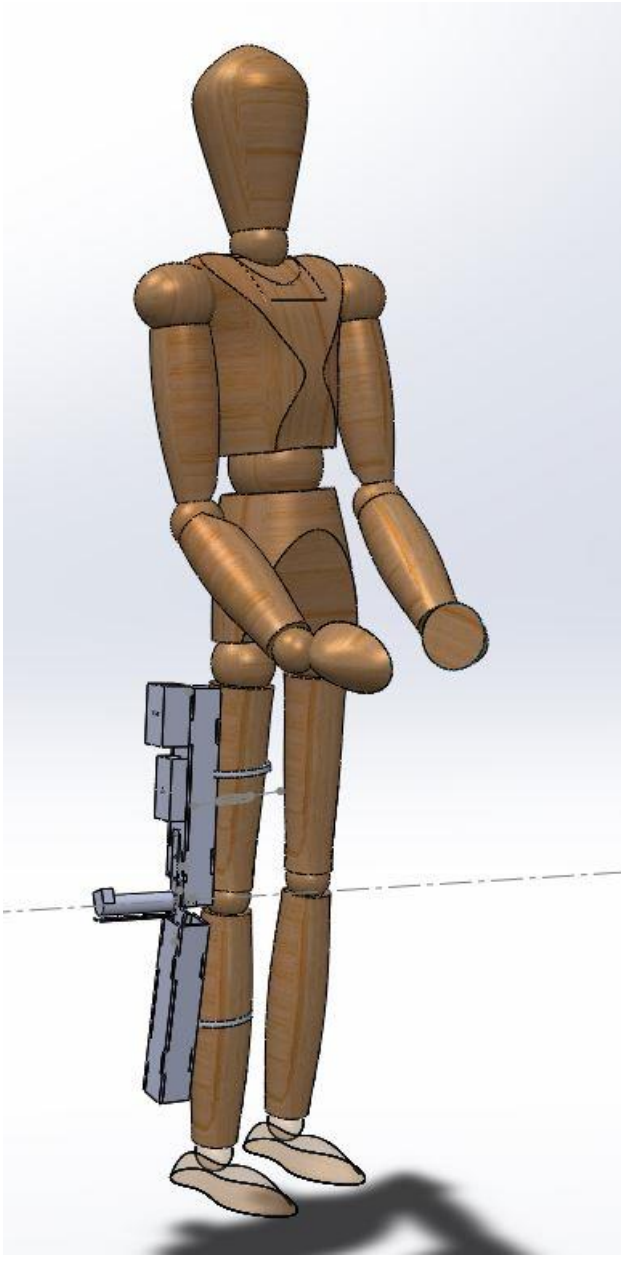
Timothée Fréville
Daria de Tinguy
Louis Lolivier

Projet Genou Héraclès



ESME Sudria Lyon

Projets Ingé 2 2018-2019

Fréville Timothée, Daria de Tinguy, Louis Lolivier

Sujet du projet : Projet Genou Héraclès

COMPTE RENDU :

Table des matières

Introduction 3

Cahier Des Charges 6

Composants 9

a) **Driver** 9

b) **Capteur EMG** 12

c) **Buck** 14

Circuit Electrique 15

a) Alimentation 12V 15

b) Alimentation 5V 16

c) Alimentation 3,3V 17

d) Alimentation du capteur EMG 17

Mécanique 20

PWM 25

ADC 30

Capteur de Fin de Course 36

Fonction du mouvement 39

Introduction

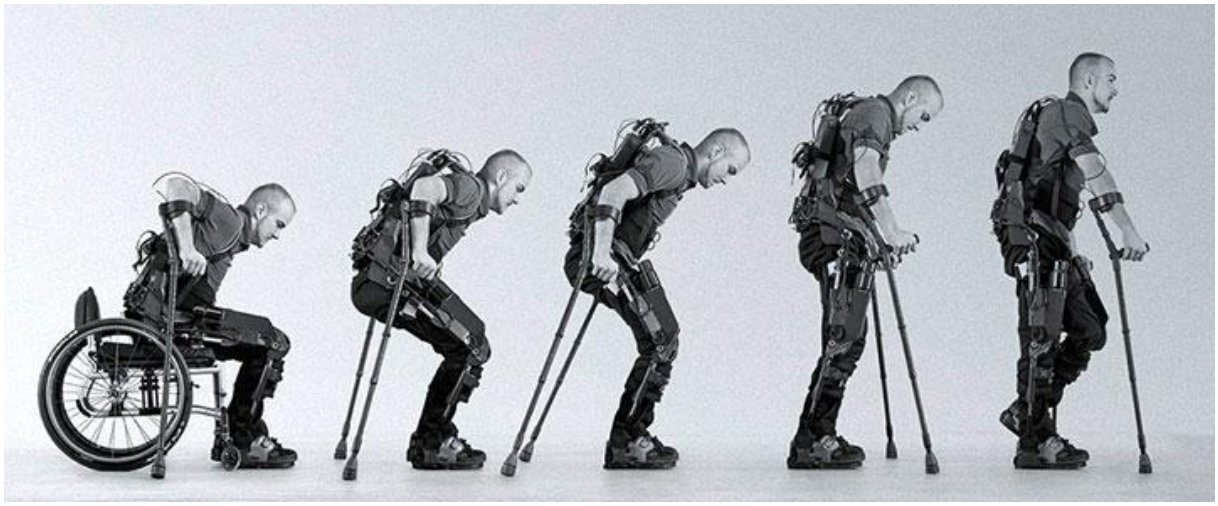


FIGURE 1HTTP://TPEHOMMEAUGMENTE.E-MONSITE.COM/PAGES/PARTIE-2/LES-EXOSQUELETTES-MOTORISES.HTML

L'idée même d'une protese mécanique pour remplacer un membre n'est pas si récente que cela. En un certain point de vu les armures du moyen-âge étaient déjà des augmentations en soit et tous les outils utilisés par l'humanité sont des extensions de ces membres donc l'idée de vouloir s'affranchir de certaine contrainte physique est aussi vieille que l'invention du 1ere outil par l'Homme.

FIGURE 2HTTPS://WWW.MEUBLEDESTYLE.FR/ARMURE-MEDIEVALE-DE-CHEVALIER-185-CM-MEUBLE-DE-STYLE,FR,4,MOR-JLA001-BK.CFM

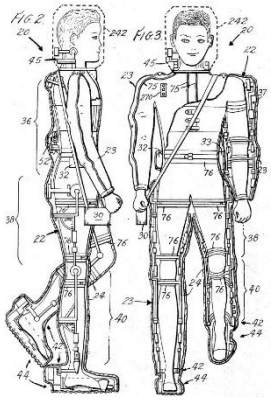


FIGURE 3HTTP://CYBERNETICZOO.COM/MAN-AMPLIFIERS/1964-EXOSKELETON-KULTSAR-AMERICAN/

En 1964 l'américain Emery Kultsar design l'un des premiers exosquelettes. Son objectif est de protéger les travailleurs des conditions extrême 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 pour le porteur. Le Brevet de cette machine a été déposer dans les années 60 mais elle était complétement irréalisable pour l'époque.

Comme on peut le voir dans la culture actuelle dans des film comme Blade Runner ou dans les livres d'anticipation comme ceux d'Issac Asimov. Quelle soit protése ou augmentation humaine l'idée d'un corps augmenté à traverser 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 4HTTPS://MASHABLE.COM/2017/09/29/BLADE-RUNNER-WHICH-VERSION-WATCH/?EUROPE=TRUE

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 somme renseigner sue les projets similaires qui ont déjà été réaliser. 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 intéressante 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 le second que nous avons trouvé 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 utilisés pour réaliser notre projet.

Figure 1. De gauche à droite, le projet REWALK et le projet ALICE

Figure 2. De gauche à 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
Assurer les charges	Renfort	Capacité de charge supplémentaire sans efforts additionnel	4kg	Compris dans les caractéristiques moteur	500g
	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 maximum 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 20kH	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 :

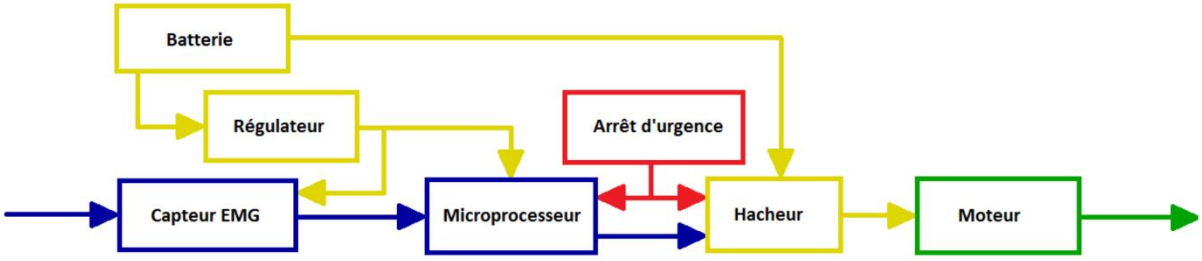


Schéma fonctionnel :

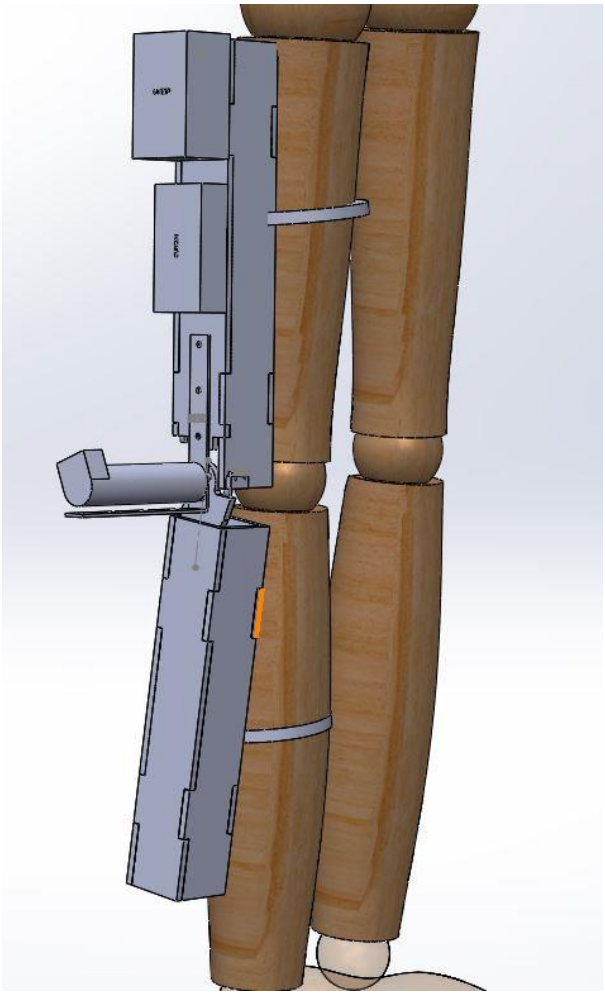
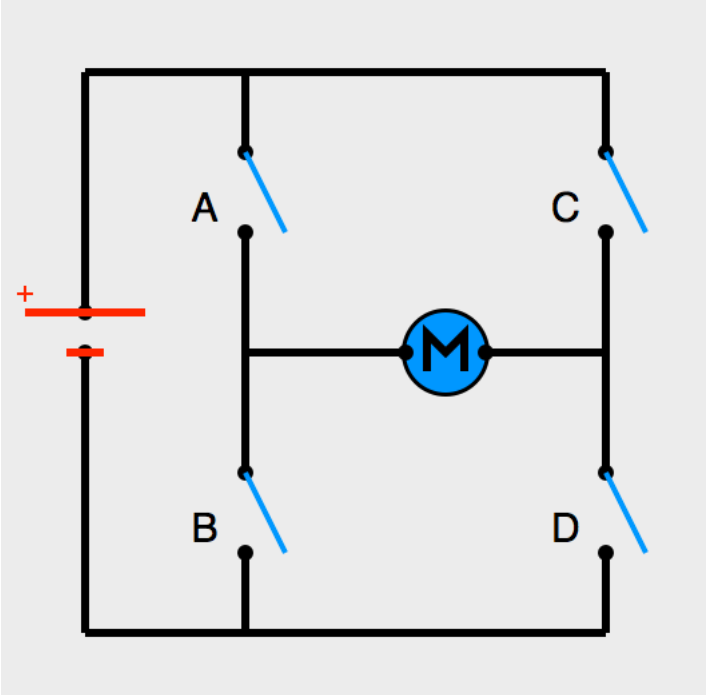


FIGURE 5: SCHEMA FONCTIONNEL DE LA PARTIE GENOUX D'HERACLES

Composants

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.

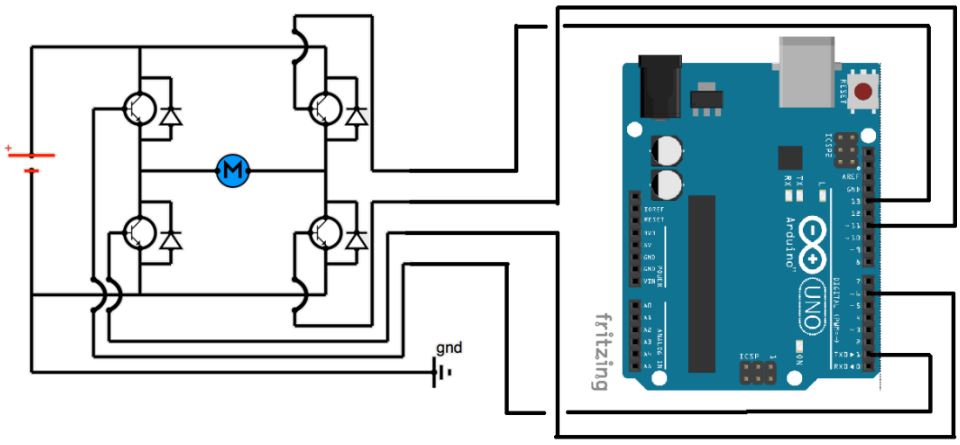


Le principe est plutôt simple :

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

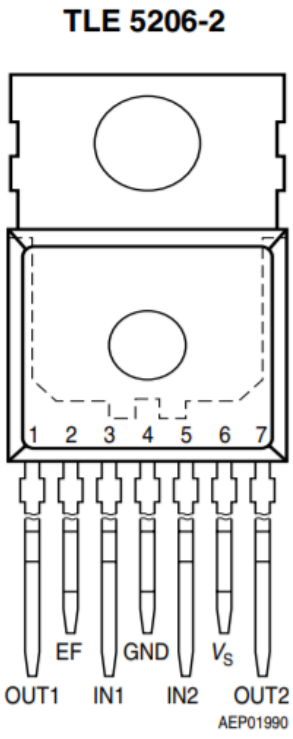
En fermant **B** et **C** on fait tourner le moteur dans le sens opposé

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'entrainement du moteur ne crée du courant qui pourrait endommager le microprocesseur. On a donc le schéma suivant :



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.

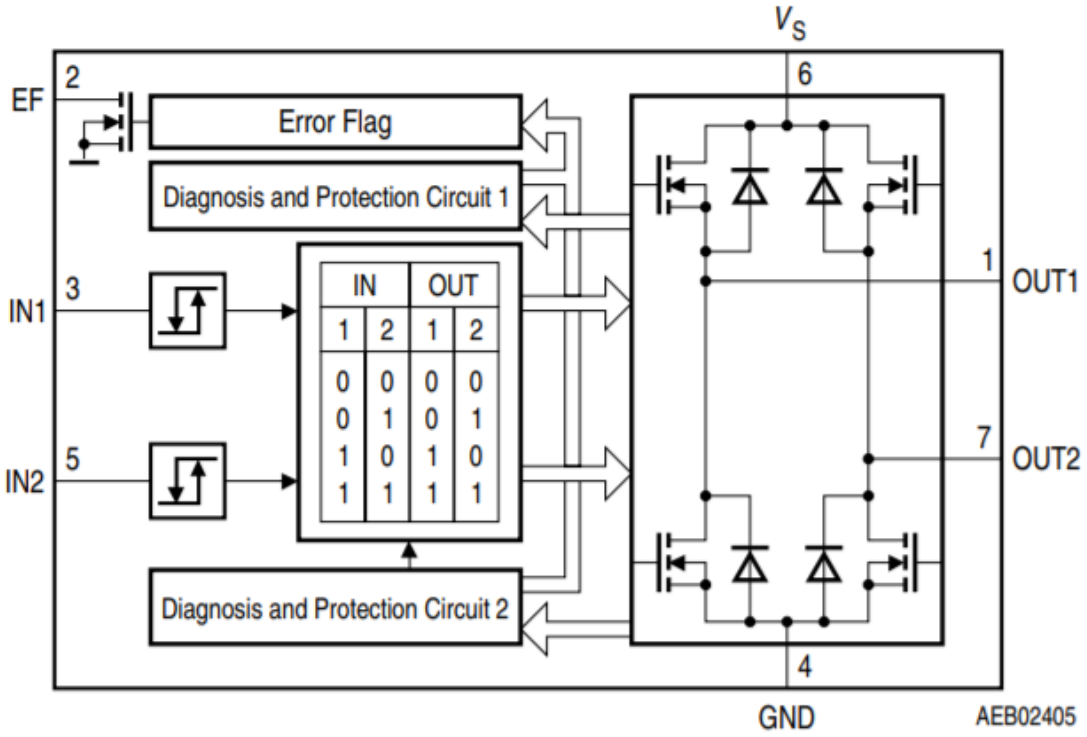
TLE52062GAUMA1



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

Diagramme bloc du Driver :



Sens Horaire :

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

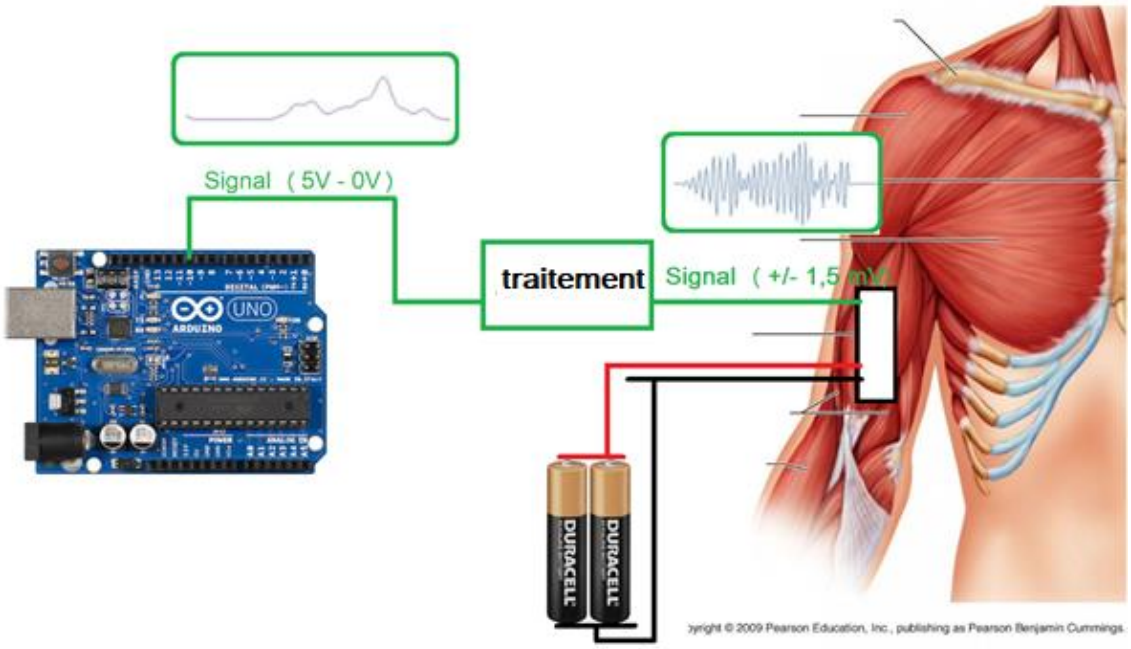
Sens anti-Horaire :

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

b) 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. [1]

Définissions les besoins actuels : on veut récupérer le signal d'un muscle qui d'après Wikipédia et d'autres sites envoi entre +/- 1.5mV [2], le reconditionner et le diriger vers un Microprocesseur pour ensuite pouvoir l'utiliser. Ce qui nous donne le schéma suivant :

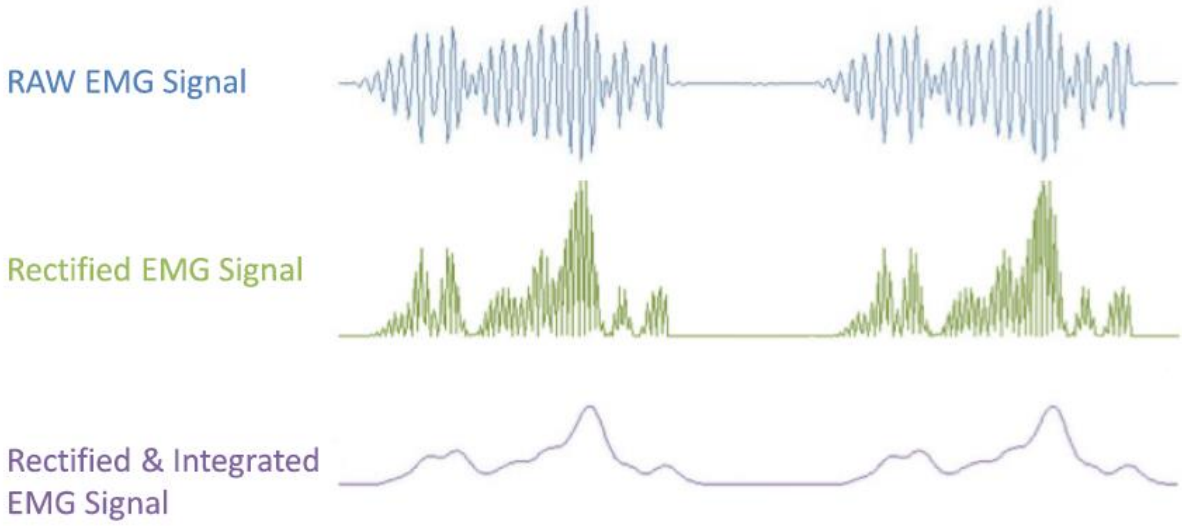


Pour le capteur musculaire on à 2 choix qui s'offrent à nous :

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 1er temps redresser le signal, puis filtrer l'enveloppe avec un passe bas comme cela :



- **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.

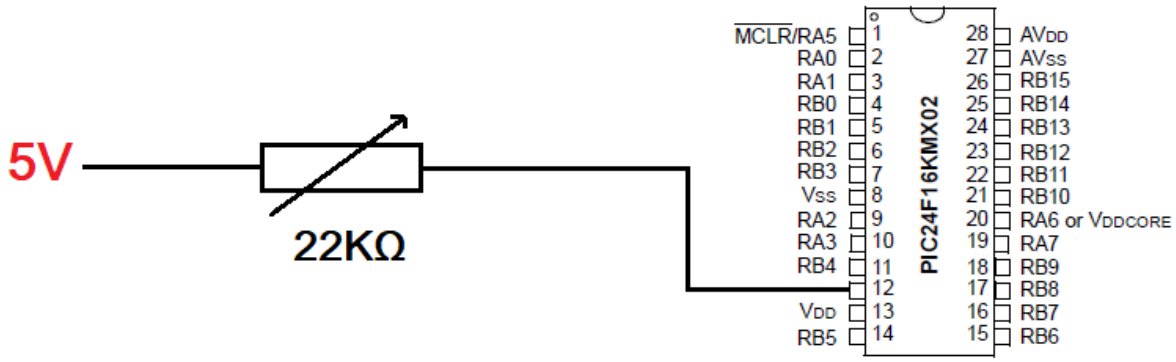
Conclusion :

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.

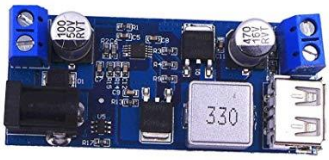
L'EMG et le potentiomètre fonctionnent de la manière suivante :

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



c) Buck

Niveau alimentation on a besoin de 12V pour le moteur. Cela dit certains des composant comme le microcontrôleur sont alimentés en 5V. Pour faire la jonction entre les 2 on va placer un Buck qui fait la conversion 12V/5V. Le Buck est parfait pour ce genre d'usage a faible/moyenne puissance.



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.

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 transformé 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 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 1 Ampère). 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.

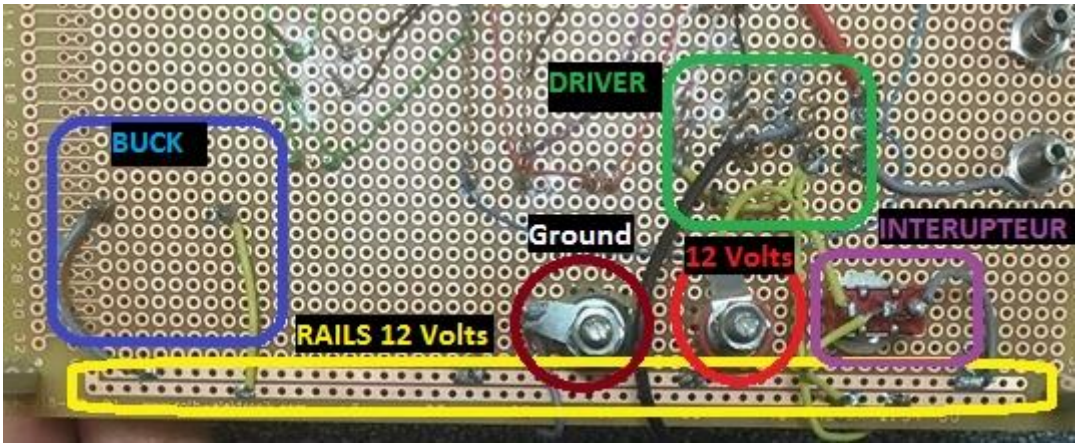


Photo du câblage de l'alimentation 12 Volts

Alimentation 5V

La plus grosse partie de notre système électronique, et particulièrement le microcontrôleur, est alimenté 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 reliés 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 Imax, le courant maximum accepté par la LED = 0.020 Ampères soit R = 5/0.020 = 250 Ohms. On obtient le schéma de câblage suivant :

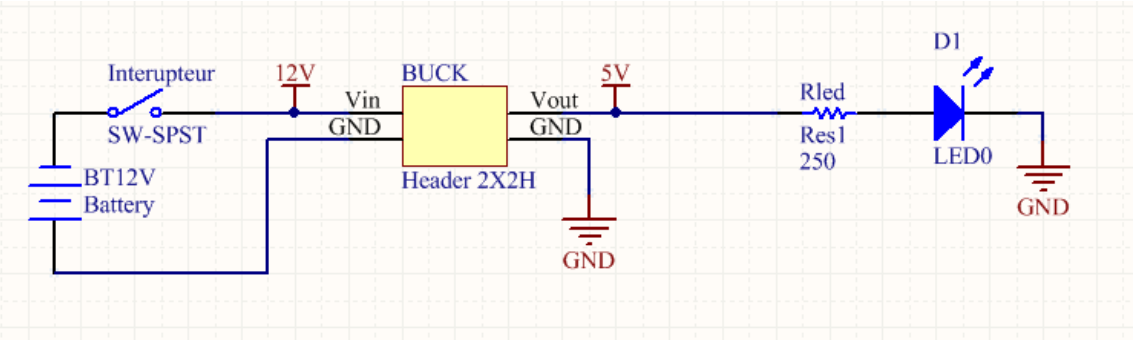


Schéma du câblage du Buck

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.

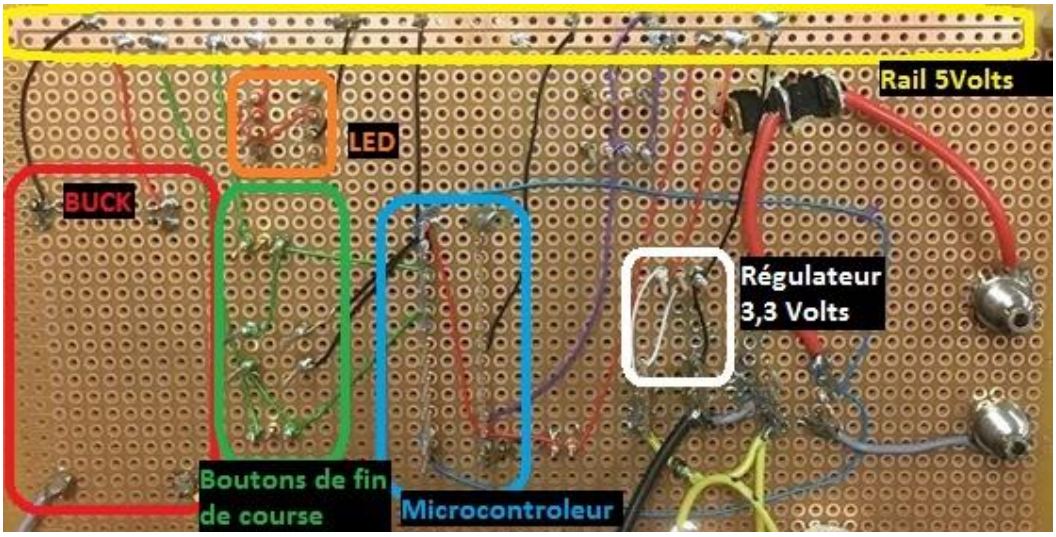


Photo du câblage de l'alimentation 5 Volts

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. Comme pour la plupart des régulateurs linéaires, deux condensateurs sont nécessaires. Ici il s'agit de condensateur 10 microfarads branchés comme suit :

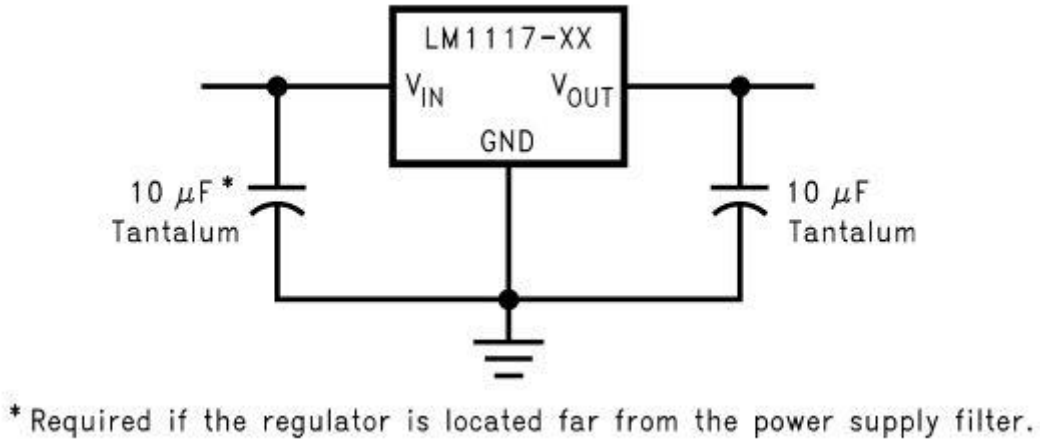
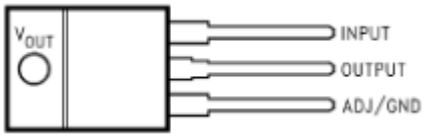


Schéma de câblage du LM1117-XX selon la datasheet

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



Configuration des pins du LM1117-3.3 selon la datasheet

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 :

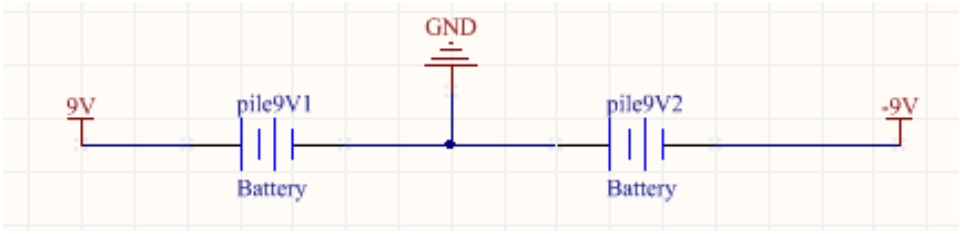


Schéma de câblage de l'alimentation du capteur

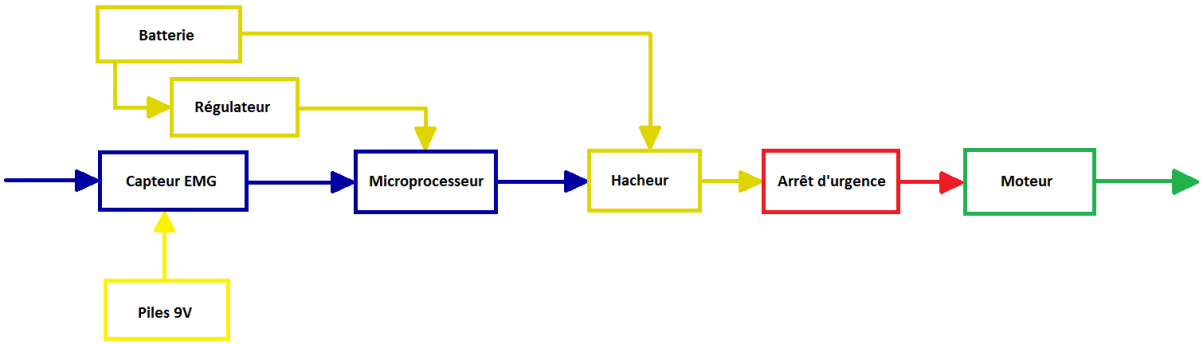


Schéma synoptique du système

En reprenant le code couleur du schéma synoptique du système, on peut diviser le câblage en 4 parties :

Jaune : Alimentation du système

Bleu : Intelligence du système

Rouge : arrêt d'urgence

Vert : Partie mécanique

On retrouve ces parties sur le schéma de câblage :

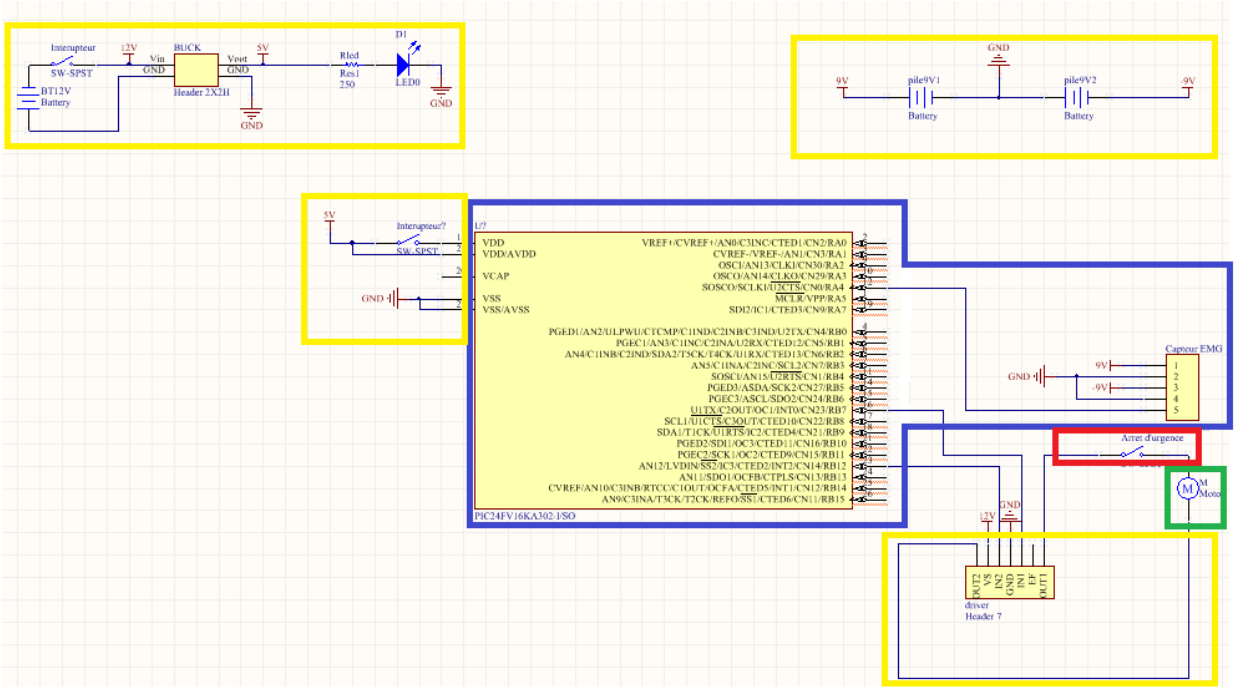


Schéma de câblage du système divisé en 4 parties

Mécanique

Explication de nos choix concernant la mécanique de l'ensemble.

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 comptions 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.

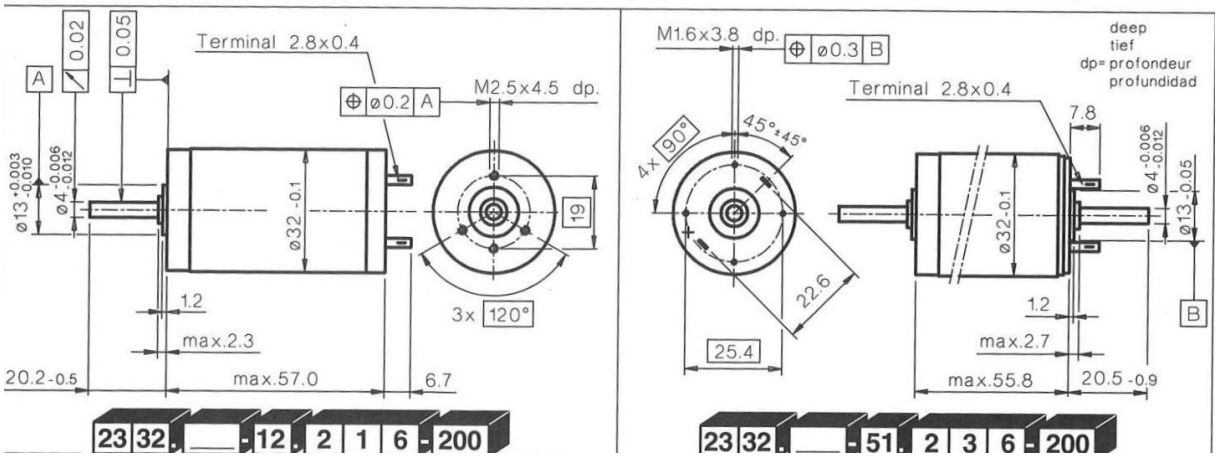


FIGURE 6: SCHEMA DU MOTEUR

Caractéristiques moteur	N° de bobinage (N° de commande)	966
		15
		12,00
		4750
1 Puissance conseillée	W	86,9
2 Tension nominale	Volt	57,3
3 Vitesse à vide	tr/min	34,4
4 Couple de démarrage	mNm	3780
5 Pente vitesse/couple	tr/min/mNm	3,18
6 Courant à vide	mA	9200
7 Courant de démarrage	mA	1200
8 Résistance aux bornes	Ohm	27,60
9 Vitesse limite	tr/min	10200
10 Courant permanent max.	mA	77,0
11 Couple permanent max.	mNm	23,0
12 Puissance max. fournie à la tens. nom.	mW	415
13 Rendement max.	%	14,6
14 Constante de couple	mNm/A	24,3
15 Constante de vitesse	tr/min/V	0,53
16 Constante de temps mécanique	ms	12,50
17 Inertie du rotor	gcm²	1,90
18 Inductivité	mH	
19 Résistance therm. carcasse/ambiant	K/W	
20 Résistance therm. rotor/carcasse	K/W	

FIGURE 7 : 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.

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 (gratuitement du moins). Nous avons donc choisi un squelette en bois de 6mm d'épaisseur pour de premiers essais.

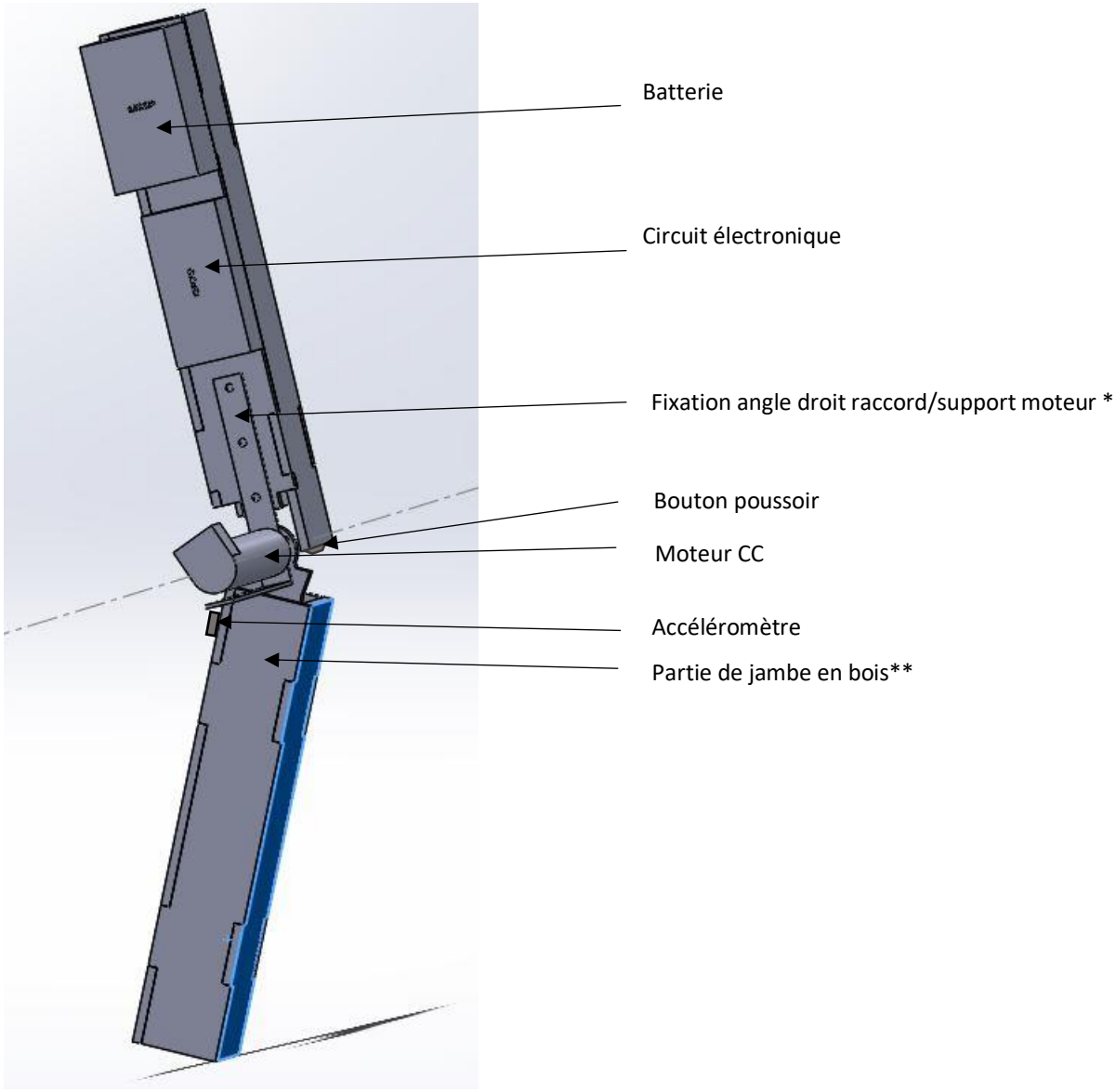


FIGURE 8 : SOLIDWORKS HERACLES V2

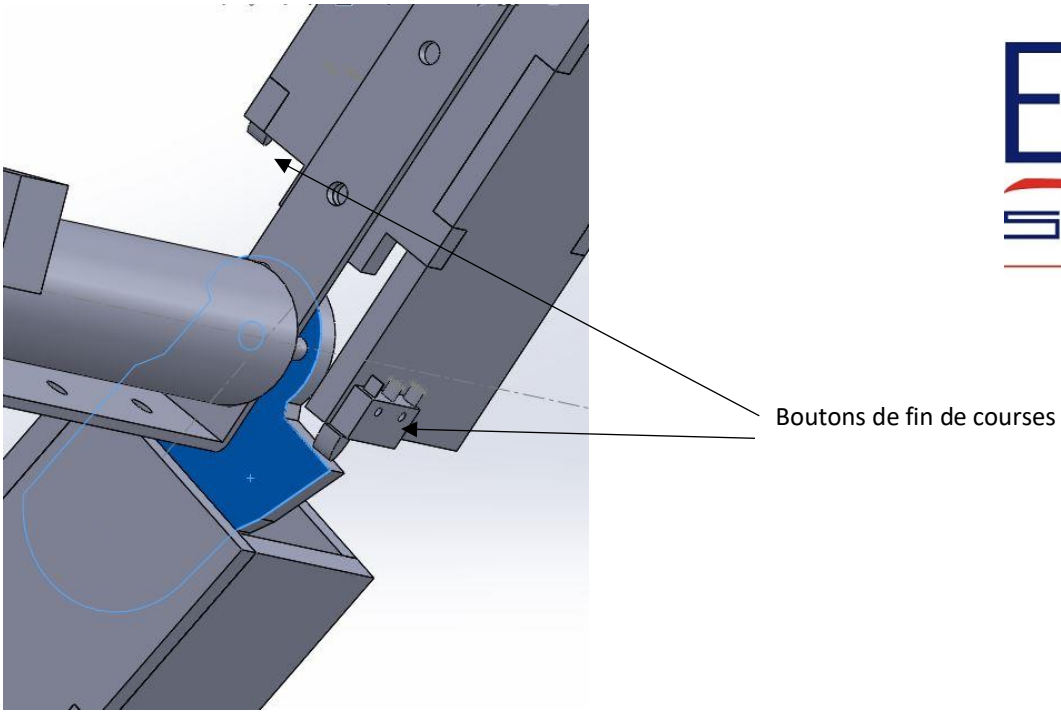


FIGURE 9: SOLIDWORKS HERACLES ZOOM SUR LES BOUTONS DE FIN DE COURSES

*Fixation angle droit raccord/support moteur

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 supérieur « fixe » de la jambe (par rapport au moteur du genoux) et la partie inférieure mobile de la jambe et à prévenir le désaxage du moteur par rapport à son axe en cas de choc.

**Partie de jambe en bois

Les parties hautes et basses de la jambe ainsi que le raccord pivot ont été usinés 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.

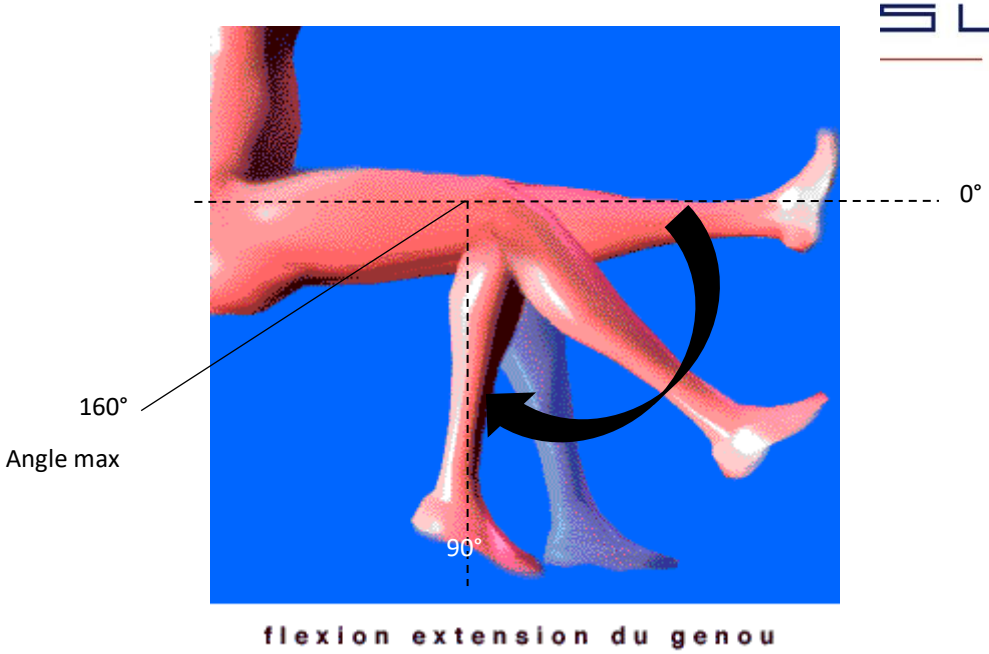


FIGURE 10: JAMBE DEPLIEE ET REPLIEE AU MAXIMUM.

HTTPS://ENTRAINEMENT-SPORTIF.FR/GENOUX-CROISSANCE.HTM

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, position assise, voire course -pas optimisé pour ça toutefois-.

Les éléments mécaniques bloquent donc les mouvements à -7° (en prenant en compte qu'une jambe tendue n'est pas exactement droite) jusqu'à 119°.

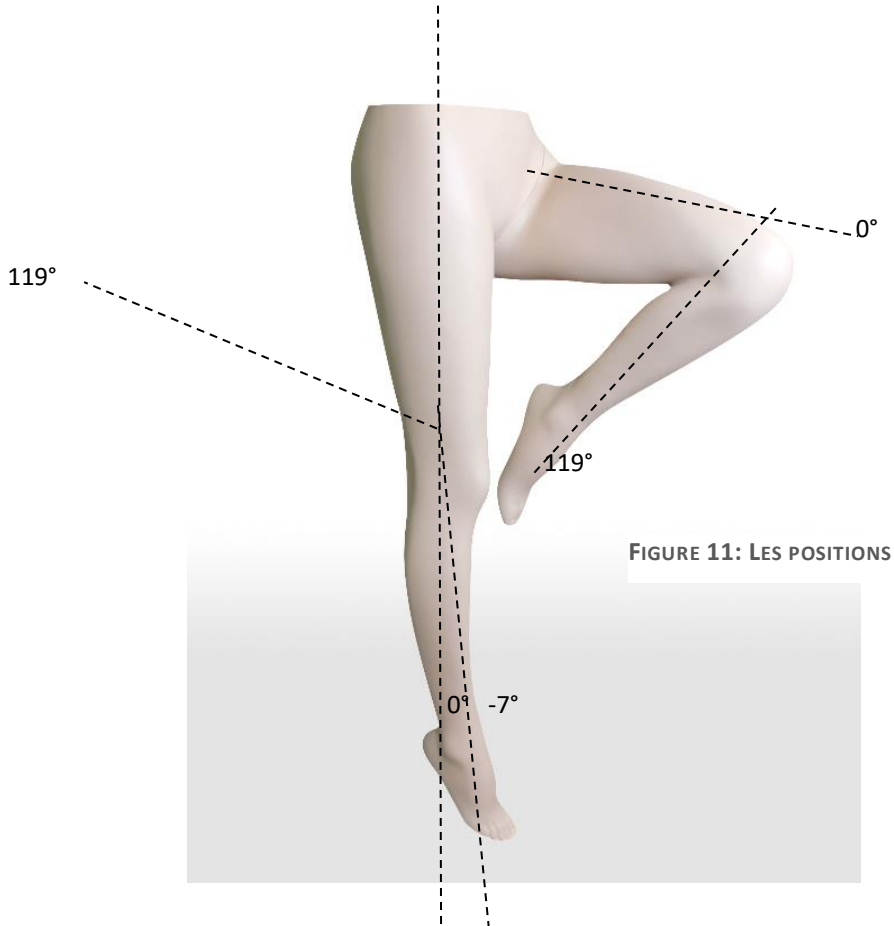


FIGURE 11: LES POSITIONS MAXIMALES AVEC HERACLES

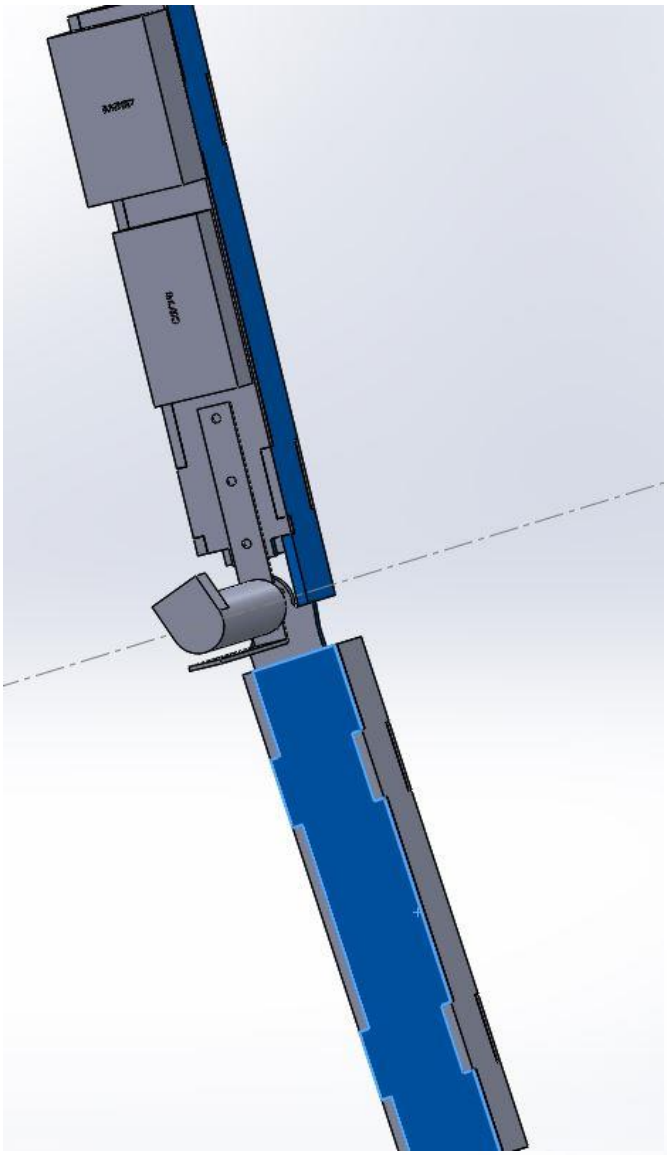


FIGURE 12: SOLIDWORK HERACLES _ANGLE DE 119°

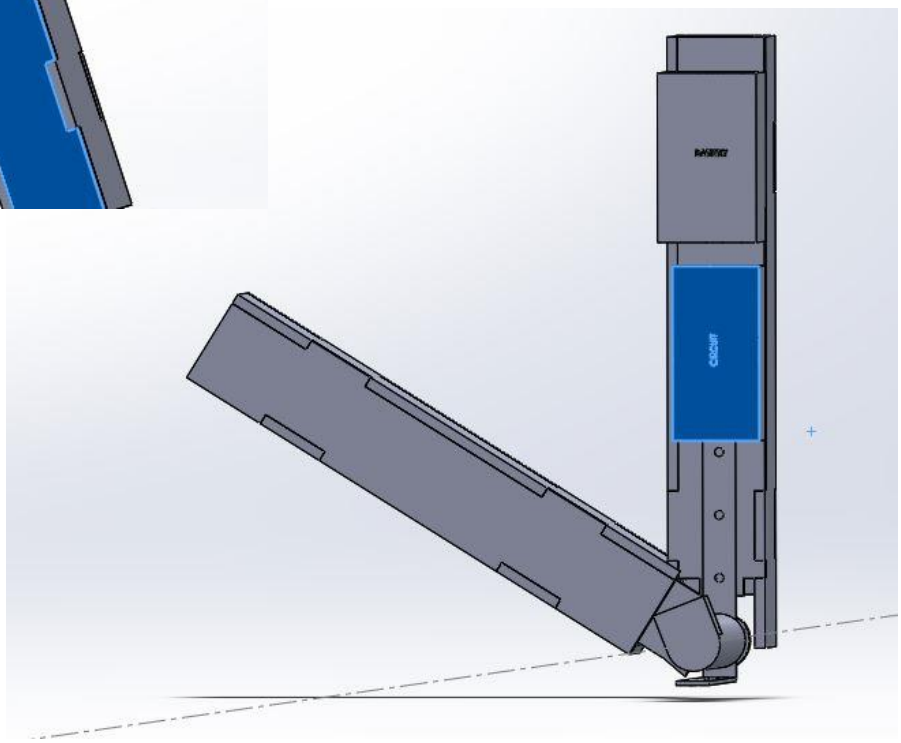


FIGURE 13 : SOLIDWORK HERACLES _ANGLE DE -7°

PWM

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 :

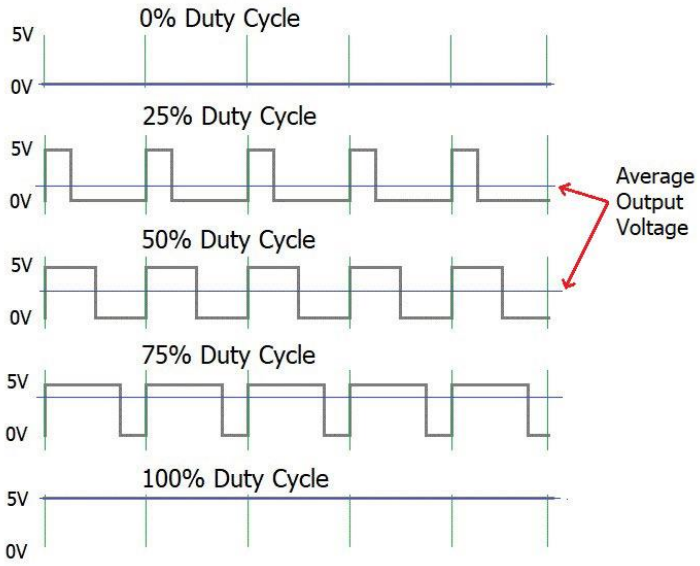


FIGURE 14HTTPS://CIRCUITDIGEST.COM/TUTORIAL/WHAT-IS-PWM-PULSE-WIDTH-MODULATION

- 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 intéresse.

Ra : valeur ou le signal chute

Rb : valeur pour 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%

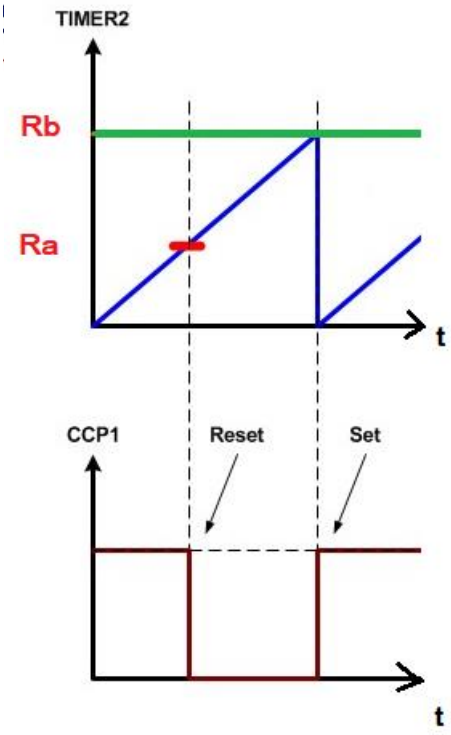


FIGURE 15:HTTP://WWW.ERMICRO.COM/BLOG/WP-CONTENT/UPLOADS/2010/04/LASERLIGHT_08.JPG

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

    //Configure MCCP 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 = 0b00000;       // Select Sync/Trigger source (Self-sync)

    //Configure MCCP 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)
    CCP1TMRL = 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 MCCP operating mode
    CCP2CON1Lbits.CCSEL = 0;            // Set MCCP operating mode (OC mode)
    CCP2CON1Lbits.MOD = 0b0101;         // Set mode (Buffered Dual-Compare/PWM mode)
    //Configure MCCP 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 = 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.

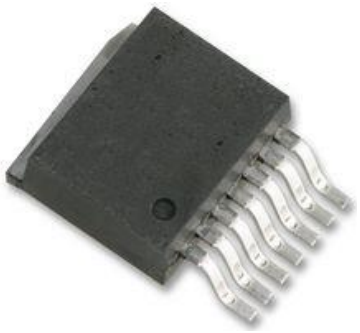
Avant de passer aux fonctions qui vont gérer les valeurs des registres des PWM on va d'abord définir quelques registres importants pour ne pas se perdre. Ainsi en début de code du fichier

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

pwm.c

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 fonctionner le moteur dans un sens ou dans l'autre

sans avoir à changer un tas de registre. Pour cela on va retourner sur le fonctionnement du Driver. Petit rappel du fonctionnement :



Sens Horaire :

- **IN1** doit être en High
- **IN2** envoi 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 1er 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.

ADC

Explication de nos choix concernant l'ADC

DATASHEET : DATASHEET 12-BIT A/D CONVERTER WITH THRESHOLDDETECT (39739B)

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 $2^{nbrs\ de\ bits\ de\ conversion}$-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.

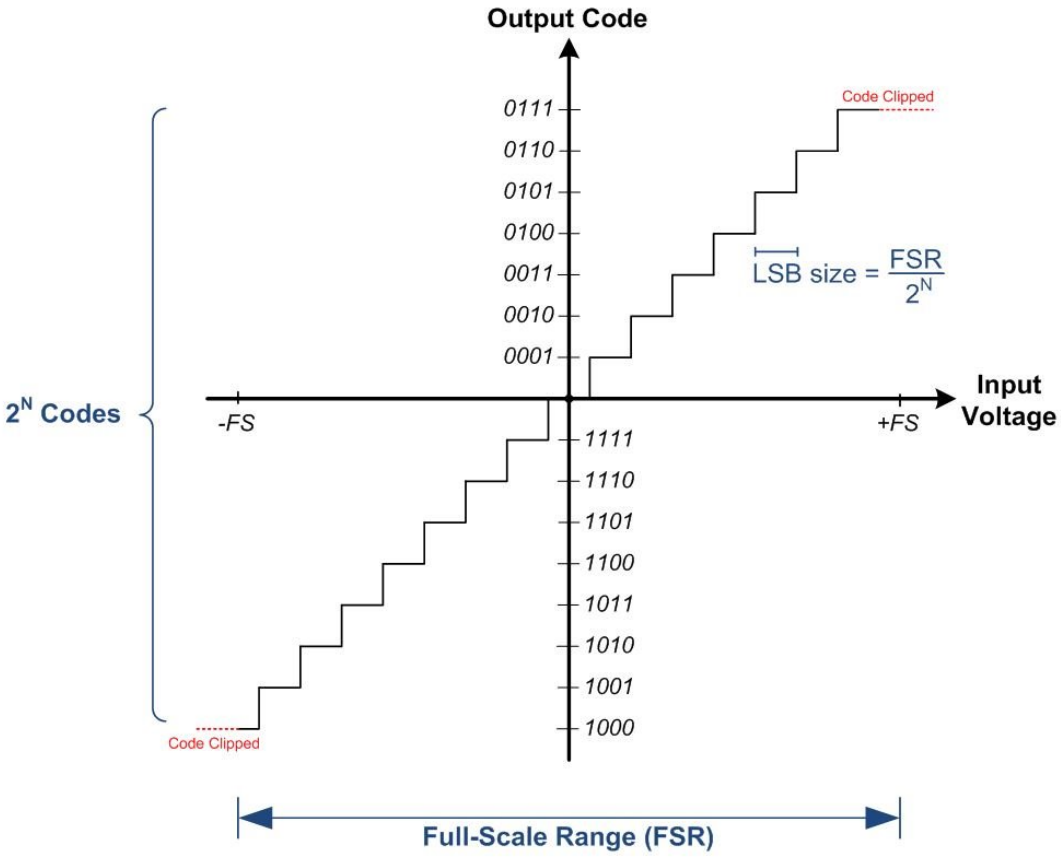

Output Code
0111
0110
0101
0100
0011
0010
0001
1111
1110
1101
1100
1011
1010
1001
1000
Code Clipped
LSB size = FSR / 2^N
Input Voltage
-FS
+FS
2^N Codes
Code Clipped
Full-Scale Range (FSR)

FIGURE 16 : EXEMPLE DE GRADATION DES VALEURS D'ENTREES REPARTIES ENTRE LES VALEURS MIN ET MAX POSSIBLES AVEC N=4

Source :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

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.

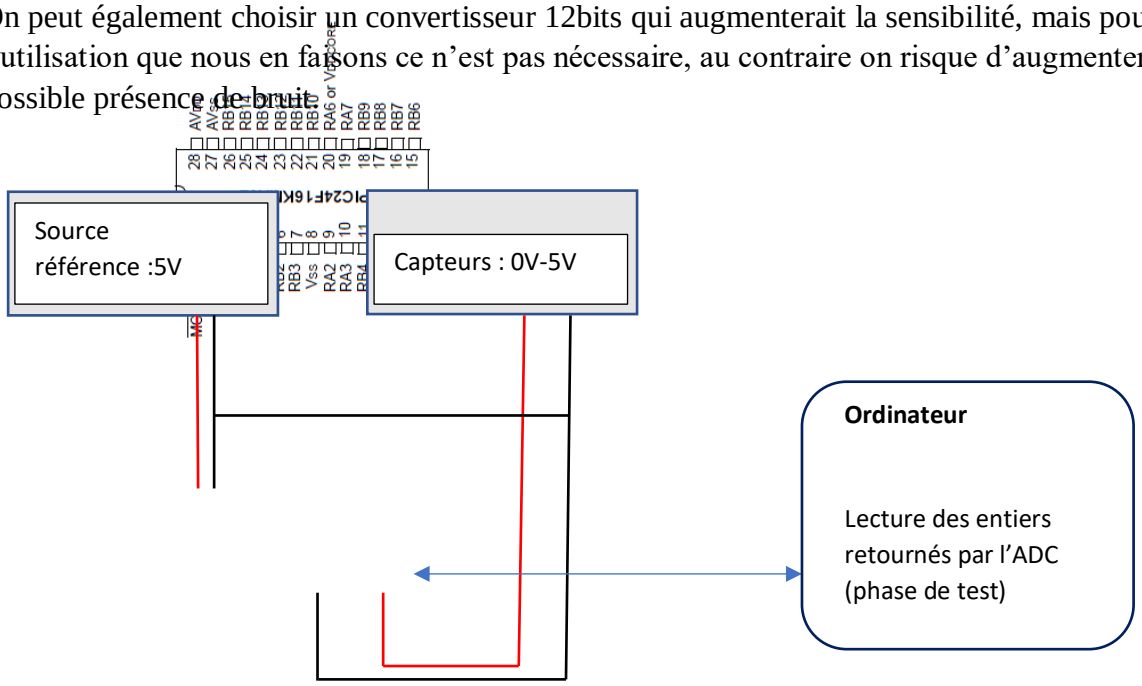


FIGURE 17: SCHEMA DE CABLAGE DU MICROCONTROLEUR EN PHASE DE TEST POUR VERIFIER LE FONCTIONNEMENT DE L'ADC.

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.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 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

$$T_{AD} = T_{CY}(ADCSx + 1)$$

$$ADCSx = \frac{T_{AD}}{T_{CY}} - 1$$

FIGURE 18 : PASSAGE DE LA DATASHEET 12-BIT A/D CONVERTER WITH THRESHOLDDETECT

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;
}
```

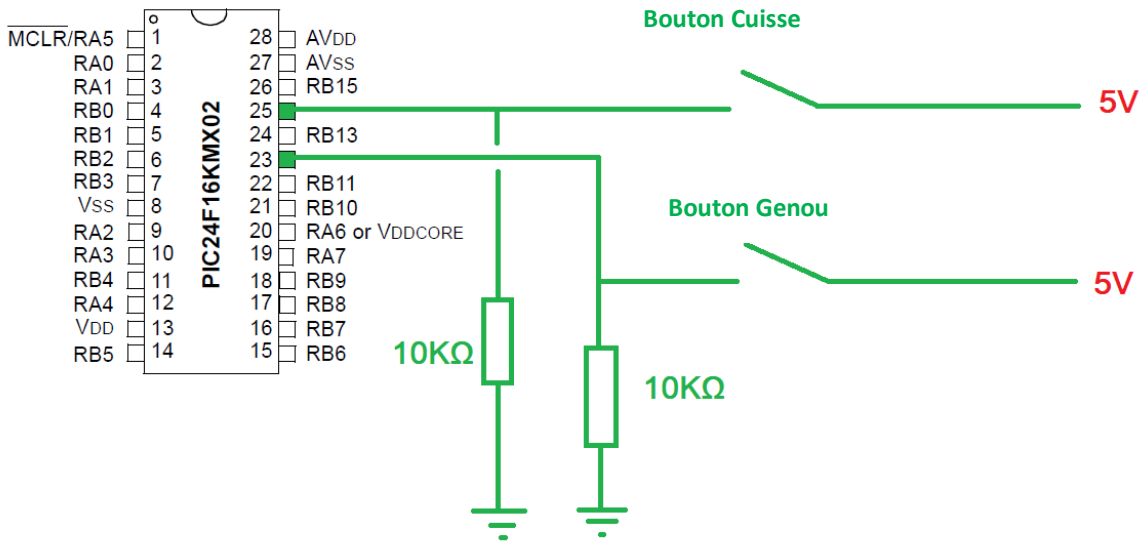
Nous avons choisi de prendre 3 valeurs (ça aurait tout aussi bien pu être 4 ou 5, ce fut choisi empiriquement) et d'en faire une moyenne avant de l'envoyer au reste du programme.
Les delays présent dans le ADC_read n'impactent pas le mouvement fluide de l'exosquelette.

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 :



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 on 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 Interupt
IEC1bits.INT2IE = 1;         // enable INT0 Interupt
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 Interupt
IEC1bits.INT1IE = 1;         // enable INT0 Interupt
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 *innterupt.c.*

Au début :

#define Bouton_fin_biceps IFS1bits.INT1IF

#define Bouton_fin_coude IFS1bits.INT2IF

```
//definition des variable que l'on vas souvent utiliser
#define Bouton_fin_biceps IFS1bits.INT1IF
#define Bouton_fin_coude 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 servirons 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_coude=0;
}
```

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

Fonction du mouvement