FRC))
#pragma config SOSCSRC = ANA      // SOSC Source Type (Analog Mode for use with crystal)
#pragma config LPRCSEL = HP       // LPRC Oscillator Power and Accuracy (High Power, High Accuracy Mode)
#pragma config IESO = ON          // Internal External Switch Over bit (Internal External Switchover mode enabled (Two-speed Start-up enabled))

// FOSC
#pragma config POSCMOD = NONE      // Primary Oscillator Configuration bits (Primary oscillator disabled)
#pragma config OSCIOFNC = CLKO     // CLKO Enable Configuration bit (CLKO output signal enabled)
#pragma config POSCFREQ = HS       // Primary Oscillator Frequency Range Configuration bits (Primary oscillator/external clock input frequency greater than 8MHz)
#pragma config SOSSEL = SOSCHP    // SOSC Power Selection Configuration bits (Secondary Oscillator configured for high-power operation)
#pragma config FCKSM = CSDCMD     // Clock Switching and Monitor Selection (Both Clock Switching and Fail-safe Clock Monitor are disabled)

// FWDT
#pragma config WDTPS = PS32768    // Watchdog Timer Postscale Select bits (1:32768)
#pragma config FWPSA = PR128      // WDT Prescaler bit (WDT prescaler ratio of 1:128)
#pragma config FWDTEN = ON        // Watchdog Timer Enable bits (WDT enabled in hardware)
#pragma config WINDIS = OFF       // Windowed Watchdog Timer Disable bit (Standard WDT selected(windowed WDT disabled))

// FPOR
#pragma config BOREN = BOR3       // Brown-out Reset Enable bits (Brown-out Reset enabled in hardware, SBOREN bit disabled)
#pragma config RETCFG = OFF       // (Retention regulator is not available)
#pragma config PWRTEN = ON        // Power-up Timer Enable bit (PWRT enabled)
#pragma config I2C1SEL = PRI      // Alternate I2C1 Pin Mapping bit (Use Default SCL1/SDA1 Pins For I2C1)
#pragma config BORV = V18         // Brown-out Reset Voltage bits (Brown-out Reset set to lowest voltage (1.8V))
#pragma config MCLRE = ON         // MCLR Pin Enable bit (RA5 input pin disabled, MCLR pin enabled)
```

```
// FICD
```

```
#pragma config ICS = PGx1          // ICD Pin Placement Select bits (EMUC/EMUD share PGC1/PGD1)
```

```
// #pragma config statements should precede project file includes.
```


```
// Use project enums instead of #define for ON and OFF.
```

```
#include <xc.h>
```


Annexe mécanique

LISTE BOM V1					
Ilance les articles à réapprovisionner ? Oui					
Référence	Nom	Description	Prix unitaire	Quantité voulue	Valeur de stock
part number: 0 390 257 693	Motoréducteur courant continu CHP	24V, 3,5A, 57W, 90rpm, 4.5Nm, 27.5:1 -datasheet (p6): https://www.kelvingear.com/ftp/productsFiles/1120/Bosch.CHP.EN.pdf	75€ HT	1	96.8 TTC
UP-VW1220P1	PANASONIC, Batterie rechargeable, 12V, 4 Ah	pic de 12A, fct à 4Ah, 12V, 1,35Kg -datasheet: http://www.farnell.com/datasheets/2095059.pdf	25,9€ TTC	2	58,80 €
892-92+C8:H852	Réducteur Portescap Engrenage droit	8:1, 0.7Nm, 5000rpm	118,40 €	1	

FIGURE 5: LISTE BOM V1, LES ELEMENTS ELIMINES DANS LA V2.



Technical data

Part number	0 390 257 693
Nominal voltage	U_N 24 V
Nominal power	P_N 57 W
Nominal current	I_N 3,5 A
Maximum current	I_{max} 16 A
Nominal speed	n_N 90 min ⁻¹
Nominal torque	M_N 4,5 Nm
Breakaway torque	M_A 16 Nm
Reduction	i 52 : 2
Direction of rotation	L/R
Type of duty	S 1
Degree of protection	IP 23
Weight	approx. 1,10 kg

Clockwise (-) at gn terminal (green)
 Anti-clockwise (+) to red, (-) to green.

FIGURE 6: MOTOREDUCTEUR CC CHP ET SES CARACTERISTIQUES

Le moteur était prévu en 24V, Plus puissant il demandait deux batteries de 12V de plus d'1kg chacune. Mais le moteur pouvait supporter jusqu'à 10kg à bout de bras (40cm de longueur).

Ce moteur était optimal pour les besoins que nous avons calculés :

Pour une structure de 10kg à supporter et une distance poids-moteur de 0.2m on a donc un couple de 19,62Nm et d'une vitesse estimée en étudiant nos mouvements de genoux de 13.33 RPM max

Le cahier des charges correspondant à nos premières attentes ci-dessous.

Cahier des Charges genoux Héraclès V1

Contexte :

Dans le cadre d'un projet inter-majeur, nous pensons développer un exosquelette nommé Héraclès avec une équipe en mécatronique et une équipe en systèmes embarqués.

Héraclès serait une paire de jambe d'exosquelette comparables à celles du projet Hercule. Les jambes doivent suivre le mouvement de l'utilisateur et alléger ses charges. L'exosquelette doit pouvoir soutenir une charge de 100kg.

Dans l'optique de ce projet nous allons construire un genou de l'exosquelette.

Problématique :

Le genou d'Héraclès doit donc pouvoir soutenir une charge de 10kg, détecter le mouvement du muscle de la cuisse de l'utilisateur et ainsi suivre/copier les mouvements naturels de la jambe (pression minimum sur la jambe humaine).

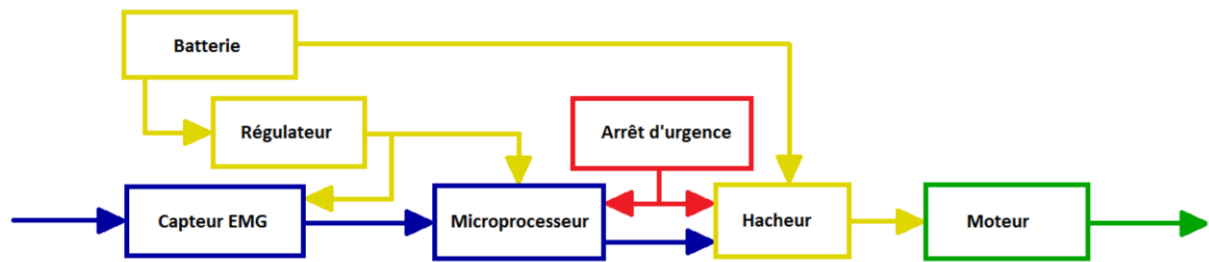
Durant le fonctionnement :

Fonction de service	Fonctions techniques	Critères	Niveaux	Caractéristiques	Flexibilité
Capacité de mouvement	Rapidité de détection du mouvement	Faible latence de détection du mouvement	2x Electrode musculaire		xxx
	Ergonomie	Accompagnement du mouvement	Moteur au genou	12V 0.44Nm 0.98Nm	Quelques ms Quelques mm
Assurer les charges	Renfort	Capacité de charge supplémentaire sans efforts additionnel	10kg	compris dans les caractéristiques moteur	1kg
	Support	Poids ressenti sur les membres postérieurs	1kg		0.5KG

Alimentation	Autonomie	Durée de fonctionnement à vide	1h	12V 7A 60Ah	30min
	Réutilisabilité	Rechargeable		Batterie Lithium	300fois
Protéger l'utilisateur	Protéger contre des erreurs de programmes	Bouton stop *	Arrêt mécanique (coupe la liaison contrôleur/moteur)	Bouton poussoir	Latence nulle
	Protéger contre le dysfonctionnement	Angles conforme aux articulations humaines		Bloquer à l'angle maximum genou : 0°-160°	Aucune
Utilisation des Commandes	Ergonomie	Simplicité d'accès aux commandes			
Être confortable	Nuisance sonore	Bruit acceptable durant le fonctionnement	48dB	Moteur Vitesse limité à 18tr/min	2db
	Position agréable	. Rigueur des supports . Forme des supports	Sangles à scratch		

* Point d'amélioration : Le moteur garde la dernière position en mémoire et retourne au repos progressivement

Schéma synoptique :



Il y a possibilité de faire le système avec deux moteurs au-dessus et en-dessous du genou avec des vis sans fin.

Schéma fonctionnel :

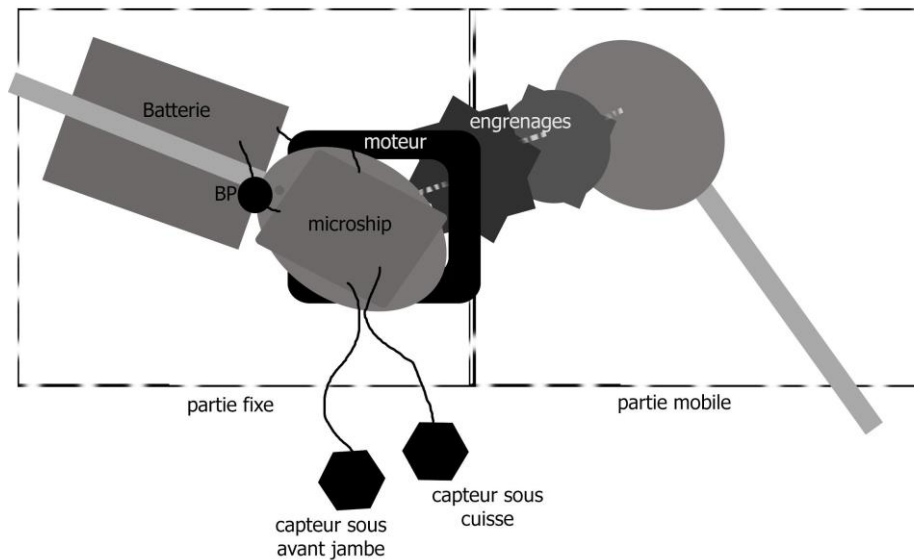
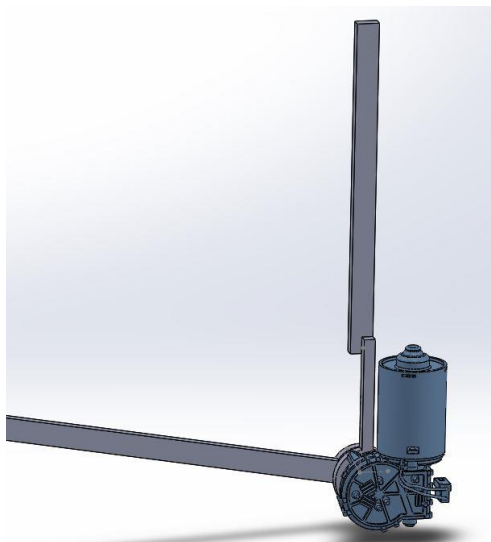


Schéma fonctionnel 2, partie moteur :



Sources :

IMAGES :

[HTTP://TPEHOMMEAugMENTE.E-MONSITE.COM/PAGES/PARTIE-2/LES-EXOSQUELETTES-MOTORISES.HTML](http://tpehommeaugmentee-monsite.com/pages/partie-2/les-exosquelettes-motorises.html)
[HTTPS://WWW.MEUBLEDESTYLE.FR/ARMURE-MEDIEVALE-DE-CHEVALIER-185-CM-MEUBLE-DE-STYLE,FR,4,MOR-JLA001-BK.CFM](https://www.meubledestyle.fr/armure-medievale-de-chevalier-185-cm-meuble-de-style,fr,4,mor-jla001-bk.cfm)
[HTTP://CYBERNETICZOO.COM/MAN-AMPLIFIERS/1964-EXOSKELETON-KULTSAR-AMERICAN/](http://cyberneticzoo.com/man-amplifiers/1964-exoskeleton-kultsar-american/)
[HTTPS://MASHABLE.COM/2017/09/29/BLADE-RUNNER-WHICH-VERSION-WATCH/?EUROPE=TRUE](https://mashable.com/2017/09/29/blade-runner-which-version-watch/?EUROPE=TRUE)
[HTTPS://ENTRAINEMENT-SPORTIF.FR/GENOUX-CROISSANCE.HTM](https://entrainement-sportif.fr/genoux-croissance.htm)
[HTTPS://CIRCUITDIGEST.COM/TUTORIAL/WHAT-IS-PWM-PULSE-WIDTH-MODULATION](https://circuitdigest.com/tutorial/what-is-pwm-pulse-width-modulation)
[HTTP://WWW.ERMICRO.COM/BLOG/WP-CONTENT/UPLOADS/2010/04/LASERLIGHT_08.JPG](http://www.ermicro.com/blog/wp-content/uploads/2010/04/laserlight_08.jpg)
[HTTPS://E2E.TI.COM/BLOGS_/ARCHIVES/B/PRECISIONHUB/ARCHIVE/2016/04/01/IT-S-IN-THE-MATH-HOW-TO-CONVERT-ADC-CODE-TO-A-VOLTAGE-PART-1](https://e2e.ti.com/blogs_/archives/b/precisionhub/archive/2016/04/01/it-s-in-the-math-how-to-convert-adc-code-to-a-voltage-part-1)
<https://roboiks.co.in/sensors/biomedical-sensor/emg-muscles-signal-sensor-v3-with-cable-and-electrodes>

INTRO :

[HTTPS://REWALK.COM/ABOUT-PRODUCTS-2/](https://rewalk.com/about-products-2/)
[HTTPS://HACKADAY.IO/PROJECT/25105-ALICE-ROBOTIC-EXOSKELETON](https://hackaday.io/project/25105-alice-robotic-exoskeleton)
[HTTPS://WWW.ARMY-TECHNOLOGY.COM/PROJECTS/RAYTHEON-XOS-2-EXOSKELETON-US/](https://www.army-technology.com/projects/raytheon-xos-2-exoskeleton-us/)
[HTTPS://WWW.YOUTUBE.COM/WATCH?V=UFLTDJ0fCHQ](https://www.youtube.com/watch?v=UFLTDJ0fCHQ)

ENERGIE :

[HTTP://WWW.TI.COM/LIT/DS/SYMLINK/LM1117.PDF](http://www.ti.com/lit/ds/symlink/lm1117.pdf)

ACCÉLÉROMÈTRE :

[HTTPS://WWW.ARDUINO.CC/EN/TUTORIAL/ADXL3XX](https://www.arduino.cc/en/tutorial/adxl3xx)
[HTTPS://WWW.TECHNO-SCIENCE.NET/DEFINITION/3668.HTML](https://www.techno-science.net/definition/3668.html)

EMG:

<https://www.gotronic.fr/art-capteur-emg-sen0240-27861.htm>
<https://www.robotshop.com/ca/fr/capteur-electrique-muscle-myoware.html>

ADC:

DATASHEET 12-BIT A/D CONVERTER WITH THRESHOLD DETECT (39739B)

