

Programmation du PIC

Contraintes imposées sur le système :

- Commande par un microcontrôleur PIC 16F84 (pas de sortie analogiques).
- Variation de vitesse réalisées à partir d'un C.N.A. utilisant un mot de 8 bits en entrée.
- L'accélération et la vitesse maximale dépendent de l'angle d'inclinaison de la bande.

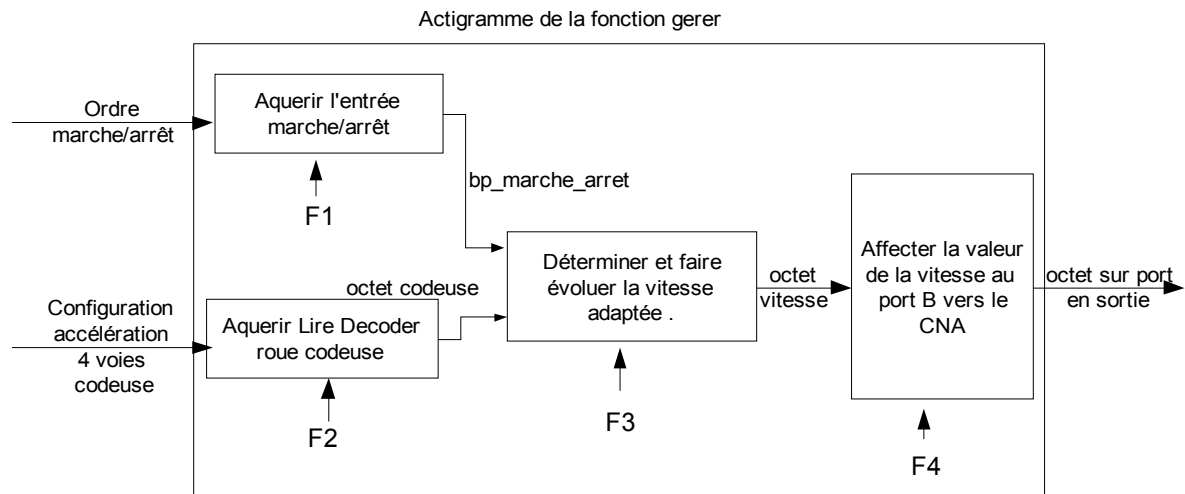
Hypothèses et choix réalisés.

- L'utilisation d'un microcontrôleur PIC 16F84 permet d'utiliser 13 entrées sorties :
 - 8 E/S, RB0 à RB7 sur le port B
 - 5 E/S, RA0 à RA4 sur le port A
- Le C.N.A. doit être commandé depuis le PIC par un mot de 8 bits
Choix : le port B disposant de 8 lignes, il sera donc configuré en sortie pour communiquer avec le CNA.
- Il reste donc 5 lignes sur le port A, qui seront utilisées comme entrée de commande(bouton poussoir) et de configuration (roue codeuse).

Les affectations sont réalisées de la manière suivante :

- Utilisation d'une roue codeuse en entrée de configuration (angles d'inclinaison de la bande). La roue codeuse utilise 4 lignes pour permettre de sélectionner 10 valeurs d'angles.
- On ne dispose donc que d'une seule ligne pour la commande de marche et d'arrêt de la bande. Il conviendra donc de détecter une impulsion sur le bouton (front montant) pour changer de régime (marche ou arrêt).

Actigramme des fonctions:



F1 : Lire l'entrée bouton marche/arrêt.

F2 : Lire et décoder l'entrée roue codeuse.

F3 : Déterminer la vitesse à obtenir.

F4 : Affecter la vitesse sur le port B.

Programme n°1 :

Objectif :

valider pilotage du CNA et réalisation des rampes accélération décélération.

Principe: programme simple:

- un cycle programme correspond à un cycle de la bande transporteuse.
- La fonction F2: lecture et décodage de la roue codeuse n'est pas envisagée.
- Pas de gestion de l'accélération en fonction de la pente
- pas d'arrêt possible en cours de programme (une impulsion sur le bouton marche génère les trois phases : accélération, palier vitesse constante, décélération)

Programme utilisé : prog_base_3.c

Mise en place d'une rampe d'accélération pour éviter le renversement de la canette au démarrage et à l'arrêt.

Code :

```

/*****Rampe accélération*****/
for (cpt=0; cpt<255; cpt++)
{
    sortie=cpt+1;
    delay(tempoAccelDecel);
}

```

gestion des tests (simulation)

- lancer la simulation
-->(les sorties doivent rester à 0)
- mise à 1 entré Ra1 (bouton poussoir marche)
--> les sorties clignotent (incréméntation port B, accélération)
puis restent fixes au niveau 1 (palier vitesse constante) puis
clignotent jusqu'a revenir au niveau 0 (décélération)

résultats des tests :

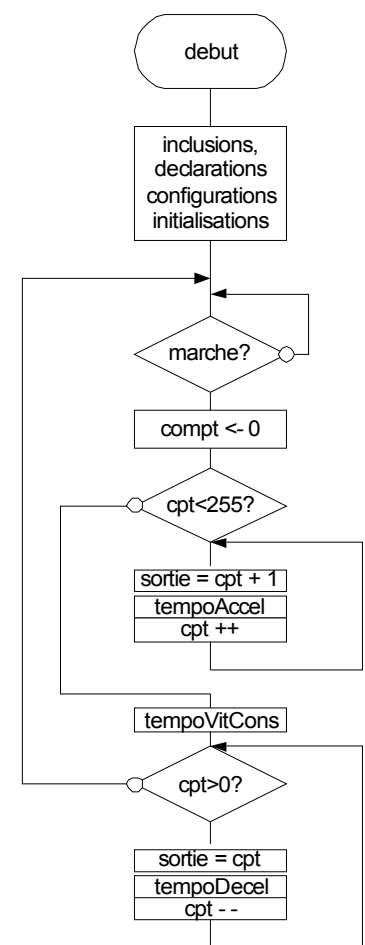
test OK.

La valeur de temporisation a du être adaptée pour que le clignotement soit perceptible.

validation sur maquette:

test OK

Algorithme:



Programme n°2 :

Objectif :

- décodage roue codeuse.
- Gestion des fonction, appels de sous programme pour garantir évolutivité , lisibilité et maintenabilité du programme.

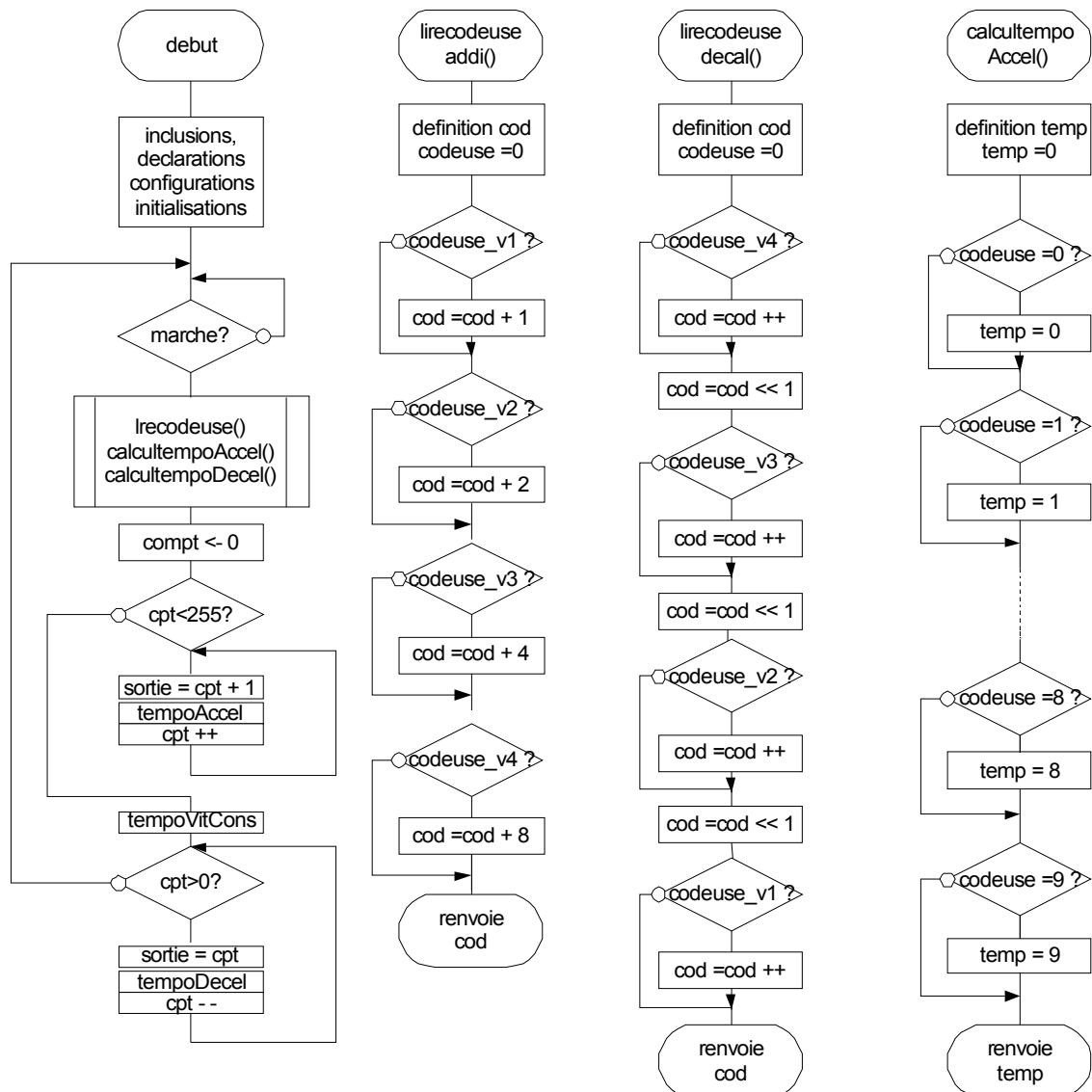
Principe: programme simple:

- un cycle programme correspond à un cycle de la bande transporteuse.
- L'accélération dépend de la valeur codée depuis la roue codeuse.
- pas d'arrêt possible en cours de programme

Programme utilisé : codeuse_b20.c

Algorithme:

Réalisation d'appels de fonctions dans un souci



Spécificités :

L'appel des fonctions de lecture et décodage de la roue codeuse est réalisé après le test du bouton marche afin de permettre à l'utilisateur de modifier la valeur de l'angle jusqu'à l'instant où il envoie une impulsion sur le bouton marche.

**Appel de fonctions :
MODIF**

Principe de de la fonction lirecodeusedecal() :

codeuse	voie 4	voie 3	voie 2	voie 1
poids	8	4	2	1

Exemple dans l'hypothèse
"1001"

Etats successifs de la variable cod

cod = 0	0	0	0	0
si voie_4 cod++ (vrai)	0	0	0	1
cod = cod << 1	0	0	1	0
si voie_3 (faux)	0	0	1	0
cod = cod << 1	0	1	0	0
si voie_2 (faux)	0	1	0	0
cod = cod << 1	1	0	0	0
si voie_1 cod++ (vrai)	1	0	0	1

Gestion des tests (simulation)

- Pour valider la réussite du décodage de la roue codeuse, on affecte à la sortie la valeur prise par la valeur de tempoAccel pendant le palier de vitesse constante.
- Test avec différentes valeurs de l'entrée roue codeuse.
- On réaffecte ensuite la valeur 255 comme valeur de sortie pendant le palier de vitesse constante.

Résultats des tests :

test OK.

Validation sur maquette:

Nécessité de compléter les voies de la roue codeuse pour avoir un signal cohérent.
test OK.

Programme n°3 :

Objectif :

- Permettre le lancement d'une phase de décélération à tout instant par impulsion sur le bouton marche/arrêt
- Simplifier la gestion des entrées.

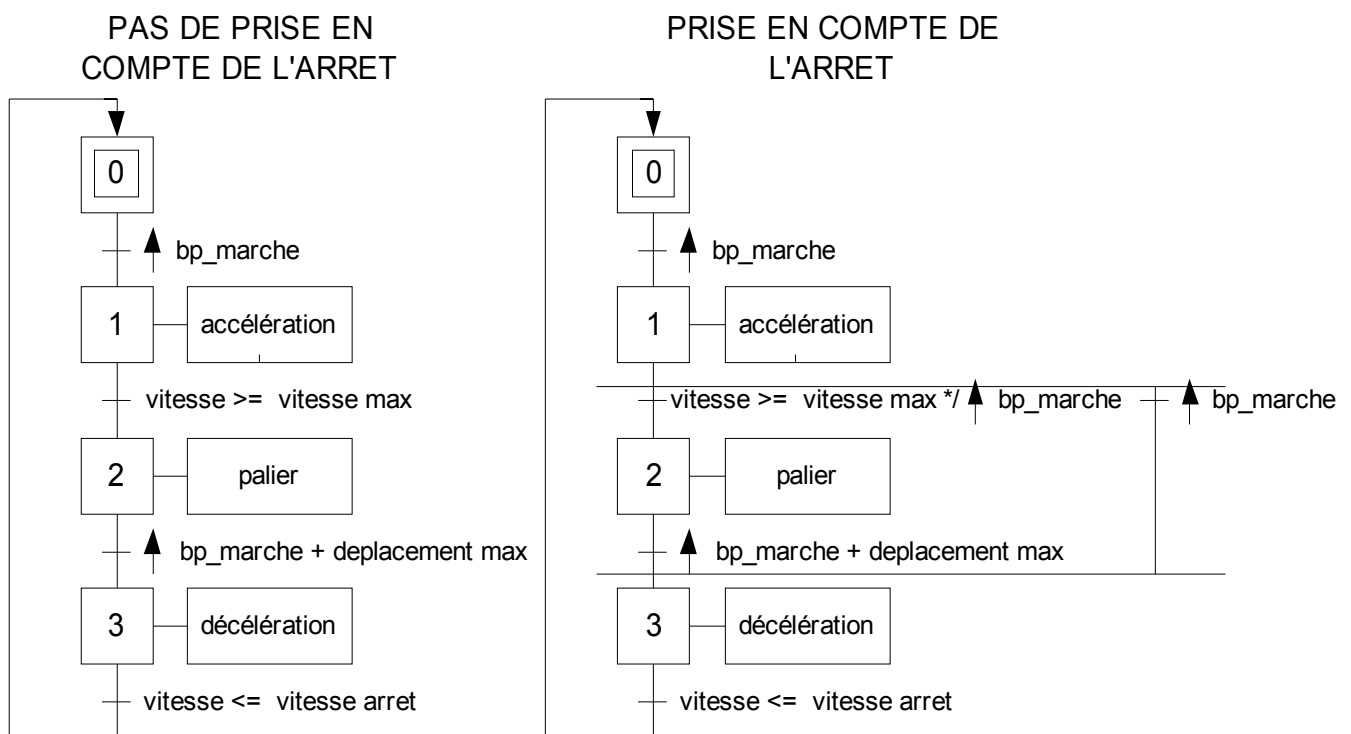
Principe:

- Les entrées doivent être scrutées en permanence de manière à détecter une impulsion sur le bouton marche/arrêt.
- Fonctionnement de type automate : un cycle programme dure quelques millisecondes et correspond au calcul du nouvel état du système en fonction de l'état précédent.
- Le fonctionnement peut-être décrit puis programmé sous forme d'un grafcet (4 phases ou étapes)
- Détection de fronts montants sur l'information bp_marche.
- Utilisation d'une table pour attribuer la valeur de la temporisation (accélération) en fonction de la variable codeuse.

Programme utilisé : phase6complet.c

grafcet décrivant les 4 phases de fonctionnement

- version initiale sans arrêt par bouton poussoir
- et version avec arrêt possible par impulsion sur le bouton arrêt / marche



programme type « programme automate »

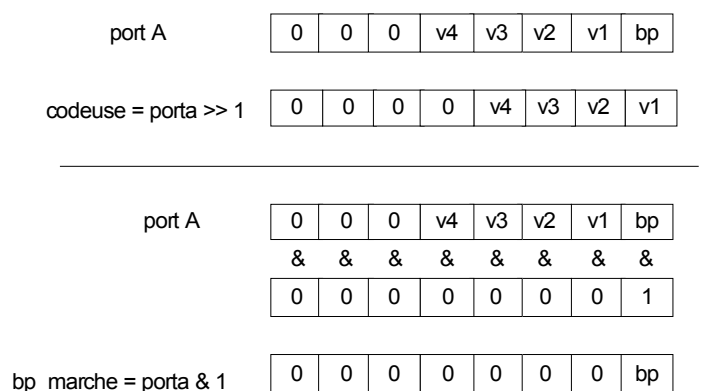
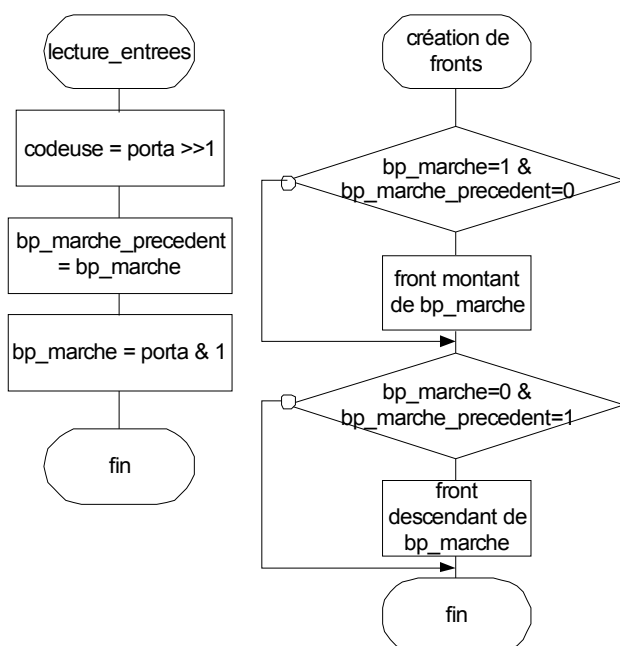
le programme scrute en permanence les entrées et détermine le nouvel état du système en fonction de l'état précédent et des entrées lues.

Le programme est entièrement réalisé à l'aide de fonctions, la fonction **main()** ne contenant que les initialisations et appels de sous fonctions.

```
void main()
{
    /*Initialisation des paramètres de conception*/

    /*Initialisation des paramètres de réglage*/

    lecture_entree(porta); // suppression du front initial ;
    /******Debut boucle infinie******/
    sortie = 0; phase = 0;
    for(;;)
    {
        /******Attente marche******/
        lecture_entree(porta);
        evoluer_phase();
        vitesse = piloter_vitesse(vitesse);
        sortie = vitesse;
    }
    //fin for(;;)
}
//fin void main()
```

gestion spécifique de l'entrée codeuse et détection du front montant

La variable codeuse est déterminée par suppression du bit `porta.0` qui représente le bouton poussoir. Cette méthode utilise le même principe que la fonction **lirecodeusedecal()** définie plus haut. En effectuant un décalage à gauche de un bit sur la variable « port » on supprime le bit « bouton poussoir » et on décale les 4 bits de la variable codeuse dans les 4 bit de poids faible.

A chaque cycle il convient de d'archiver la valeur précédente de `bp_marche` (définie dans la dernière boucle) dans la variable `bp_marche_precedent` pour pouvoir la réutiliser dans le cas de la détection d'un front. La valeur de `bp_marche` est obtenue grâce à l'extraction du bit `porta.0`. Attention l'archivage doit avoir lieu avant l'affectation de la nouvelle valeur depuis le port d'entrée.

L'opérateur bit à bit `&` est utilisé pour réaliser cette extraction au moyen d'un masquage

La détermination d'un front montant est réalisée par comparaison de l'état du bouton poussoir marche/arret entre 2 cycle successifs du programme. Cela est obtenu par comparaison des variables `bp_marche_precedent` et `bp_marche`.

```
if
((bp_marche==1)&&(bp_marche_precedent==0))
{
    //obtention d'un front montant de bp_marche
}
```

gestion et évolution du grafctet.

```
void evoluer_phase()
{
    switch (phase)
    {
        case 0 : {
            if ((bp_marche==1)&&(bp_marche_precedent==0))
            { phase = 1; }
            break;}

        case 1 : {
            if (vitesse >= vitesse_max )
            { phase = 2; }
            if ((bp_marche==1)&&(bp_marche_precedent==0))
            { phase = 3; }
            break;}

        case 2 : {
            if (deplacement >= deplacement_max[codeuse])
            { phase = 3; }
            if ((bp_marche==1)&&(bp_marche_precedent==0))
            { phase = 3; }
            break;}

        case 3 : {
            if (vitesse <= vitesse_arret)
            { phase = 0; }
            break;}
    }
}
```

La fonction **evoluer_phase()** réalise la programmation du grafctet, elle a pour but de faire évoluer les différentes phases en fonction des données du port A et de l'état du système vitesse et phase actuelle.

Les phases sont autant d'étapes de grafctet :

- Phase 0 : phase d'arrêt, phase où la bande est immobile.
- Phase 1 : phase d'accélération, jusqu'à la vitesse max.
- Phase 2 : phase de palier, où la vitesse est constante (maximale).
- Phase 3 : phase de décélération, jusqu'à l'arrêt de la bande => Phase 0.

Cette fonction est réalisée grâce à l'instruction *switch*, qui permet de faire une série de test alternatifs. Chaque *case*, qui représente une alternative, se finit par l'instruction *break* qui permet de sortir du *switch* sans passer par les autres *case*.

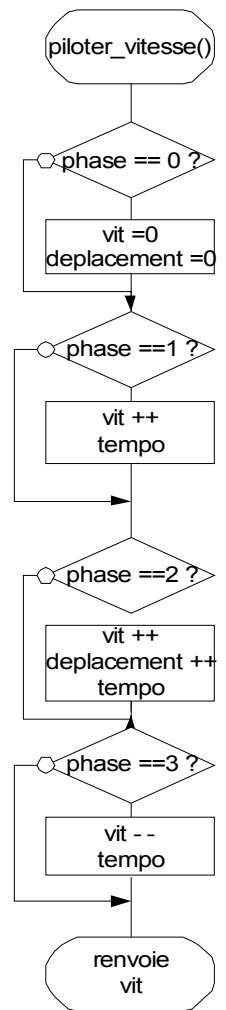
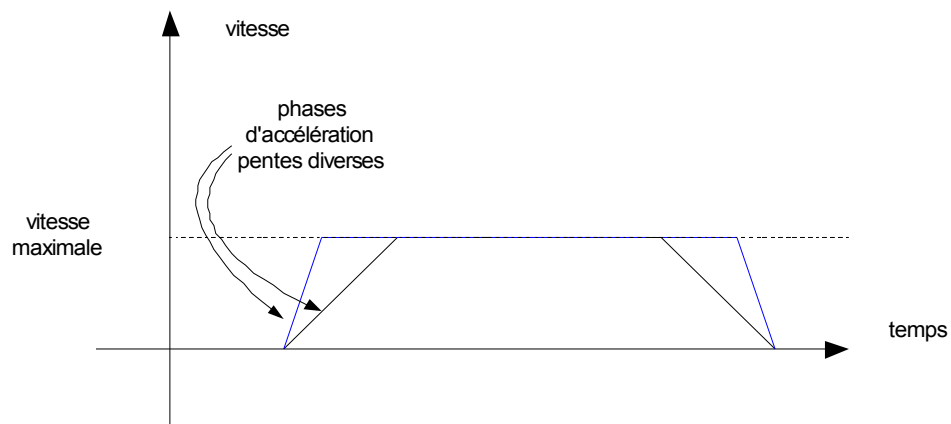
Cette fonction ne retourne rien, mais permet de faire évoluer les phases en modifiant la variable **phase**. Cette variable sera utilisée ensuite dans la fonction `piloter_vitesse()`.

Détermination de la vitesse:

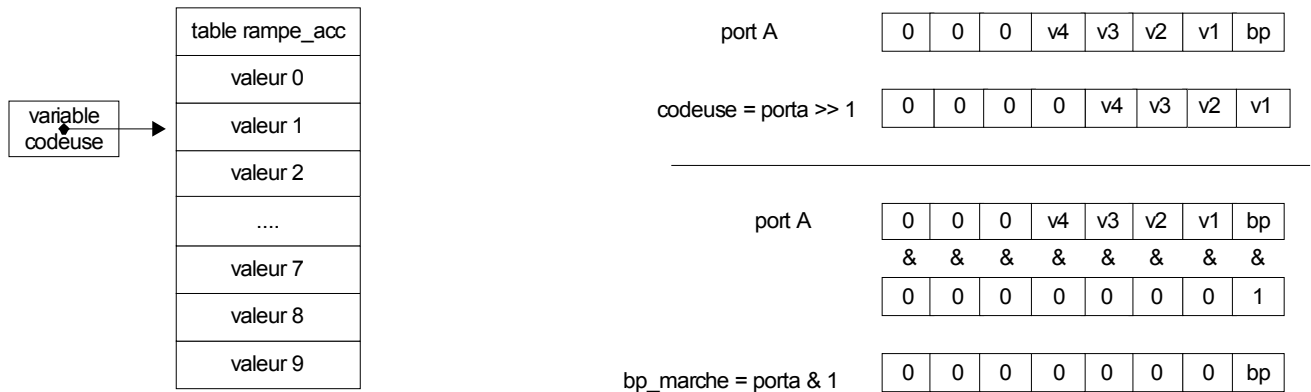
La fonction **piloter_vitesse()** a pour but de déterminer la vitesse à laquelle doit tourner le moteur de la bande.

Cette vitesse dépendant de :

- la phase (étape de grafcet)
- la vitesse précédente ou le déplacement calculé pour le palier



détermination de la temporisation (accélération) par utilisation d'un pointeur sur une table



Dans cet extrait du programme, on peut voir que la temporisation est réalisée en fonction de **codeuse**. Cette méthode est utilisée pour que la variable **codeuse** serve de pointeur dans la table **rampe_acc**. Ce qui permet de programmer la pente en fonction de la roue codeuse, et de sauvegarder ces valeurs dans une table, et non dans une sous fonction comme cela était réalisé précédemment.

gestion des tests (simulation)

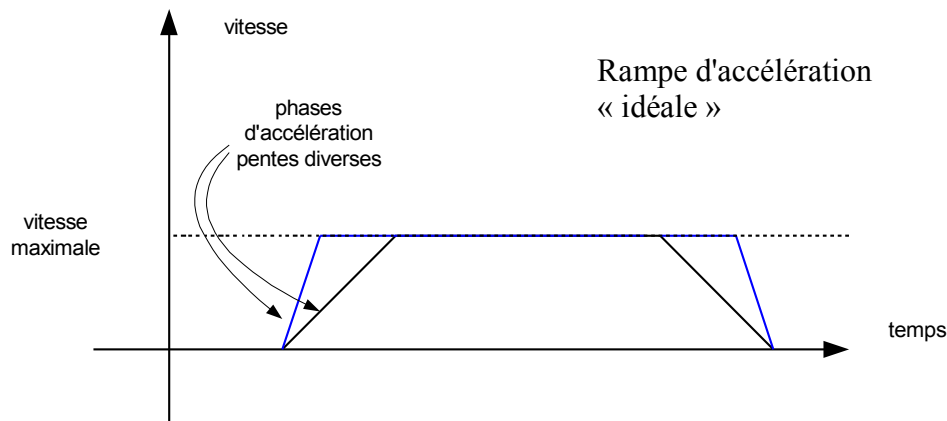
- test du fonctionnement du front montant par incrémentation d'une variable à chaque front, et affichage de cette variable sur la sortie.
- Programmation du grafcet avec passage d'étapes en étapes par front montant du bouton marche/arrêt et affichage du numéro de l'étape sur les sorties.
- Programmation du grafcet sans possibilité d'arrêt par bouton marche/arrêt .
- Test gestion pointeur sur la la table avec affichage sur la sortie.

résultats des tests :

test OK

Détermination des temporisation

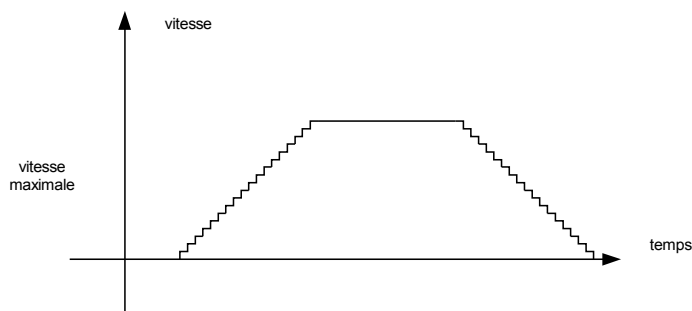
Pour éviter que la canette ne se renverse, nous avons du créer des rampes d'accélération. Ces rampes d'accélération sont en fait réalisées en « escalier », de la façon suivante :



Sur ce schéma, on peut voir que l'accélération est représentée par une droite.

Le CNA est commandé sur 8 bits, donc la pleine échelle du CNA est composée de 2^8 valeurs, c'est à dire que la pente est constituée de 255 paliers.

Pour réaliser ces marches, le programme répète 256 fois cette partie :



```
for(cpt=0;cpt<255;cpt++)
{
    sortie=cpt+1;           //car cpt max = 254
    delay(tempoAccelDecel);
}
```

Pour déterminer la valeur de « tempoAccelDecel », j'avais le résultat des études de la partie mécanique, c'est à dire une accélération maximale en fonction d'un angle. Pour faire la suite, des relations de cinématique du point ont été utilisées :

n°position	Valeurs de α	Valeurs de t_{max}	$a_{max}=(-0,1737\alpha+1,9666)$
1	0	0,13	Après intégration,
2	2,5	0,17	$v_{max}=(-0,1737\alpha+1,9666)t_{max}$
3	5	0,24	D'où
4	7,5	0,39	$t_{max}=v_{max}/(-0,1737\alpha+1,9666)$
5	10	1,13	Des calculs mécaniques, ainsi que des relevés, ont permis de
6	11,27	30,81	déterminer la vitesse maximale de la bande :
7	13,77	-0,61	$v_{max} = 0,26 \text{ m/s}$
8	16,27	-0,3	De ces relations, on peut tirer les durées totales des phases
9	18,77	-0,2	d'accélération pour des positions prédéfinies (ci contre). Les valeurs
10	21,27	-0,15	de t_{max} négatives correspondent à un angle où la canette doit se

renverser, quelque soit la vitesse de démarrage.

La valeur obtenue correspond au temps total de l'accélération; Pour obtenir la valeur de chaque « pas », il faut diviser cette valeur par 255. Celle ci descend donc au dessous de la seconde, et ne peut être réalisée avec une fonction *delay(n)* qui attend *n* secondes.

J'ai donc utilisé la fonction *microdelay(n)*. Cette fonction attend $3n+1$ microsecondes. Pour obtenir le n approprié, j'ai modifié ma feuille de calcul :

Cependant, certaines valeurs sont au delà de 255, ce qui pose problème, car les seules chaînes acceptées par le PIC sont des octets (de 0 à 255).

Valeurs de tempo	Valeur en μs	Val $\mu delay$	Val $\mu delay$
0,0005163	516,29	171,1	171
0,0006627	662,66	219,89	220
0,0009249	924,86	307,29	307
0,0015304	1530,39	509,13	509
0,0044325	4432,46	1476,49	1476
0,1208228	120822,84	40273,28	40273
-0,0023836	-2383,55	-795,52	-796
-0,0011801	-1180,13	-394,38	-394
-0,0007842	-784,2	-262,4	-262
-0,0005872	-587,2	-196,73	-197

Pour pallier à ce problème, j'ai utilisé quatre fonctions *microdelay()*, qui permettent de diviser encore les valeurs n par quatre les faisant toutes rentrer dans un octet.

Tableau des valeurs finales :

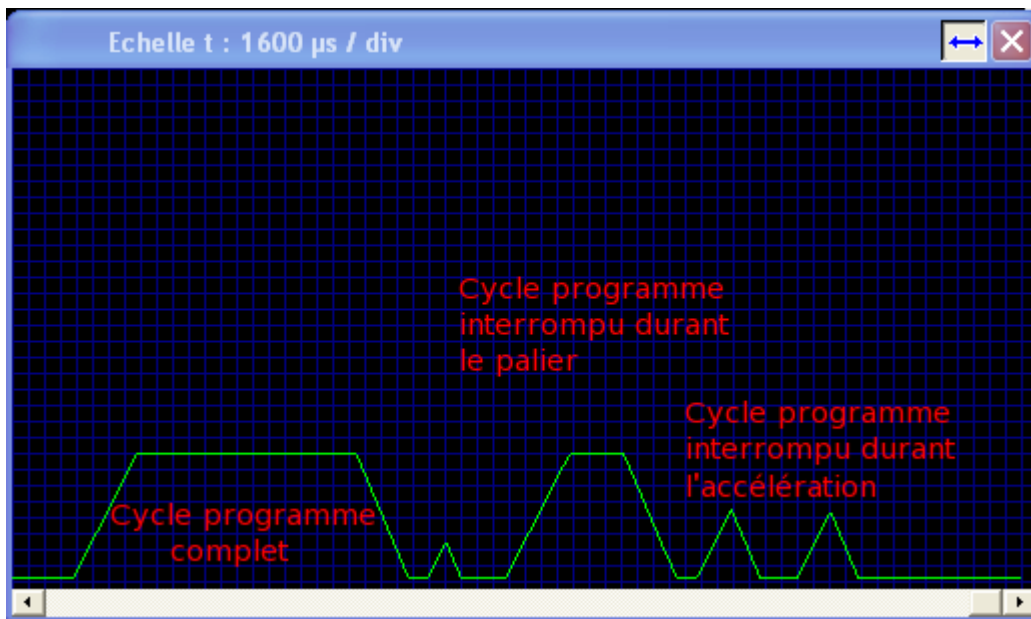
Val $\mu delay / 4$
43
55
77
127
369
10068
-199
-99
-66
-49

Problèmes rencontrés et tentatives de résolution

Lors des dernières séances de PPE, notre système, qui avait bien fonctionné jusqu'ici, a commencé à ne plus fonctionner correctement. Pour tenter de résoudre nos problèmes, nous avons utilisés divers moyens de débogage. Pour ma part, j'ai eu recours à des simulations sur le logiciel de développement, DevPic84c.

Pour tester ce programme, le troisième, j'ai utilisé une fonction du logiciel qui permet de simuler un CNA sur le port B. Cette fonction permet d'observer la sortie d'un CNA virtuel.

Voici le relevé documenté de ce programme. On peut y voir les 3 phases de fonctionnement normal,



c'est à dire l'accélération, le palier à vitesse constante, la décélération.

Ce programme permettant aussi d'arrêter le cycle durant n'importe quelle phase, j'ai fait les essais correspondants durant la phase d'accélération et celle de palier.

Nous avons également repris les programmes initiaux, qui avaient été essayés et validés sur la maquette physique.

Malheureusement, tous ces tests n'ont pas réussi et le système n'est pas opérationnel, bien que le programme fonctionne séparément sur l'ordinateur.