

CNAM

ELECTRONIQUE ELE118

Programmation avancée des microcontrôleurs

----- TP -----



Environ 4 à 5 séances de manipulation

COMPLEMENT SUR LA MAQUETTE ET L'OUTIL DE DEVELOPPEMENT

PROJET :

→Partie principale

Programmation en C sur HC12 d'une application, recherche du multitâche et du temps réel sans exécutif. (Moteur pas à pas, Timer, CAN ...) Mise en ROM de l'application.

→ Si possible... Etude d'un petit exécutif temps réel personnel (Fonctionnant en temps partagé).

ANNEXES : Vecteurs d'interruption HC12
 Mouvements de la pile
 Instructions assembleur HC12 (voir Cours première partie)

Rappel : deux poly de cours sont disponibles pour ce cours ELE118 :

- 1 ELE118 Cours première partie**
- 2 ELE118 Cours seconde partie**

→En plus d'un compte rendu de TP classique, pour bien repérer les questions demandant obligatoirement une réponse dans le compte rendu sont indiquées dans la marge gauche par le signe (sorte de parchemin) : 

Par G. PALLOT

SOMMAIRE

1.	MAQUETTE ET OUTIL DE DEVELOPPEMENT	1
1.1.	Le matériel utilisé	1
1.2.	Possibilités d'entrée sortie de caractères	4
1.3.	Le logiciel de développement « IAR System »	5
2.	PARTIE PRINCIPALE DU PROJET	17
2.1.	Introduction	17
2.2.	Commande du moteur pas à pas sans interruption	18
2.3.	Commande du moteur pas à pas par interruption	21
2.4.	Application définitive	25
2.5.	Affichage de la vitesse en tours/secondes	28
2.6.	Commande de la vitesse par clavier	31
2.7.	Mesure d'une tension	34
3.	TEMPS PARTAGE	37
3.1.	Etude d'un programme de démonstration	38
3.2.	Développement d'une application	43
4.	ANNEXES SUR LE HC12 : PILE ET VECTEURS D'INTERRUPTIONS	47
4.1.	Evolution automatique du pointeur de pile en HC12	47
4.2.	Vecteurs d'interruption HC12	47

1. MAQUETTE ET OUTIL DE DEVELOPPEMENT IAR

1.1. Le matériel utilisé

Maquette de développement, dans un coffret avec face avant transparente :

- Une **carte Axiom CML12SDP256** à base de processeur **type HC12 : le MC9S12DP256**.
- Des extensions (câblées non pas sur le bus de l'HC12 mais ce qui se fait de plus en plus, sur ses ports parallèles et séries).

Outil de développement :

- Un ordinateur de type PC.
- Un logiciel de développement **IAR System**, tournant sous Windows XP

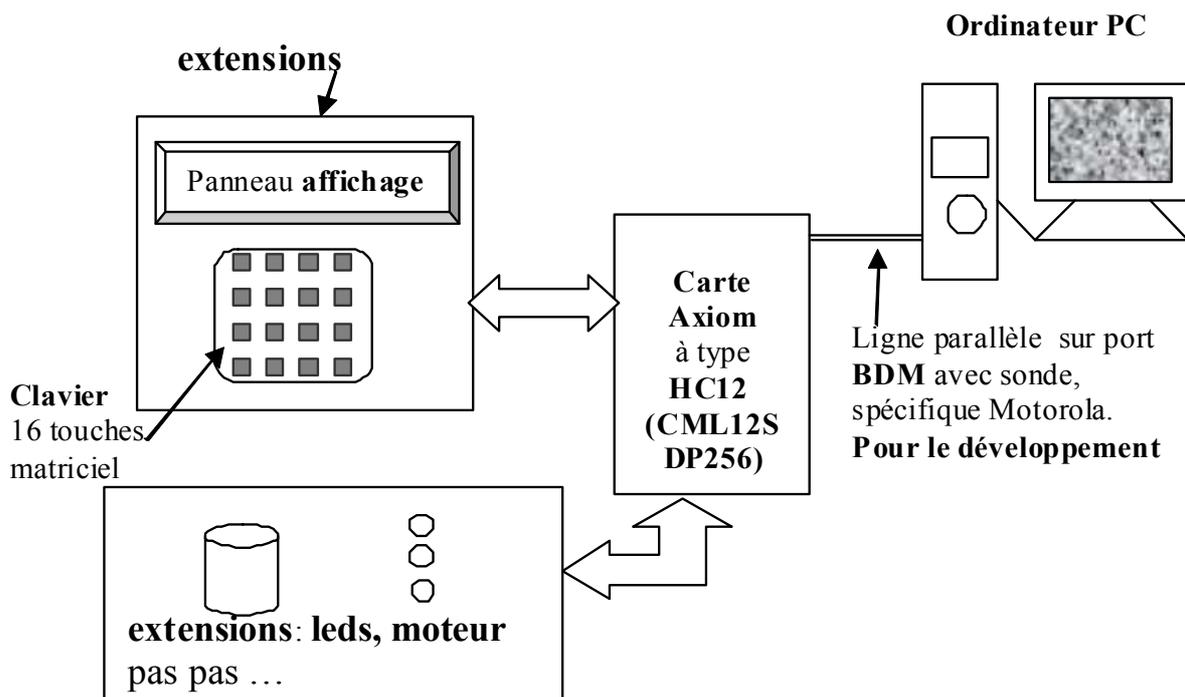
On édite, compile, assemble, ... sur le PC.

On peut utiliser un **simulateur** sur le PC.

On **télécharge** le code sur la carte cible en **mode mise au point**.

Possibilité en exécution et en phase de mise au point d'utiliser le PC comme terminal I/O.

L'HC12 possédant un **FLASH interne**, l'**application définitive** se programme sans utilisation d'un programmeur de PROM externe.



➤ **Présentation brève des éléments de la maquette de développement :**

L'HC12 possédant de nombreux ports, les diverses extensions (Clavier, panneau cristaux liquides, moteur ...) sont câblées non plus directement sur le bus, mais comme on le fait beaucoup désormais sur les ports eux mêmes : séries (bus IIC), séries rapides (bus SPI) ou parallèles.

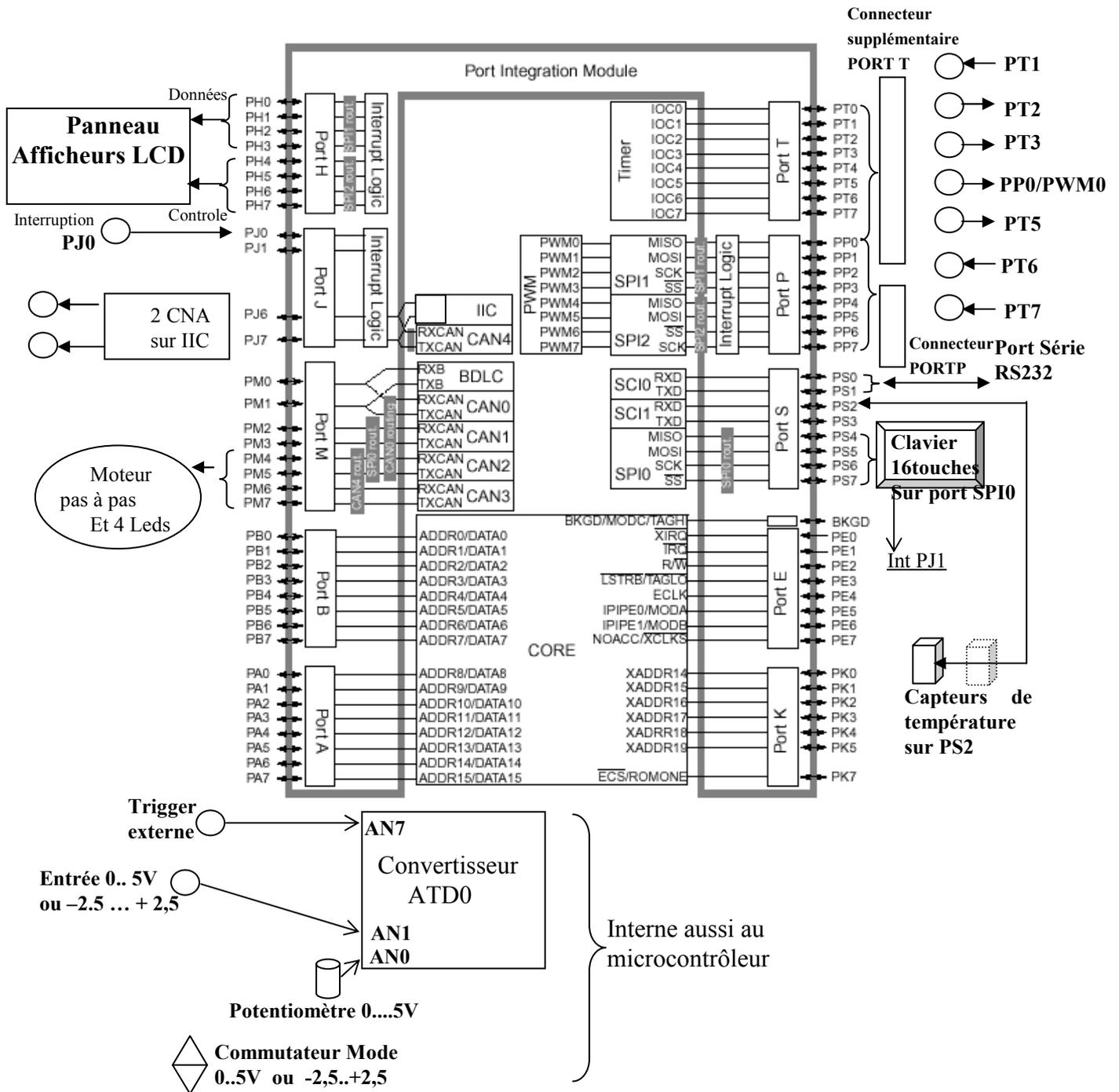
Le tableau et schéma suivant résumant les différents éléments, toutes entrées et sorties protégées par Ampli op et diodes zener.

<i>Eléments</i>	<i>Sur</i>
Moteur pas à pas et 4 leds	Port M (Bits PM7..PM4)
Panneau Cristaux liquides , à contrôleur intégré.	Port H
Circuit Double CNA	Bus IIC
1 ou 2 Capteurs de température	Bit PS2 du Port S
Clavier 16 touches	Bus SPI0 avec circuit interface série rapide //, ligne d'interruption PJ1
Potentiomètre et tension de 0 à 5v	Entrée analogique AN0
Ligne d'entrée analogique (avec interrupteur pour travail en 0..5V ou -2,5V... + 2,5V)	Entrée analogique AN1, Ze 10kΩ
Entrée Trigger (pour échantillonnage)	AN7
Diverses entrées et sorties sur le port T ou Timer et pulse Accumulateur.	Entrées PT1, PT6, PT7 Sorties PT2, PT3, PT5
Une ligne d'interruption	Bit PJ0 de PORTJ
Ligne Sortie du Port P, ou Sortie PWM	PP0/PWM0 (avec et sans filtrage)
Deux connecteurs supplémentaires (internes)	Sur Port T et Port P.

ATTENTION : pour **F_BUS > 4MHZ Debug** et même lancement par **GO Impossibles !!!!**

Il faut alors Télécharger par le Debug

Lancer juste par le bouton **Reset**, si accessible sur les maquettes. Sinon couper l'alimentation et rallumer !



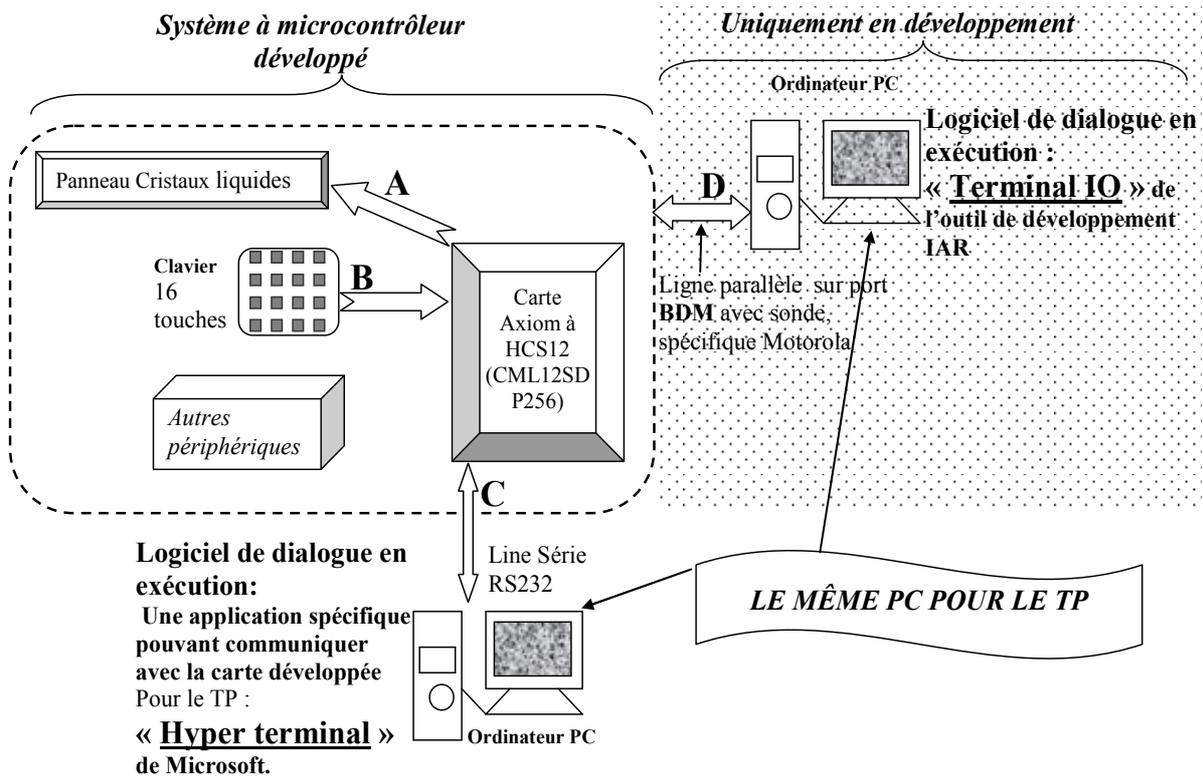
Du fait de la présence de périphériques sur des ports du microcontrôleur, la carte utilise des **driver** et des **routines d'initialisations** pour les ports tels que **SPI**, bus **IIC**.

Avec le même quartz de 4MHz, La fréquence **F_BUS** qui est **par défaut de 2MHz**, peut être modifiée comme suit : **4, 8, 16** ou **24MHz** (Existence d'une PLL dans l'HC12).

→ On pourra donc toujours **laisser** la fonction **ini_carte()** ; juste en début du main()

On notera qu'elle n'est **vraiment nécessaire** que pour l'usage du **clavier**, du périphérique **CNA**, **et si on désire modifier F_BUS**. Si on ne met pas cette fonction, et si un message apparaît disant que **F_BUS** inconnu, placer alors **#define F_BUS 2**

1.2. Possibilités d'entrée sortie de caractères



→ Lignes de dialogues pour l'application développée

- A Envoi de caractères (codes ASCII) : Fonctions spécifiques
- B Lecture de codes de touches : Fonctions spécifiques
- C Ligne série RS232 de dialogue avec un ordinateur : Fonctions spécifiques

→ Lignes n'existant que lors du développement de l'application

- D Téléchargement du programme, exécution pas à pas et Debug
- Entrée sortie de caractères en exécution: Fonctions du C Standard (printf, scanf)

EN DÉVELOPPEMENT SEULEMENT

On utilise évidemment un ordinateur (pour compiler, télécharger, Débugger ...).

Un Terminal IO en Debug permet d'utiliser les fonctions standard du C : putchar, printf, scanf comme sur un PC muni d'un système d'exploitation. Ici l'affichage se fait sur le terminal de développement et son interface IO (par l'intermédiaire de la liaison parallèle BDM et la sonde de développement. Donc n'existe évidemment pas sur l'application finale.

SUR L'APPLICATION FINALE

Si l'appareil est prévu pour dialoguer avec une application exécutée sur un ordinateur (sur ligne série ou USB), on doit alors développer des fonctions spécifiques, en utilisant des driver pour faciliter la programmation.

POUR NOS TP

On utilise le **même ordinateur** pour le développement et comme terminal devant dialoguer avec la carte pour l'application finale étudiée.

→ **Sauf cas particulier de plus gros systèmes, pour des cartes à microcontrôleur, printf et scanf ne fonctionnent à priori que sur l'écran de Debug.**

1.3. Le logiciel de développement « IAR System »

L'utilisation de l'interface de travail se découvrira petit à petit en TP, une démarche principalement heuristique est finalement préférable.... On présentera ici seulement quelques points essentiels.

Sous windows NT ou 2000, vous devrez vous connecter en tant que :

User : hc12

Password : hc12

Vous ferez attention à respecter les extensions (.xxx) de fichiers suivantes :

- .prj** projet
- .h** fichiers.h classiques du C
- .s33** source assembleur
- .c** source C
- .r33** codes objets partiels ou final.
- .a33** fichier binaire codé ASCII format S1/S9 pour téléchargement sur programmeur de PROM. Non utilisé ici.

Si un **mot de passe réseau** vous est demandé, c'est **cnam**, tout simplement

→ Complément sur les fichiers **.a33** (S1/S9)

L'HC12 possédant de la FLASH interne pour l'application définitive, **ils sont inutiles, le téléchargement standard se faisant comme en développement dans un format natif propre à IAR.**

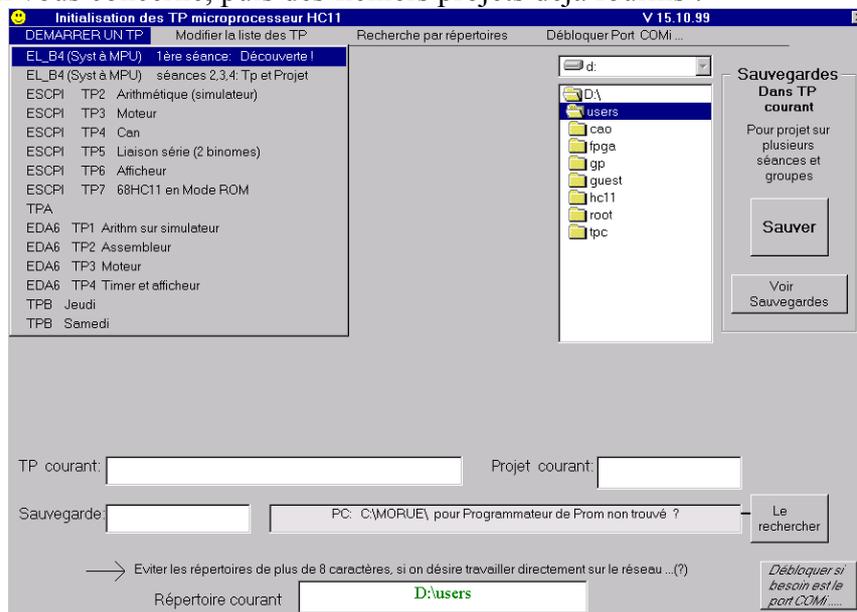
On peut cependant (par option) générer un fichier **.a33** de type S1/S9 (ou d'autres formats polyvalents comme par exemple **.695** de type ieee-695), pour s'adapter à d'autres cartes cibles et outils de développement.

1.3.1. Ouverture d'un projet, et commandes principales

L'outil a besoin pour chaque programme principal d'un projet. Celui ci, enregistré sous un nom muni de l'extension **.prj**, comprend en plus des fichiers sur lequel on travaille, les options importantes de compilation, d'assemblage, d'édition de lien.

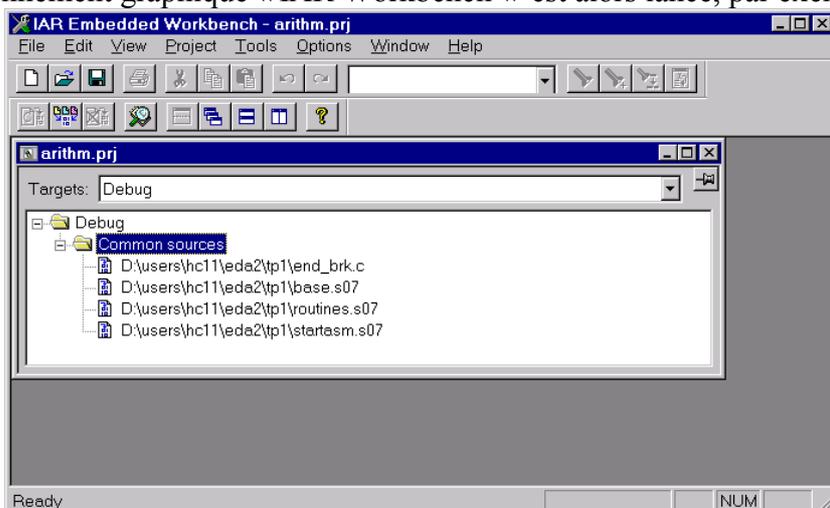
Vous disposez au départ d'un interface supplémentaire écrit en Visual Basic, qui vous permet de travailler dans des répertoires précis.

Démarrer le en cliquant 2 fois sur l'icône  **HC11**. Une fenêtre s'ouvre dans laquelle on peut choisir le tp qui vous concerne, puis des fichiers projets déjà fournis :



- Choisir dans le menu «**démarrer un TP**» le T.P correspondant au travail que vous allez réaliser (par exemple ici: **EL_B4 1ère séance Découverte**).
A ce moment vous pouvez choisir :
 - **Initialiser** (le mot de passe est **cnam**)
 - **Reprendre le TP courant**,
 - Reprendre un **TP sauvegardé** lors d'une séance précédente sous un nom de binôme.

- Choisir ensuite le projet sur lequel vous travaillerez qui peut être :
 - Un nom standard **base.prj**
 - Un nom spécifique au TP comme par exemple : **etape1.prj**, **exercice1.prj**, etc.
 L'environnement graphique « IAR Workbench » est alors lancé, par exemple :



Des fichiers peuvent figurer déjà dans le projet, il est possible de modifier par la commande **Projet / Files** et etc ...

Les fichiers sans chemins spécifiés sont dans le **répertoire courant** du .prj
Sinon les chemins sont bien indiqués.

Avertissement : **Ne pas chercher à créer un projet avec un nouveau nom**, l'assemblage ou plein d'autres choses ne fonctionneront pas sans modifications de quelques options.
On peut par contre prendre **base.prj** si il existe ou un **autre existant**, et **le sauver dans le même répertoire (à vérifier !)** sous un autre nom. On modifiera ensuite sa liste de fichiers de travail.

1.3.1.1. Projets en Assembleur seul

Moins pratique, car cet outil est fait pour travailler dans un environnement C avec compilateur ...

Dans un but pédagogique, on travaillera ainsi au début, mais on passera très vite à l'environnement C.

Les **fichiers de commande pour l'édition de lien** sont des fichiers minimum situés dans un sous répertoire du compilateur, nommés :

ABSOLU_ASM.xcl cas d'un travail uniquement par sections absolues
RELOGEABLE_ASM.xcl cas de travail avec sections relogeables

On peut vérifier que l'on travaille bien sur ces fichiers de commande dans le menu **Projet/Options/XLINK/Include** (et éventuellement corriger...).

1.3.1.2. Projets en langage « C » ou C et assembleur

- ◆ Le projet devra obligatoirement contenir :

base.c ou **autre.c**

votre programme principal en « C »

(ou **n'importequoi.c** structuré comme base.c)

routines_mini_c.s33 (fonctions assembleur de validation des interruptions générales et simple fonction tempo par boucles)

On peut y ajouter :

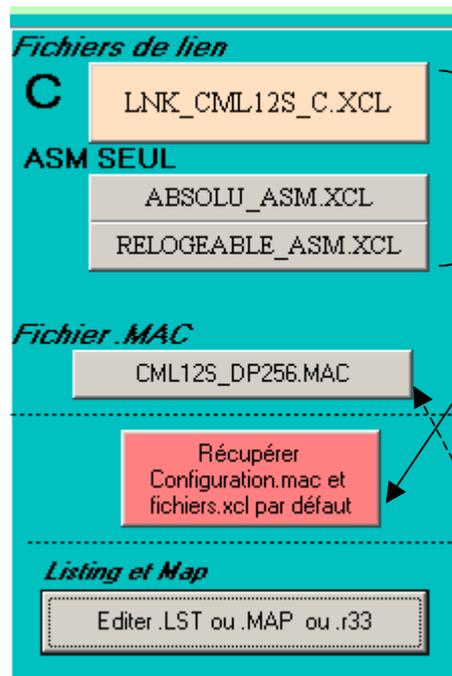
- **routines_aff_c.c** contenant des routines personnelles de conversion et d'affichage sur l'afficheur à cristaux liquides.
- On ajoutera plus tard d'autres fichiers **.C** ou **.S33** (par exemple des routines de gestion du clavier, des routines pour le temps partagé)

- ◆ **L'éditeur de lien en C** doit travailler avec le fichier général de commande **LNK_CML12S_C.xcl** (le vérifier si des messages ésotériques surviennent).

On peut vérifier que l'on travaille bien sur ce fichier de commande dans le menu **Projet/Options/XLINK/Include** (et éventuellement corriger...).

1.3.1.3. Examen ou modification des fichiers de link

Dans l'interface Visual Basic :



On peut:

- Lire et éditer les différents fichiers.xcl
(On peut voir alors aussi le chemin exact de ce fichier, situé dans un sous répertoire du compilateur)

- Récupérer les fichiers par défaut.

Ne pas trop s'occuper ni modifier le fichier.mac qui est un fichier de renseignements sur le type de processeur utilisé, et le mode de téléchargement.

Ce fichier est d'ailleurs vraiment très spécifique de du HC12 et de IAR, et n'existe pas pour de nombreux autres microcontrôleurs et systèmes de développement.

1.3.1.4. Compilation, assemblage, édition de lien, Debugger, téléchargement

- ◆ Utilisez les commandes du menu **Project**:

- **Compile** pour compiler ou assembler un fichier particulier.
- **Build All** du menu **Project** pour créer un fichier exécutable.

En cas d'erreurs de compilation ou à l'édition de lien, consultez les indications de la fenêtre **Messages**. A chaque type d'erreur correspond un code spécifique (la localisation de l'erreur s'effectue facilement en double cliquant sur le message d'erreur). La mise au point d'un programme consiste donc dans une première phase à la correction des erreurs de syntaxe.

- **Debugger** effectue Buid All suivi du lancement du Debugger et téléchargement.

◆ La vérification du fonctionnement (**Debugger**) peut s'effectuer de deux manières :

- *en mode simulateur* (HC12 simulée de manière logicielle). Dans ce mode, et seulement dans celui ci, on a accès à un **compteur de Tck** (dans la fenêtre registres) pour mesurer le **temps d'exécution** d'un programme ou d'une partie de programme.

Seul le processeur de l'HC12 peut être simulé, et non ses périphériques internes ni encore moins externes. Donc mettre en commentaires ou modifier certaines lignes....

- *en mode moniteur* (utilisation du port spécial BDM de l'HC12 pour le dialogue avec le PC et CSPY de IAR). On rappelle que l'HC12 peut se passer de programme moniteur pour la mise au point. La carte d'application est alors fonctionnelle et tous les périphériques.

Ce choix de mode de fonctionnement est une option du projet (commande **Option/Project** puis **C_SPY** : on modifie la cible (Target) en choisissant le driver : *Simulator* ou *BDM P&E Micro MULTILINK parallèle port*).

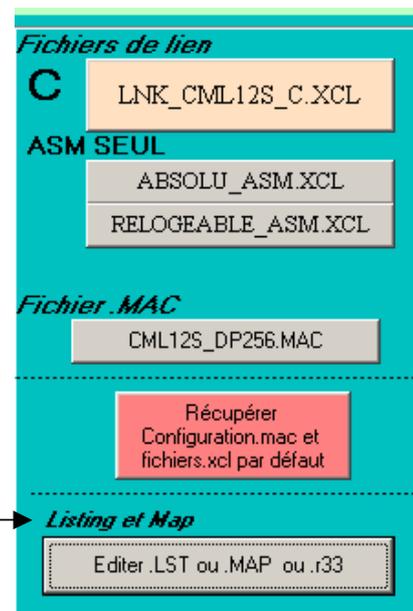
1.3.1.5. Consultation des fichiers listing (.lst) et plan mémoire (.map)

Il faut pour que ces listing soient générés, que dans le menu **Project option** (du Work Bench) les options de génération de listing soient bien cochées dans les catégories :

Compilateur (ICC6812)
Assembleur(A6812) Link(XLINK).

C'est normalement le cas par défaut.

Dans l'interface Visual Basic, la fenêtre ci-contre (déjà vue précédemment pour examiner les fichiers de link) permet l'**accès à ces fichiers**, et par curiosité de voir où ils sont générés.

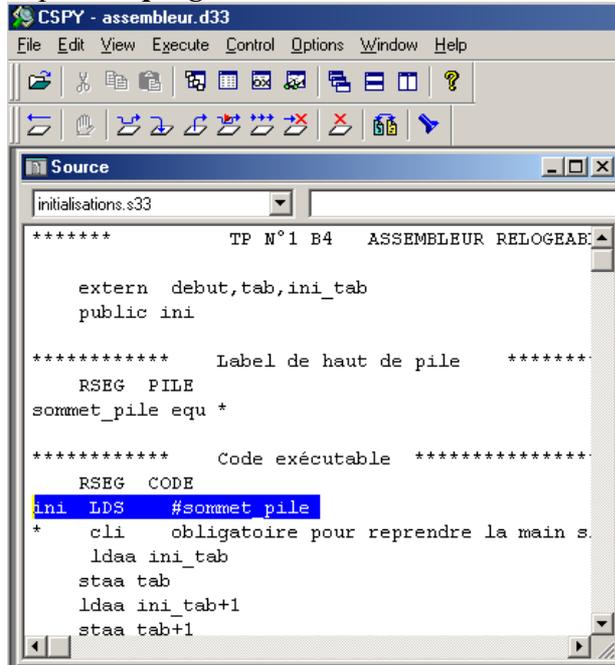


1.3.2. Debug IAR Systems, simulateur ou carte

Le Debugger IAR se nomme « CSPY »

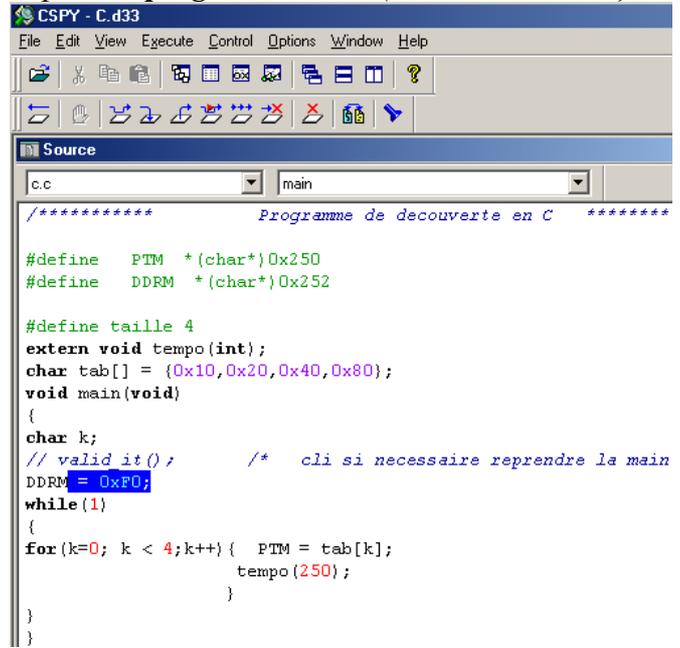
Démarez en exécutant la commande Debugger du menu Project. (Si un code objet est à recréer, un Build All est exécuté automatiquement). Le téléchargement (d'un fichier.d33 en format natif IAR) s'effectue. On obtient par exemple :

→ pour un *programme tout assembleur* :



Le Point d'entrée est en bleu : le « start up » avec initialisation du pointeur de pile

→ pour un *programme en C (et c + assembleur)*:



Le Start up a déjà été exécuté (initialiasant le pointeur de pile et les variables globales et statiques initialisées.

Le point d'entrée est en bleu, première ligne en C.

1.3.2.1.1.1. Commandes de base

- **Execute/Go ou F4 Exécution du programme en temps réel**
(L'option Real Time dans Control ne sert à rien : en DBM la commande Go provoque toujours l'exécution en temps réel si pas de points d'arrêt).
- **Execute/Step ou F2 «pas à pas»** d'un programme sans entrer dans les fonctions écrites dans le même langage.
Pour entrer alors dans une telle fonction, utiliser **Execute/Step into** ou touche **F3** du clavier.
- Icône main :  Pour stopper le déroulement d'un programme tournant sans fin. C'est la seule façon ! (Le PC envoi en fait par la ligne série un caractère, ce qui déclenche une interruption). **Ne pas faire de Reset Hardware** sinon on plante de Debugger !
- **Click droit et Toggle Breakpoint** placer et enlever des points d'arrêt.
Edit Breakpoint Surtout pour tous les supprimer
- **Execute/Reset** Avant d'exécuter une seconde fois ou de refaire du pas à pas sur de votre programme Cette commande est indispensable pour réinitialiser.
- **View/Toggle Source/disassembly ou F8** pour faire apparaître le code source (en assembleur ou en « C »). Par défaut, on est en mode code C.

• Les menus Window

Vous permettent d'activer les fenêtres que vous désirez. Les fenêtre **Source** et **Report** sont ouvertes par défaut, ne pas les fermer ! On peut activer aussi les fenêtres :

- **Register** : état en pas à pas des registres internes du 68HC11.

En **mode Simulateur** seulement, on peut ici **mesurer des temps d'exécution** par le compteur de **Tck**

- **Terminal I/O** : qui sert de fenêtre d'affichage en **Debug** seulement acceptant la fonction printf(...) du C (Et des routines d'affichages de routines.s07). Scanf(...) du C est également possible.

- **Watch** : Examen de variables

Faire **click droit dans Expression** pour introduire nouvelle expression à examiner.

Un + devant un tableau indique que l'on peut le développer. Pratique pour examiner tout le contenu.

Attention : les variable locales ne peuvent être évidemment visualisées que si on est dans la fonction concernée !

Expression	Value
k	0
tab	0x3000
[0]	16
[1]	32
[2]	64
[3]	-128

1.3.2.1.1.2. Problèmes de temps réel en mode Debug

- Step ne saute par les fonctions C écrites en assembleur (car changement de type de langage) Step into est alors identique ! C'est bête.
- **Le processeur est très ralenti en pas à pas.** Certaines lignes de C peuvent mettre ½ seconde ou plusieurs secondes à s'exécuter.

Dans un programme tout assembleur, une fonction tempo en assembleur de par exemple 200ms peut mettre plusieurs minutes à s'exécuter, et on a l'impression d'être planté !

Dans un programme C, un pas à pas en C exécutant la fonction tempo en assembleur fait rentrer dans la fonction !

Un pas à pas en C sur une fonction en C avec de nombreuses bouclent peut être aussi très longue à s'exécuter !

Donc ne pas lancer en pas à pas l'exécution de fonctions possédant de longues boucles.

Remèdes :

Mettre des **points d'arrêt**, et faire des **Execute/GO** ou F4. L'exécution (bien que pas tout à fait temps réel est alors bien plus rapide).

Ou supprimer les tempos en pas à pas.

• Pas à pas et interruption

Mettre un point d'arrêt dans le programme d'interruption, et faire GO. Un pas à pas est alors possible dans le programme d'interruption.

Plus délicat si plusieurs interruptions possibles. N'en valider qu'une seule.

1.3.2.1.1.3. Quitter C-SPY

Attention : Pour rééditer votre programme ou en écrire un autre, vous devez toujours quitter et fermer CSPY.

CSPY s'ouvre alors automatiquement sur un nouveau Build all

1.3.3. Etude plus détaillée des options d'un fichier de projet

Le constructeur d'un compilateur fournit un certain nombre d'options de travail par défaut qui ne correspondent pas toujours ni même souvent à la configuration de travail souhaitée (logiciel et matériel).

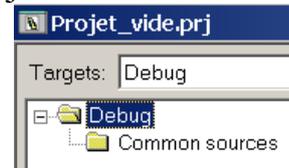
Important : Si par hasard vous faites New Project, vous vous retrouvez avec un projet vide et avec des options de base qui ne conviennent a priori pas du tout à votre environnement !
 Donc en pratique, on reprendra toujours un fichier projet.prj correspondant à son environnement de travail (options matériels et logiciels, fichiers de bases...), et on modifiera juste la liste des fichiers pris en compte.
Ne jamais faire File New Project !!! avec ce compilateur, il faudrait revoir toutes les options !

Mais si vous devez un jour vous adapter à un autre environnement, il sera nécessaire de connaître les différentes options. C'est ce que nous voyons ici.

1.3.3.1. Accès aux options

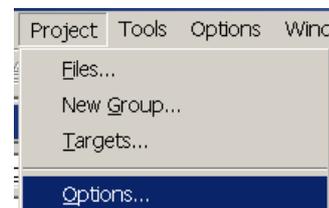
1) Dans l'interface Visual Basic, choisir un projet : « .prj »

→ **Attention**, pour voir toutes les options, il faut que soit sélectionné (bleui) Debug dans la fenêtre du projet, qui est ici : Projet_vide.prj (bizarre, mais that's life).



2) Faire Projet/option :

→ Pour chaque cas, on devra comprendre les différentes options (voir les commentaires associés).



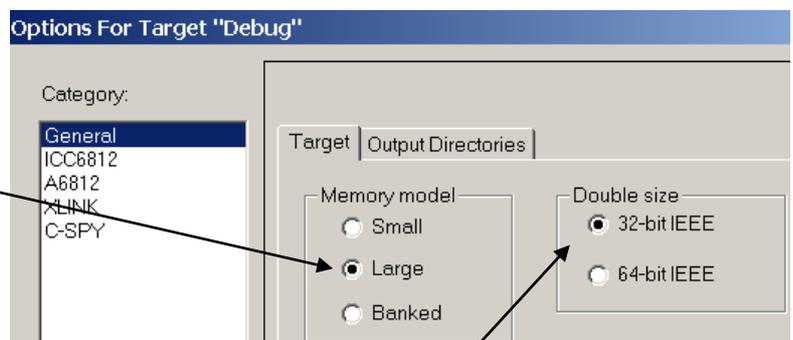
On trouve 5 Catégories, et pour chacune des onglets.

1.3.3.2. GENERAL

→ Target

Large

Le mode Banked permet de travailler par Page pour étendre l'espace mémoire (non étudié ici).



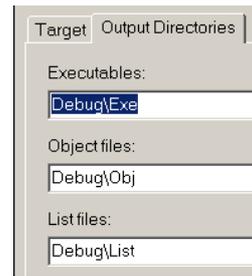
32_bits_IEEE (choix de flottants simple ou double précisions).

GENERAL

→ **Output Directories**

Les sous répertoires du répertoire de travail ou seront générées les différents fichiers.

Informations par défaut.

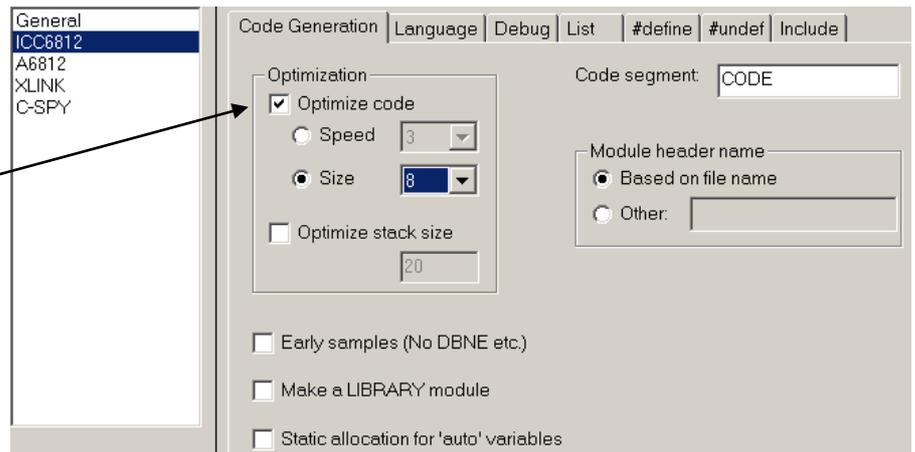


1.3.3.3. ICC6812 COMPILATEUR

→ **Code Generation**

Niveaux d'optimisation.

Ici : **Optimize code**, avec Niveau **8** par exemple pour la **taille** ou la **vitesse** (ce qui revient en fait souvent au même..)



ICC6812

→ **Language**

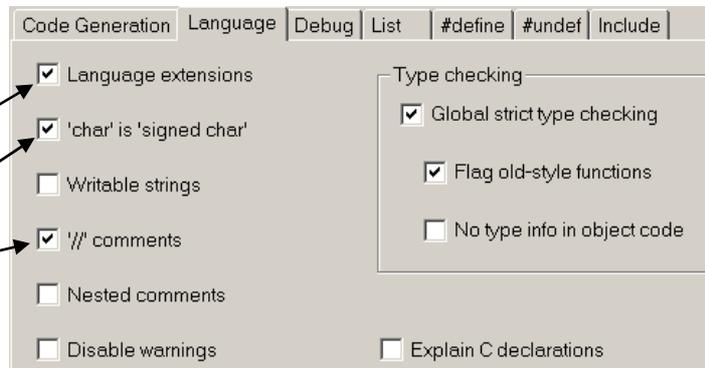
Des options très utiles à ajouter et donc à cocher.

Language extensions.

Travail avec des **char signés**

Commentaires sur une ligne avec

// (option C++ en fait)



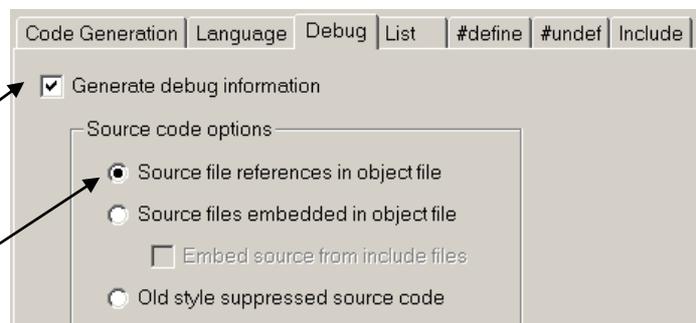
ICC6812

→ **Debug**

Deux Options importantes déjà cochées :

Generate Debug information

Source file references in object file (moins de place en mémoire que le code source en entier).

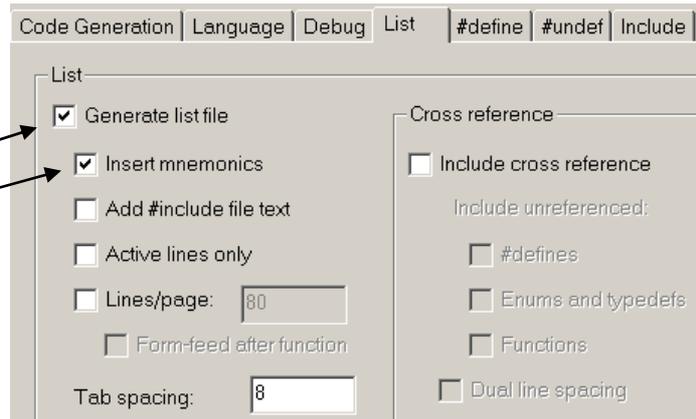


ICC6812

→List

Coché pour visualiser un listing contenant assembleur avec lignes de C incorporées :

Generate List File
Insert Mnémonic



ICC6812

→Include

\$TOOLKIT_DIRS désigne ici le répertoire (avec chemin complet) ou est installé le compilateur IAR.

ici en fait:

C:\users\hc12\hc12_tools\wbench\6812\

\inc un sous répertoire nommé inc.



Ne rien modifier, c'est le chemin des **entêtes : fichiers .h** du C (telles que stdio.h, math.h, stringf.h)

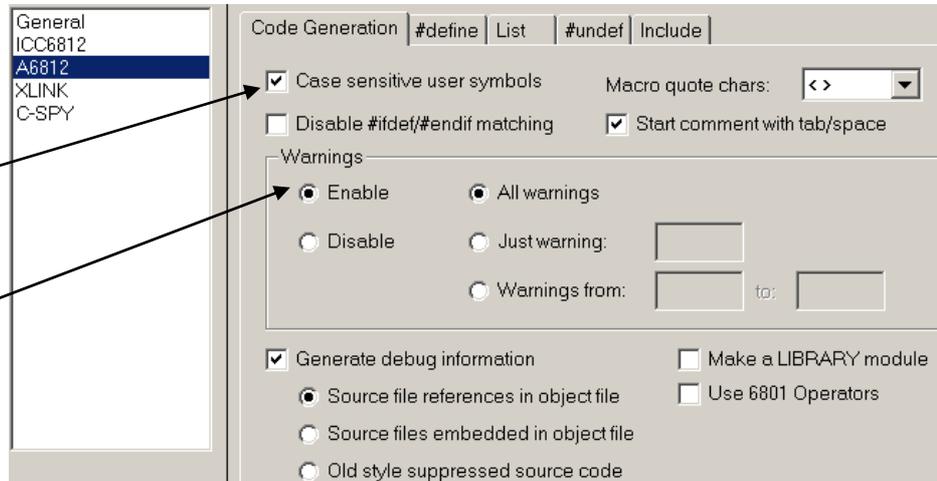
1.3.3.4. A6812

ASSEMBLEUR

→Code Generation

Par défaut :
Symboles sensibles à la casse (majuscule et minuscule)

Enable Wanings



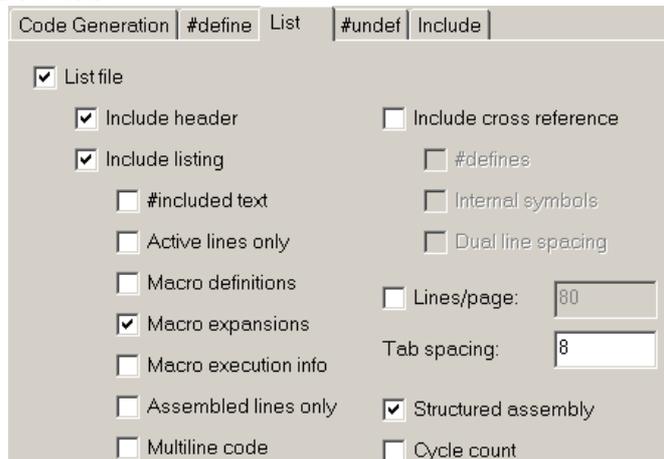
C'est mieux, car certains warning sont dangereux, pas d'autres. L'utilisateur est ainsi averti.

A6812

→List

listing du résultat de l'assembleur (avec code objet).

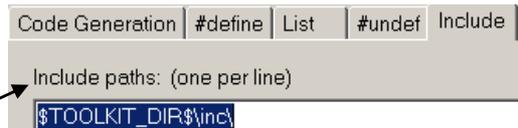
Options très utiles :



A6812

→Include

Comme plus haut, emplacement de fichiers .h du compilateur.



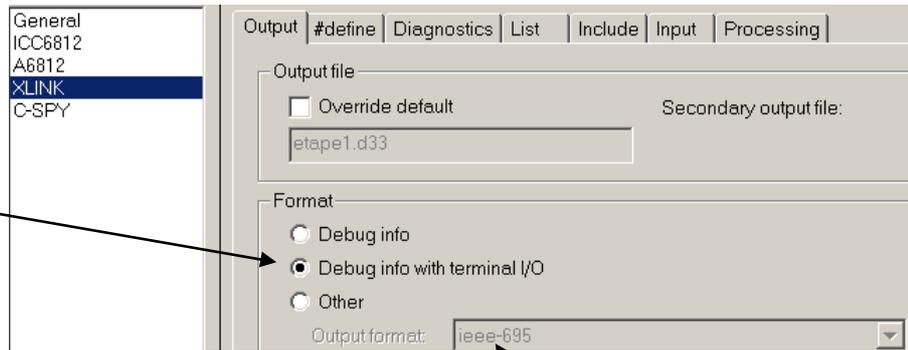
1.3.3.5. XLINK

EDITEUR LIEN

→Output

A la base :

Debug with terminal IO



On choisit ici les formats du code final après édition de lien (on peut aussi choisir des formats pour d'autres outils de téléchargement et de Debug, comme par exemple le classique ieee-695).

EDITEUR LIEN

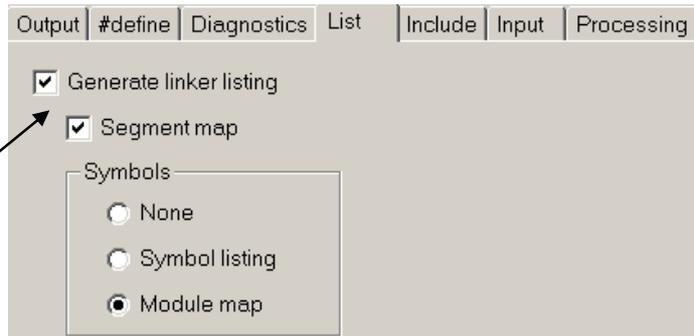
→List

Listing du link:

Génération des fichiers de sortie renseignant sur l'implantation mémoire: fichiers .map

Options très utile:

Generate Linker Listing
Segment map



EDITEUR LIEN

→Include

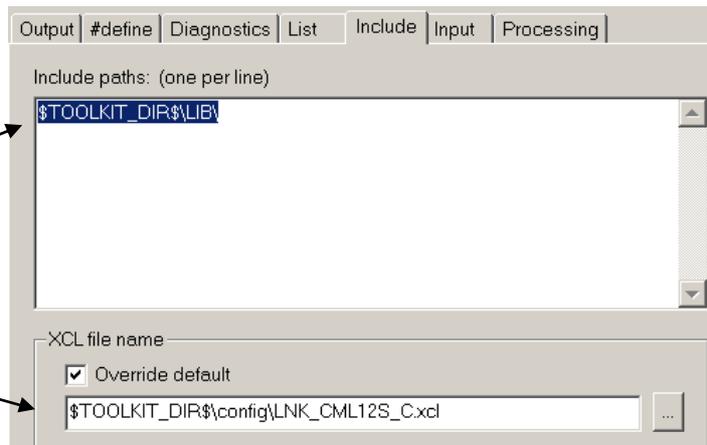
Par défaut !

Chemin des bibliothèques du C:
\$TOOLKIT_DIR\$lib

Chemin du fichier de commande de l'édition de lien:

Ici en fait:

C:\users\hc12\hc12_tools\wbench\6812\config\LNK_CML12S_C.XCL

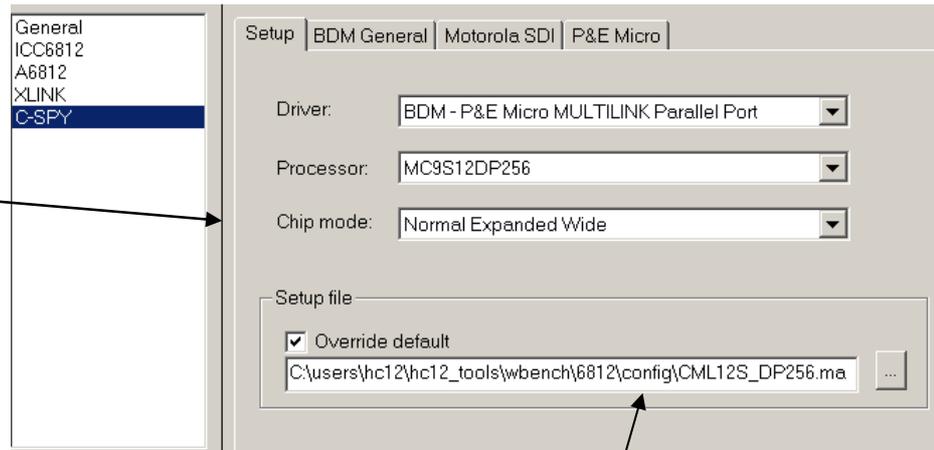


1.3.3.6. C-SPY

DEBUGGER

→Setup

Très spécifique de ce processeur, de cette carte et de IAR, voir doc en plus sur mon site si vous le souhaitez...). **Ne rien modifier !!!**



Driver : BDM P&E Micro Multilink Parallel port

Processeur : MC9S12DP256

Chip Mode : Normal expanded wide (avec bus et mémoires externes).

Set up file:

C:\users\hc12\hc12_tools\wbench\6812\config\CML12S_DP256.MAC

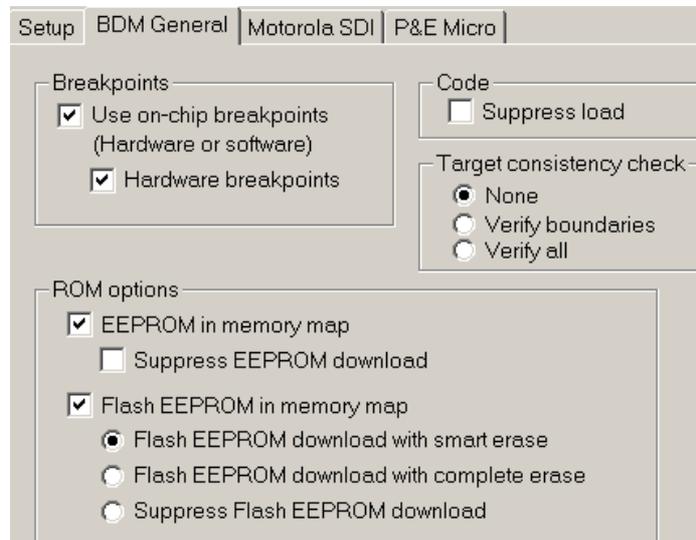
(Fichier de configuration du téléchargement: spécial IAR et ces processeurs, assez compliqué et trop spécifique pour être ici étudié).

C-SPY

→BDM General

On trouve ici des options de points d'arrêt, de choix de FLASH ou EEPROM interne, de chargement...

Ne rien modifier! Spécifique de cette sonde de développement Motorola.



C-SPY

→Mototola SDI

Si on se servait de port série pour le téléchargement. Sans objet en mode BDM.

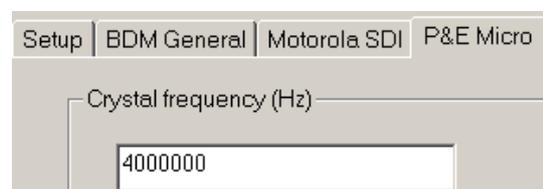


C-SPY

→P&E Micro

Ne rien modifier! Spécifique de cette sonde de développement Motorola.

Entre autre, fréquence du Quarz.



1.3.3.7. Créer un nouveau fichier

Menu : **File New Source_test**

ATTENTION, si on fait **File New fichier**, il faut bien vérifier si l'on sauve correctement ce fichier dans le répertoire de travail, ici : **cours_edtp** !!!!!!!
(Chemin complet à bien vérifier, par exemple **C:\users\hc12\Ele118\tp\cours_edtp**)

1.3.3.8. Ajouter ou supprimer un fichier du projet

Attention : Ce n'est pas parce que le fichier se trouve dans le répertoire courant qu'il est dans le projet !

Menu : **Project Files**

On peut alors ajouter ou enlever des fichiers du projet.

2. PARTIE PRINCIPALE DU PROJET

Programmation en C, moteur, clavier, afficheurs, interruption Travail sans exécutif temps réel

2.1. Introduction

2.1.1. But final

- **Essai de multitâche et de temps réel, mais « sans exécutif temps réel »** Programmation en C sur le microcontrôleur 8 bits HC12 Motorola.
- Point de départ : application simple : faire tourner un **moteur pas à pas** à une **vitesse exacte**.
- **Mise en mémoire EEPROM ou FLASH de l'application**, afin que la carte démarre sur celle ci de manière autonome.
- **Affichage de la vitesse** sur un **afficheur à cristaux liquides**.
- **Modification de celle-ci par clavier**.
- **Affichage simultanée de la tension** en volts fournie par un **potentiomètre**.
- Mise en FLASH de l'application complète si il reste du temps .. ou étude d'une fonction en assembleur....
- En parallèle : aperçu des différents options du compilateur.

Un certain nombre de périphériques internes à l'HC12 sont décrits par ailleurs dans le cours, on en retrouvera un résumé dans ce poly de tp.

2.1.2. Important !

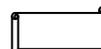
- Il **faudra toujours créer les différentes fonctions demandées!** car certaines seront réutilisées dans des étapes ultérieures du TP.
- On travaillera sur les différents **projets** fournis tels que **etape1.prj** avec des fichiers tels que **etape1.c routines_aff.c**
- Ne pas modifier les noms des projets ni des différents fichiers fournis, sinon il faudrait modifier les projet en conséquence ...
- **Ne pas créer de nouveau projet**, les options nécessaires n'y figureraient plus !!
- **On peut toujours laisser ini_carte() ; en début du main()** Il sera nécessaire lorsque vous travaillerez avec le clavier.

RAPPEL

L'icône **Main** permet de stopper toute exécution en cours en Debug.

→**En plus d'un compte rendu de TP classique**, pour bien repérer les **questions demandant obligatoirement une réponse** dans le compte rendu sont indiquées en principe dans la marge gauche par le signe (sorte de parchemin) :

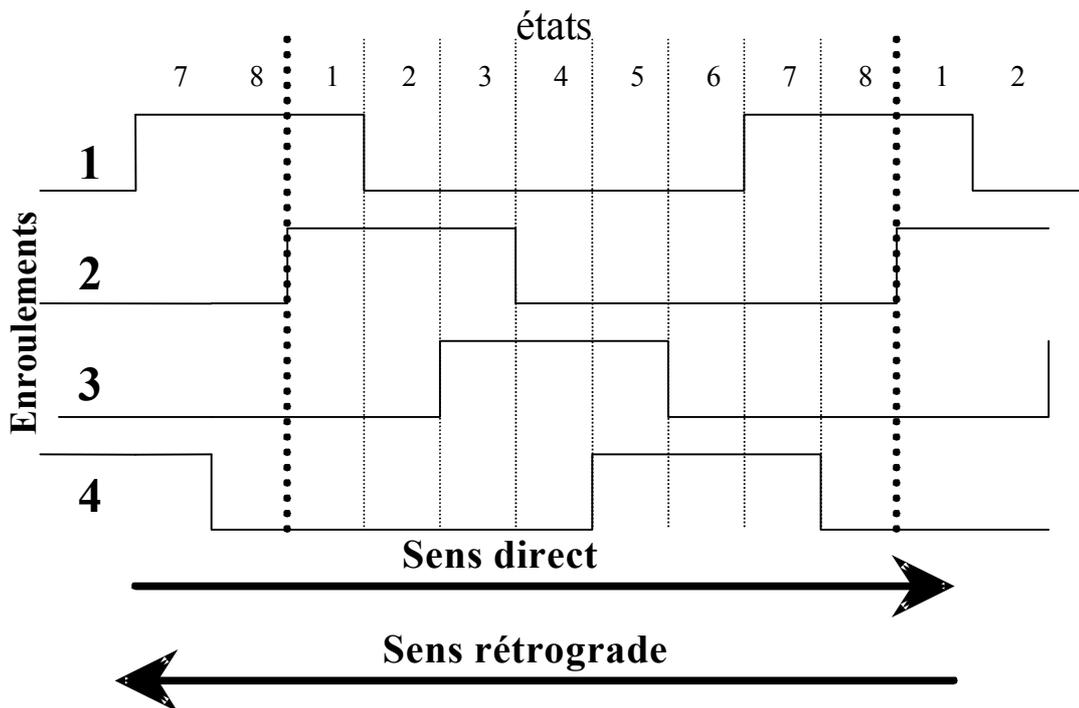
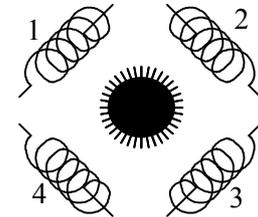
A vérifier tout de même, il peut y avoir des oublis !!!



2.2. Commande du moteur pas à pas sans interruption

2.2.1. Principe d'un moteur pas à pas:

Le rotor comprend "N" crans par tour, en envoyant successivement un courant dans les enroulements 1, 2, 3, 4, on assure la progression pas à pas du moteur dans le sens direct. Pour assurer un bon couple et un démarrage correct, un petit chevauchement est nécessaire. Il faut donc générer des signaux conformément aux chronogrammes ci-dessous:



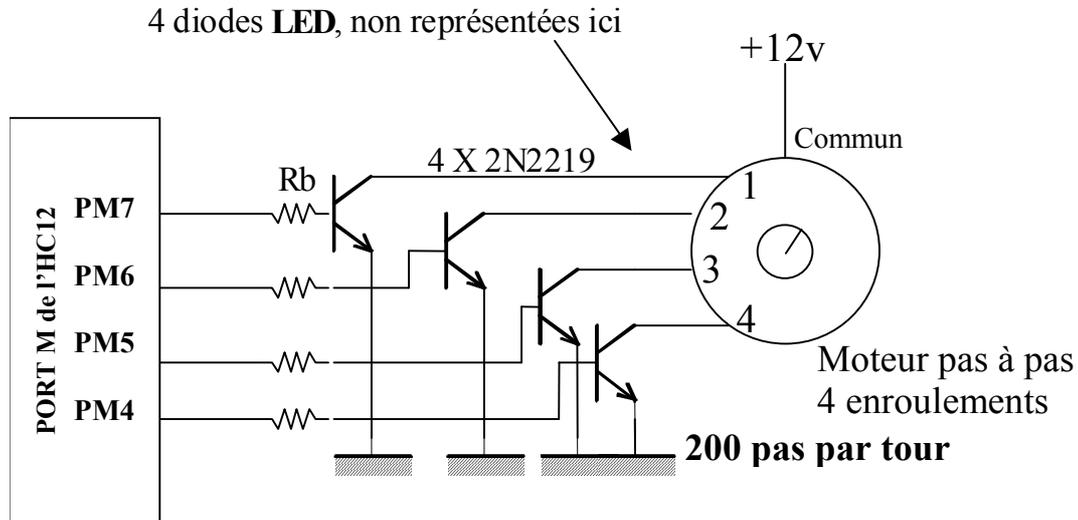
On progresse d'un pas à chaque fois que l'on change d'état. Selon les moteurs, un tour peut comprendre 60, 100 ou plus .

Nous travaillerons sur des moteurs à **N = 200 pas par tour.**

Les utilisations de ces moteurs sont nombreuses : imprimantes, table traçantes, machines outils A partir d'une position initiale, et sauf dé-synchronisation complète, on sait toujours exactement la position du moteur et de tous les organes entraînés par lui, en tenant à jour dans un registre le nombre de pas total directs - rétrogrades. Cela évite la "boucle de retour" des systèmes asservis nécessitant des capteurs de position. Une initialisation "mécanique" est toutefois nécessaire.

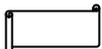
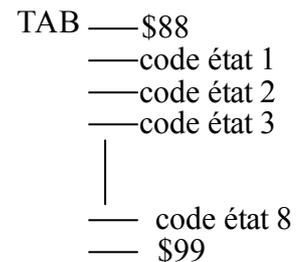
2.2.2. Commande du moteur, tableau de codes

Le moteur est câblé sur un port parallèle de l'HC12, le **PORT M**.



Il faut envoyer, **à une cadence régulière**, sur les 4 lignes PM7, PM6, PM5, PM4 (les 4 bits MSB du PORT M), des codes successifs assurant la **progression des états** 1,2,3,4,5,6,7,8,1,2,3,4.....

Pour pouvoir facilement modifier le chronogramme des signaux envoyés au moteur, il est commode de placer ces codes dans un tableau **TAB**, qui se présentera suivant la forme ci contre. Les codes \$88 et \$99 serviront de "**butée**" pour que le programme fournisse l'état 1 après l'état 8 en sens normal, et l'inverse en mode rétrograde. On peut ainsi facilement ajouter des codes au tableau pour d'autres moteurs, utiliser même des tableaux plus complexes pour assurer des progressions particulières sans modifier le programme.

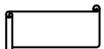


- **Ecrire les codes successifs** (en hexadécimal) correspondant aux différents états.

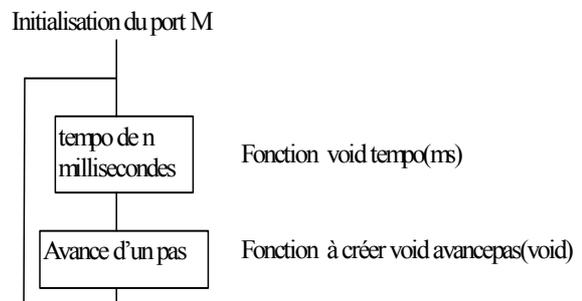
2.2.3. Programmation

Ouvrir le projet **etape1.prj** . On travaillera sur le programme **etape1.c**

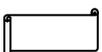
- Organigramme:



Donner en fonction du nombre N de pas par tours, la formule reliant la vitesse **V en tours/s** à la valeur **T de la tempo** (T en secondes ou mieux en millisecondes).



$$V_{tr/s} = f(T_s, N) \quad \text{et} \quad V_{tr/s} = f(T_{ms}, N) ?$$



Application numérique pour 1 Tour/s, trouver la valeur de **T en millisecondes**.

- Description brève des registres de l'HC12 concernant le **port parallèle PTM**

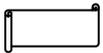
Port M : 8 bits I/O, bus CAN, bus BDLC

Registres classiques de 8 bits:

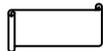
PTM	\$250	Port I/O 8 bits
DDRM	\$252	Registre de Direction
PERM	\$254	<u>Validation</u> de Résistances de <u>Pull up</u> ou Pull down
PPSM	\$255	Si validé dans PERM, Type de pull (0 up, 1 down)

→ On ne se servira que des bits PM7 à PM4 pour le moteur pas à pas.

- Travail à effectuer :
 - En se servant de la fonction fournie de prototype `tempo(unsigned int ms)`; et en **créant** une fonction **`void avancepas(void)`**; Compléter le programme fourni **`etape1.c`** faisant tourner le moteur à environ **1 tours par secondes**.
 - Vérifier à l'oeil le fonctionnement.
 - Mesure de la vitesse :



Ajouter dans la fonction `void avancepas(void)` un signal logique en sortie sur PT5, qui en changeant d'état à chaque pas vous permettra de mesurer la vitesse du moteur. Effectuer votre mesure.



- Expliquez pourquoi, même si la valeur désirée en milli-secondes n'a pas besoin d'être arrondie à un entier (c'est le cas ici), la vitesse de rotation réelle est très légèrement différente de celle théorique, et dans quel sens (plus grande ou plus petite) ? (voir la conception de la fonction `tempo`, et la boucle du programme principal).

Le programme de départ fourni **`etape1.c`**

```
#include <stdio.h>
#include "afficheur.h" // prototypes des fonctions de l'afficheur
#include "ini_carte.h" // où est définie F_BUS à 2 MHz
#include "registres_hc12.h"

extern void ini_carte(void);
void avancepas(void);
char tab_ini_aff[] = {0x28,0x0D,0x01,0x06,00}; /* 00 fin de chaine de config afficheur*/

void main(void)
{
  ini_carte(); // Nécessaire pour les différents « drivers » vers afficheurs et clavier

  /* ?????????? */
}

void avancepas(void)
{
  /* ?????????????????????? */
}
}
```

2.3. Commande du moteur pas à pas par interruption

Enregistrer le programme précédent sous le nom **etape2.c** puis ouvrir le projet **etape2.prj**
 On modifiera **etape2.c** tout en conservant ainsi le précédent programme.

2.3.1. Etude théorique

Le processeur exécute une tâche principale ou une simple boucle d'attente. Des interruptions doivent survenir à une cadence régulière pour faire progresser d'un pas le moteur. Le moteur pourra tourner à une vitesse exacte (précision du Quartz près) et le processeur peut faire autre chose !

2.3.1.1. Cadencement d'une interruption par Timer sur le HC12

Se reporter au poly de cours (seconde partie) pour la description du Timer de l'HC12 et son fonctionnement en mode capture.

$F_{BUS} = 2\text{MHz}$, et le **prédiviseur** du Timer sera programmé à 16.

2.3.1.2. Application au moteur pas à pas.

- On choisira **TC0** pour cadencer l'avance pas à pas du moteur.

Démontrer les formules ci-dessous (on demande de montrer **tout d'abord la formule générale**, puis de faire **ensuite** l'application numérique dans notre cas).

Nbre de pas par tour du moteur	Vitesse en tours/ seconde
N	$V = \frac{F_{BUS}}{\text{prédiviseur} \cdot N \cdot \text{delta}}$
200 Notre moteur	$V = \frac{625}{\text{delta}}$

- Donner alors** la valeur de **delta** pour tourner à environ **1 tour/seconde**.

2.3.1.3. L' « Interrupt Handler »

La plupart des compilateurs C permettent une programmation aisée d'interruptions par cet « Interrupt Handler ». Pour chaque interruption, on écrit une fonction sans paramètres qui devra être lancée lors de l'interruption.

Ecriture dans le cas de notre environnement IAR de travail:

```
interrupt[offset] void fonction(void)
{ ..... }
```

Cette fonction:

- Peut contenir des variables locales.
- Peut appeler d'autres fonctions.
- Mais ne peut pas recevoir de paramètres. Il faut donc les passer par variables globales.

L'**offset** correspond, pour chaque interruption, au **décalage** à partir du début du tableau des vecteurs, soit à partir de **\$FF80** (2 octets par interruption). (Cf annexe).

1) Cas de nombreuses cartes de développement avec moniteur

En mode Debug, sous moniteur, les vecteurs d'interruptions sont programmés dans la ROM moniteur pour renvoyer vers une **table de JMP** en RAM. Le compilateur écrit alors dans cette table automatiquement 3 octets : le code de JMP suivi de l'adresse de la fonction d'interruption désirée.

En mode « *Mise en ROM* » de l'application définitive, la table de JMP en RAM n'existe plus, les vecteurs sont directement écrits dans la table de vecteurs initiale, zone qui sera mise par la suite en ROM.

2) Notre carte HC12 avec développement sous BDM (Background Debug Mode)

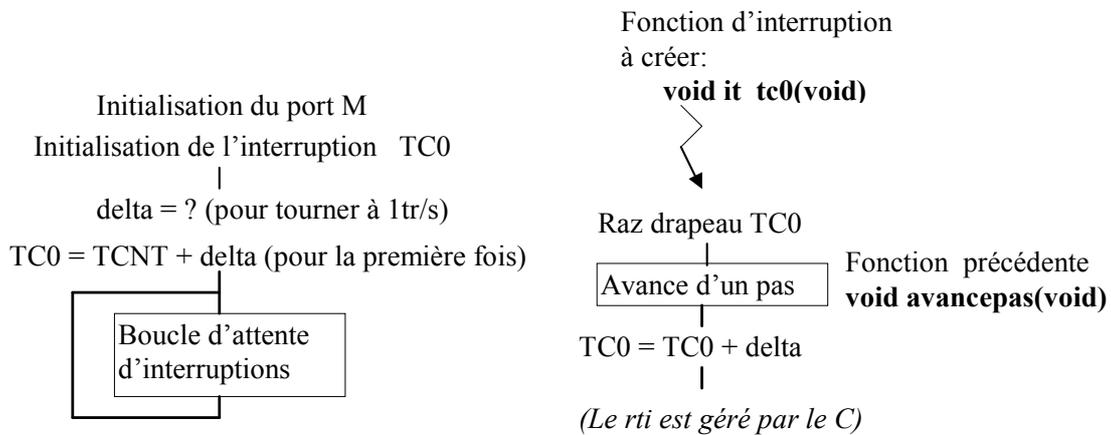
En mode Debug comme pour l'application définitive, **les vecteurs se programment en FLASH à chaque téléchargement.**

Une table de Jump pourrait se gérer en Debug mais est inutile.

2.3.2. Travail pratique

2.3.2.1. Ecriture du programme.

En s'inspirant de l'organigramme ci-dessous, écrire un programme assurant une vitesse de rotation fixe de **1 tours par secondes**.



Remarques:

- **Delta** est évidemment **sur 16 bits, et non signé ! (de 0 à 65535, unsigned int)**
- Si on veut initialiser **delta** dans main(), cette variable doit obligatoirement être une **variable globale**, on ne peut pas en effet passer des paramètres à une fonction d'interruption.
- On utilise le « Handler » d'interruption que nous offre le compilateur C, on pourra nommer la fonction d'interruption correspondante **void it_tc0(void)** et on écrira:

```

interrupt[décalage ?] void it_oc1(void)
{
    /* lignes de programmation de cette fonction */
}
    
```

Le décalage peut s'écrire de plusieurs façons :

- La valeur en hexa ou en décimal
- Une différence Adresse – Adresse basse (plus aisé pour éviter des erreurs)
- Un label défini en #define (plus facile par la suite)

Se reporter à l'annexe.

2.3.2.2. Essai, et mesure de la vitesse

Faire fonctionner.

Le programme doit tourner, régulièrement et à la bonne vitesse.....

Mesurer la vitesse au moyen de la sortie PT5 comme précédemment. Elle doit être ici rigoureusement exacte (à la précision du quartz près).



2.3.2.3. Examen du départ en interruptions : les vecteurs

- Au moyen du Debugger CSPY, et du menu **Window Memory**, lire en mémoire les 2 octets du vecteur d'interruption de TC0, à partir de $\$FF80 + ? = \$?$
- A quelle adresse doit donc se brancher le HC12 lors de l'interruption ?
- Vérifier qu'il n'y a pas de table de Jump, et à quoi le voit-on ? (Rappel code du JMP étendu \$7E).
- Examiner alors le fichier **etape2.map** contenant les adresses des différents labels et sections (l'option fournissant ce fichier est normalement présente dans votre projet. Si ce fichier n'existait pas, il faudrait vérifier dans les options du compilateur, voir avec l'enseignant ..., il est préférable de ne pas tout modifier sans savoir ce que l'on fait ...!)
L'adresse de branchement finale est elle bien celle du programme d'interruption souhaité ?

Remarque : l'adresse **it_tc0** peut aussi se trouver en plaçant en mode C, le curseur sur la ligne et en commutant le **mode assembleur** dans l'outil de mise au point.

2.3.2.4. Examen des différentes options du compilateur IAR

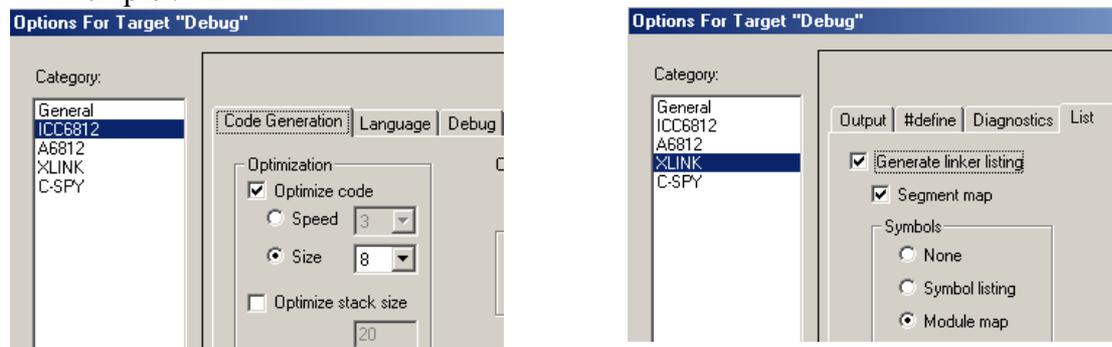
Clicker dans la fenêtre projet de IAR sur Debug (Si Debug n'est pas « en bleu » certaines options ne sont plus visibles...) et sélectionner le menu : **Projet Option**.

On trouve des options réparties en 5 catégories, et pour chacune plusieurs Onglets.

- General**
- ICC6812 (compilateur)**
- I6812 (Assembleur)**
- XLINK (Linker)**
- CSPY (Debugger)**

On rappelle que le 'compilateur' ne fait passer que du C en assembleur, même si par abus de langage on dit souvent compiler pour l'ensemble de la chaîne jusqu'à l'édition de lien.

Exemple :

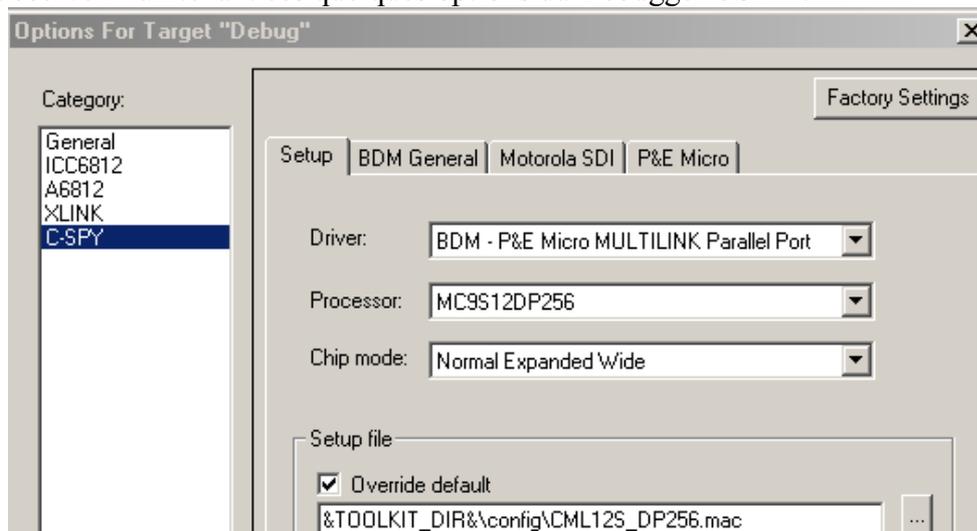


On notera que le nom **%TOOL_KIT DIR%** pour IAR, représente le répertoire où est installé tous les outils logiciels (compilateur, assembleur ...). Sinon on travaille dans un répertoire courant où sont tous les fichiers utiles.

Répondre rapidement avec de petites phrases (ne pas fournir les différentes fenêtres !) et d'une façon assez générale (ne pas relever simplement un nom de chemin sans préciser ce que c'est) à ces quelques questions :

- 1) Option sur le type char du C : as-t-on choisi signé ou non signé ?
- 2) Dans quel répertoire se situent les <fichiers.h> du C (Cette notation signifie ceux fournis pas le constructeur, tels que stdio.h, string.h ...) Et les .h personnel, comme par exemple "registres_hc12.h" ?

- 3) Il y a deux possibilités d'optimisation du code que génère le compilateur, lesquelles ? Donner pour notre cas le type et le niveau.
- 4) On a demandé de générer des Listing de compilation (fichiers.lst) dans une catégorie et Onglet, laquelle, et dire ce que contiennent ces listings.
- 5) A-t-on demandé la génération d'un fichier .map ?, dans quelle catégorie ? Que contient ce fichier.
- 6) Où se trouve indiqué le **chemin des librairies constructeur** du C
- 7) Mais où est alors indiqué le **nom de la librairie** utilisée, et quel est-il ? (revoir dans le cours).
- 8) Où indique-t-on le nom (et le chemin) du fichier de commande d'édition de lien. **donner son nom** et son chemin.
- 9) Décrire brièvement les deux fonctions principales de l'édition de lien pour une application à microcontrôleur.
- 10) D'après les options possibles, décrire brièvement les 3 formats possibles de fichier de sortie final généré par cet éditeur de lien.
- 11) Observer maintenant ces quelques options du Debugger CSPY :



C'est ici que l'on pourrait choisir le Simulateur à la place de la sonde BDM et du Debugger sur le matériel (examiner ces possibilités dans Driver).

Remarques : on trouve ici un autre fichier, situé dans le répertoire Config du compilateur IAR, de nom CML12S_DP256.MAC, c'est un fichier de configuration non pas pour l'éditeur de lien mais pour le téléchargement lui même, il permet à partir d'un processeur type HC12 donné, de choisir différents plans mémoires possibles et options (présence ou non de FLASH, d'EEPROM, modification des adresses des registres internes, ...). Ce fichier étant vraiment trop spécifique de ce microcontrôleur et de ce compilateur ne sera pas étudié. Sur de nombreux autres microcontrôleurs, il n'existe d'ailleurs pas. Le fichier de commande d'édition des lien par contre lui existe dans tous les cas !

2.4. Application définitive

On désire une **application autonome**. Il faut donc placer notre code en mémoire non volatile, ici en FLASH.

2.4.1. Résumé du travail à effectuer pour passer du mode Mise au point à l'application finale :

1) Cas d'un microcontrôleur et d'une carte classique avec Moniteur.

On peut récupérer la place mémoire qu'occupe le moniteur (partie code en mémoire morte et partie variable), le moniteur devenant alors inutile.

Avec les outils modernes de programmation en C, il suffit :

- De modifier le fichier d'édition de lien pour placer tout le code en mémoire Morte, seules les variables restant en mémoire vive.
- De modifier le type de fichier de sortie de l'éditeur de lien pour s'adapter à la carte ou à un programmeur de PROM externe (par exemple fichier Motorola S1S9 : binaire codé ASCII avec informations sur les adresses de chargement).
- On peut supprimer (si ce n'est pas automatique) les options d'information de Debug (Au compilateur et à l'assembleur). Cela peut diminuer la taille du code objet d'une façon significative.

A priori l'éditeur de lien remplace alors automatiquement la table de jump des interruptions par une programmation directe des vecteurs. (A vérifier si c'est le cas tout de même !)

2) Dans notre cas, sans moniteur, et avec mémoire FLASH interne au microcontrôleur

Le **a)** suffit et si besoin est pour gagner de la place le **c)**

Remarque : la table des vecteurs est modifiée automatiquement (cas de IAR). On rappelle que la table de jump était sans objet même en mise au point.

2.4.2. Examen plus détaillé du fichier de commande d'édition de lien : LNK_CML12S_C.XCL

2.4.2.1. Descriptions des différents « segments »

Code et **variables** sont réparties suivant différents **segments**. Nous décrivons ici les principaux :

	<i>ROM</i>	<i>Segments</i>
High		Table des vecteurs d'interruptions (2 octets par interruption) (de \$FF80 à \$FFFF)
	INTVEC	
	CONST	Données déclarées const
	CCSTR et CSTR	<u>Valeurs initiales</u> des Chaînes de caractères initialisées (et non modifiables)et chaînes constantes.
	CDATA0 et CDATA1	<u>Valeurs initiales</u> des variables initialisées dans IDATA0 et IDATA1
	CODE	Code exécutable
	RCODE	Librairies du C, startup du C, autres routines personnelles
low		

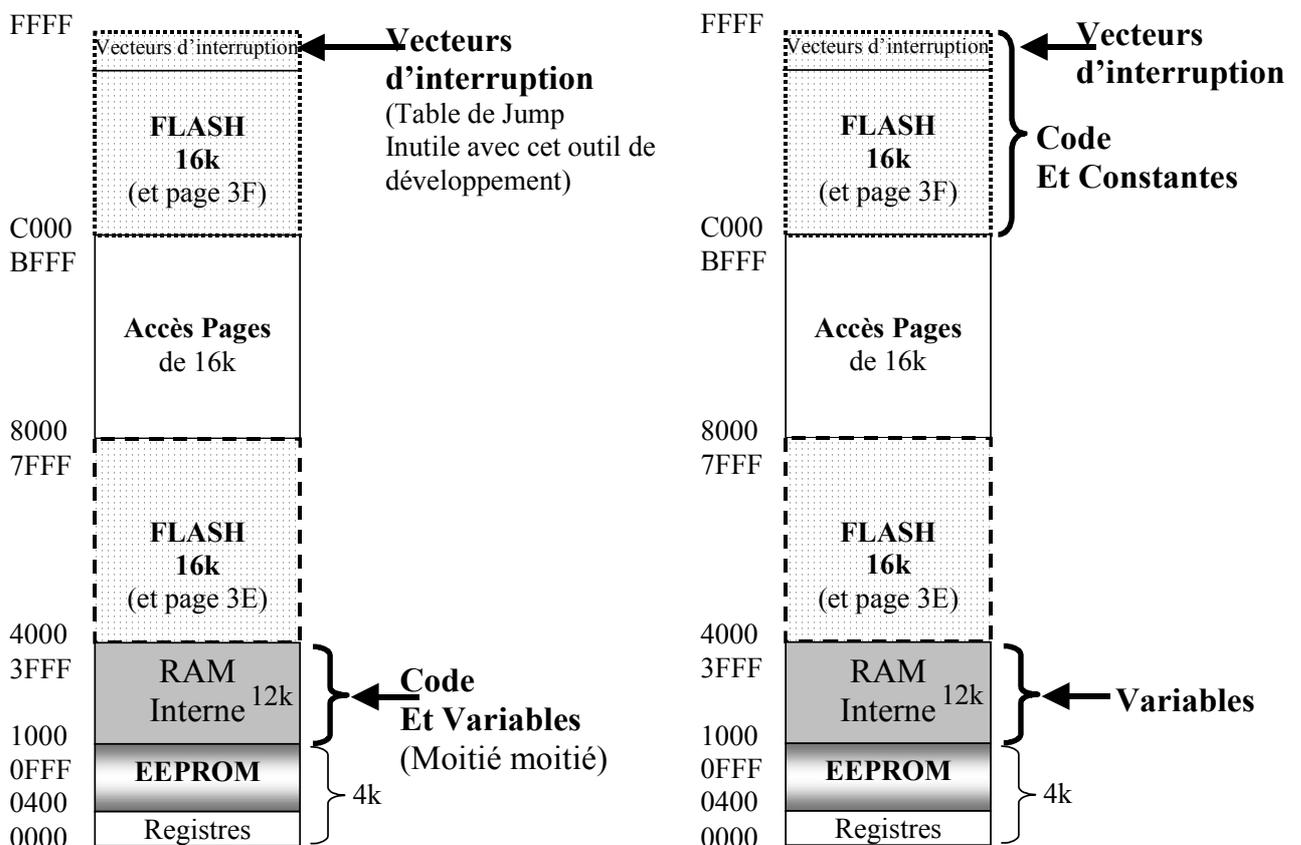
RAM Segments

High ECSTR WCSTR	Chaînes de caractères modifiables
CSTACK	Pile Système, Variables locales et passage de paramètres aux fonctions quand les registres ne suffisent plus (Ce compilateur envoi aux fonctions les premiers paramètres par registres, puis par la pile. Il y a de toute façon empilement du paramètre passé par registre en début de sous programme)
UDATA1 IDATA1	Variables globales et statiques non initialisées Variables globales et statiques initialisées (valeurs dans CDATA0 et CDATA1 des ROM segments)
UDATA0 IDATA0	Idem pour la page 0 seulement (adresses de 0 à 255, accessibles par adressage direct plus rapide)
low	

2.4.2.2. Plan mémoire de la carte HC12

EN MISE AU POINT :

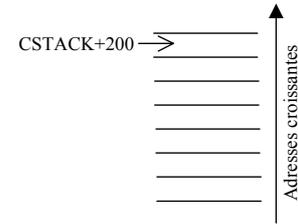
APPLICATION FINALE :



2.4.2.3. Le fichier de commande de l'éditeur de lien

Il se nomme **LNK_CML12S_C.XCL**

Le label **CSTACK + 200** veut dire que l'on réserve 200 octets de pile, l'adresse **CSTACK+200** étant au sommet de celle-ci, tout empilement s'effectuant par auto décrémentation.



Ne pas s'occuper de la distinction **-Z(CODE)** et **-P(CODE)**. Mais ceci permettrait de placer à des endroits différents si nécessaire les sections concernées.

→ Fichier actuel en vue du mode Debug (Code en RAM) :

```
// Type de micro
-c6812
//
//          Sections de type CODE
-Z(CODE)CDATA0,CDATA1,CCSTR=1000-2FFF
-P(CODE)RCODE,CODE,CONST,CSTR,CHECKSUM=1000-2FFF

// The interrupt vectors are assumed to start at 0xFF80
-Z(CODE)INTVEC=FF80-FFFF // En flash

// Sections de type DATA : Toujours en RAM pile de 512 octets
-Z(DATA)DATA1,IDATA1,UDATA1,ECSTR,WCSTR,TEMP,CSTACK+200=3000-3FFF

// Non utilisé en page 0 sont les registres de l'HC11 !
-Z(DATA)DATA0,IDATA0,UDATA0=FFFFFFFF-FFFFFFFF
// La librairie C fournie par le constructeur (contient au minimum ici le Start up du C)
// Le CHEMIN se trouve dans une option du XLINK.
cl6812
```

→ **Important** : le nom de la librairie C choisi (ici **cl6812**). Le chemin étant indiqué dans une option du XLINK.

On demande de remplir le tableau suivant :

Eléments :	Nom de section ?	Application définitive	
		En RAM ?	En mémoire MORTE ?
Code exécutable			
Variables statiques et globales			
Variables locales			
pile			
Valeurs initiales des variables globales initialisées			

2.4.3. Travail à effectuer

- Modifier ce fichier pour s'adapter à la carte pour l'application finale (Voir plan mémoire précédent).
- ReFaire Buid all
- Télécharger (Debugger). Refermer le Debugger.
- Vérifier le bon fonctionnement : démarrage automatique de l'application à la mise sous tension. dès la mise sous tension. (Le bouton Reset si accessible doit agir de même).

2.5. Affichage de la vitesse en tours/secondes

Reprendre le projet **etape2.prj** . On pourra alors **compléter** dans ce projet le fichier **etape2.c**

Attention aussi de bien récupérer le fichier de commande initial d'édition de lien, permettant un travail en Debug et mise en RAM du code (Sinon pour ce processeur et en mode BDM, on peut tout de même mettre au point en chargeant en permanence le code en FLASH, mais les téléchargement sont plus longs....). **Un bouton existe** sur l'interface Visual Basic.

On se sert des fonctions disponibles (prototypes définis dans afficheur.h) gérant l'afficheur à cristaux liquides, à savoir :

extern void config_aff(char *);	Tableau de valeurs à envoyer, pour cet afficheur, en mode 4 bits, deux lignes de 16 caractères: 0x28,0x0D,0x01,0x06,00 (00 = fin de chaîne)
extern void outchaine(char *);	Affichage d'une chaîne de caractères
extern void outinteger(int);	Affichage décimal d'un int (entier signé 16 bits)
extern void outhex16(int);	Affichage hexadécimal d'un int ou unsigned int
extern void change_ligne(void);	Changement de ligne
extern void efface_ligne(void);	Effacement de ligne
extern void curseur(char);	Positionnement du curseur (de 0 à 15 valide)
extern void virgule(void);	Affichage d'une virgule
extern void espace(void);	Affichage d'un espace
extern void putch(char);	Affichage d'un caractère
extern void tempo(unsigned int ms);	Tempo en millisecondes

Elles travaillent toutes sur la **ligne courante**, au niveau du **curseur courant**.

Changement de ligne par **change_ligne()** ;

Effacement et remise du curseur au début par **Efface ligne()**.

Il nous manque une fonction permettant d'afficher aisément des nombres avec partie fractionnaire.

2.5.1. Fonction d’affichage d’un nombre en Q16(32) virgule fixe signé

Le format Q16(32) permet de manipuler des nombres de 16 bits de partie entière (type int) suivi de 16 bits de partie fractionnaires.

2.5.1.1. Principe (déjà vu en cours ...)

Soit un nombre signé N en Q16(32) : ----- , -----

On prend la valeur absolue. On affiche le signe moins si c’est la cas.

On prend la partie entière (16 bits MSB) dans la variable **entiere** (unsigned int)

On l’affiche suivie d’une virgule.

On récupère dans **frac** (unsigned int) la partie fractionnaire en Q0(16).

On multiplie celle-ci 3 fois par 10, et à chaque fois on affiche la partie entière, et on l’enlève. On obtient ainsi successivement les dixièmes, centièmes et millièmes.

2.5.1.2. Examen de la fonction fournie :

Cette fonction vous est fournie dans le fichier **virgule_fixe.c**

Remarque : la fonction permet en outre d’afficher une chaîne de caractères juste après la valeur numérique (volts, ou tours/mn, degrés...)

Il faudra ajouter ce fichier au projet (si il ne l’est pas déjà) au niveau du menu : **Project Files ..** de l’interface IAR.

```
#include "afficheur.h"

void affich_q1632(long q1632, char *unite)
{
    unsigned int entiere; unsigned int frac; unsigned long int val32;
    int k;
    efface_ligne();
    if (q1632 < 0) { q1632 = -q1632; putch(' '); }
    /* remarque : q1632 non modifiée à l’extérieur de la fonction, car passage par valeur */
    entiere=q1632 >>16; /* ici la partie entière est donc toujours > 0 */
    outinteger(entiere);virgule(); /* affichage de la partie entière suivie de la virgule */

    frac = 0xffff & q1632; /* partie fractionnaire dans frac */
    for(k=0;k<=2;k++) /* on multiplie 3 fois par 10 */
    {
        val32 =(unsigned long)10*(unsigned long)frac; /* val32, variable de travail de 32 bits */
        entiere = val32 >> 16 ; /* chaque chiffre (dixième, centième ..., se trouve dans la partie entière */
        putch(entiere+0x30); /* on affiche le caractère ASCII de chaque valeur de 0 à 9 */
        frac = (unsigned int) (0xffff & val32); /* en enlève à chaque fois la partie entière */
    }
    outchaine(unite) ;
}
```

- Faire un petit dessin expliquant pourquoi on récupère dixièmes, centièmes, millièmes dans la partie haute après multiplication.
- Pourquoi faut-il écrire :

$$val32 = (unsigned long)10*(unsigned long)frac$$
 et non $val32 = 10*frac$?
- Aurait-on pu chercher aisément les dix millièmes, les cent millièmes etc Comparer les précisions 16 bits fractionnaires et millièmes. Conclure.

2.5.2. Affichage de la vitesse du moteur

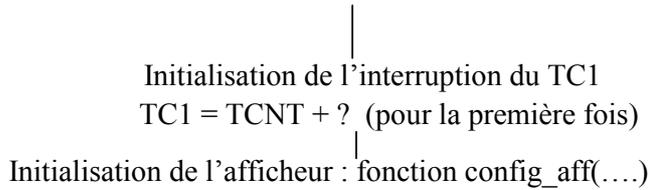
Pour ne pas trop utiliser de temps processeur, on décide d'afficher périodiquement la vitesse (avec l'unité **tours/s**) à des intervalles réguliers cadencés par le Timer, interruption tc1.

On écrira donc une deuxième fonction d'interruption nommée par exemple it_tc1 :

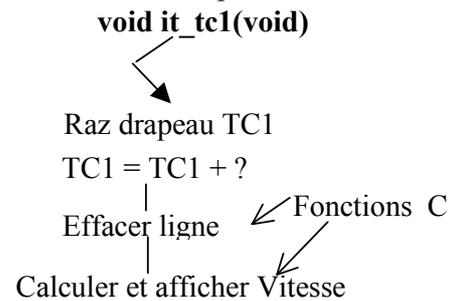
```
interrupt[?] void it_tc1(void)
{ .....
}
```

→ On programmera la cadence la plus lente possible d'interruption (**que l'on calculera**)
On s'inspirera de l'organigramme suivant :

A ajouter dans le main()



Fonction d'interruption à créer:



On calculera la vitesse par la formule déjà étudiée (cas de notre moteur à 200 pas par tours, et cadence Timer de 125kHz, ou 8µs par pas) :

$$V = \frac{625}{\delta} \text{ tours par secondes}$$

Comme cet affichage est relativement peu fréquent, on ralentira peu le déroulement temps réel en **calculant cette vitesse en virgule flottante** (ce qui est plus simple). On **convertira ensuite en Q16(32)** simplement en multipliant par 65536 et en faisant un **cast en long**. On pourra ensuite utiliser la fonction d'affichage précédente affich_q1632.

- Compléter le programme et le faire fonctionner (**Ne pas oublier** de récupérer le fichier de commande du LINK par défaut, permettant le travail en RAM Debug, et de remettre les options d'informations de Debug dans le compilateur ICC6812 si on les a enlevées).
- Comme on sait qu'on ne prouve rien par un cas particulier ! Essayer une **valeur** de vitesse **légèrement différente** (par exemple 1,1 tours/seconde) car pour notre moteur on arrive à tourner exactement à 1 Tr/s et donc on ne sait pas finalement si la partie fractionnaire s'affiche correctement même si l'afficheur indique 1,000 !
- **Observation:** on constate par moment des perturbations de la vitesse de rotation. Ceci est provoqué par l'interruption TC1 d'affichage. La perturbation peut être forte si la durée d'affichage est supérieure à la durée entre deux interruptions moteur (on peut le vérifier en faisant tourner plus vite le moteur).

- Pourquoi l'interruption d'affichage perturbe-t-elle par moment le rythme des pas moteur ? On expliquera ceci clairement et simplement .

- **Programmer le remède en l'expliquant:** tout au début de l'interruption pour affichage, après avoir remis à zéro son drapeau, démasquer toutes interruptions au niveau du masque général. On permet alors à l'interruption la plus importante (avance d'un pas), d'interrompre un autre programme d'interruption ! (Remarque : d'autres microprocesseurs ayant des interruptions de niveaux différents possèdent déjà cette possibilité).

2.6. Commande de la vitesse par clavier

Sauver **etape2.c** en **etape3.c** et ouvrir maintenant le projet **etape3.prj**
On travaille alors sur **etape3.c**

- On désire après frappe d'une touche commande (exemple D) acquérir la vitesse en Tours/seconde, au moyen d'un clavier.
- On dispose des deux fonctions de gestion clavier du poly de cours :

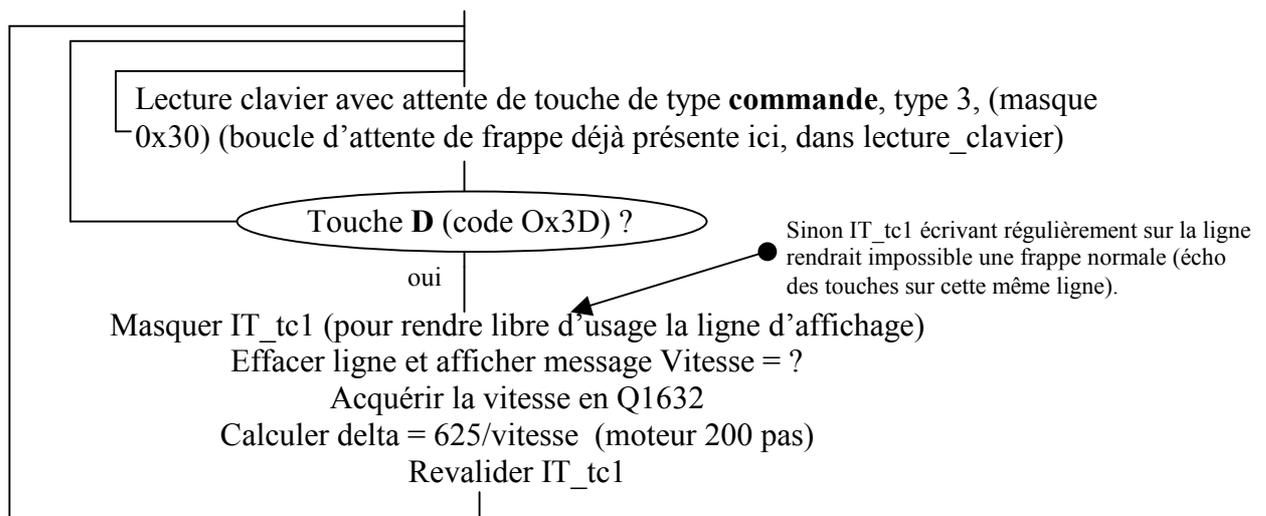
```
Extern char lecture_clavier(char type) ;
Extern char acquisition_q1632(long *) ;
```

Se reporter donc au poly logiciel du cours, comprendre le câblage du clavier sur un port série rapide de l'HC12, et bien voir la différence entre ces deux fonctions, la seconde se servant d'ailleurs de la première.

Selon les messages d'erreurs éventuels, Il faudra éventuellement ajouter dans le projet le fichier contenant les routines clavier, du style routines_clav.c

2.6.1. Boucle de lecture du clavier dans la boucle sans fin du programme principal

Remplacer le `while(1);` du programme principal précédent par :



Remarques :

- Pour masquer et revalider IT_tc1 utiliser les opérateurs logiques `&` et `|`
- Pour changer des virgules flottantes (utile pour gagner de la taille de code), on peut calculer aisément $625/vitesse$ en virgule fixe, il suffit en effet de faire la division des deux nombres en Q1632 et le rapport est directement un entier, et c'est justement ce que l'on cherche pour delta. La vitesse est déjà en Q1632, écrire donc 625 en Q1632.
- Le nouveau **delta** étant une variable **globale**, sera automatiquement pris en compte par l'interruption moteur it_tc1.

Inconvénient : la boucle d'attente ne fait rien d'autre que d'attendre un appui d'une touche de commande, ce qui peut être gênant pour des tâches ultérieures...!!!

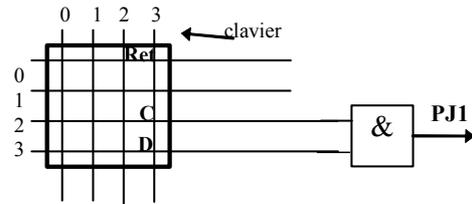
2.6.2. Boucle sans fin de l'application de nouveau libérée : interruption clavier

NE FAIRE CETTE PARTIE QUE SI VOUS AVEZ LE TEMPS
 SINON PASSER A LA SUITE : MESURE D'UNE GRANDEUR PHYSIQUE
 Sauver votre fichier **etape3.c** (qui marche) en **etape3_ok.c** pour le reprendre par la suite si vous n'arrivez pas à faire marcher l'interruption !. **Travailler par contre sur etape3.c**

Avantage : on libère la boucle d'attente principale, qui peut servir alors à autre chose.

On peut utiliser (voir le cours) les touches (C et D) en interruption sur la **ligne PJ1** du PORT J.

Il faudra au départ envoyer un zéro sur la colonne 3 pour activer. Ce 0 sera ensuite au repos toujours présent, en effet la fonction lecture clavier le replace.



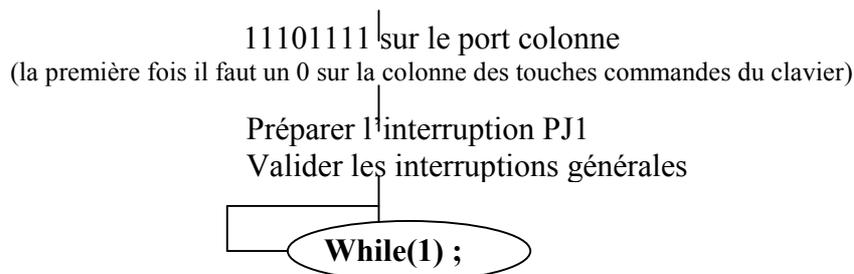
2.6.2.1. Résumé des différents registres du Port J

Voir le poly de cours première partie.

2.6.2.2. Travail à effectuer

On dispose d'une fonction supplémentaire **Extern void ecrire_colonne(char)** ; pour initialiser la première fois le port colonne. (Cette fonction n'est pas qu'une simple écriture à une adresse, puisque le clavier est câblé sur un port série rapide de l'HC12 avec un circuit de conversion série parallèle).

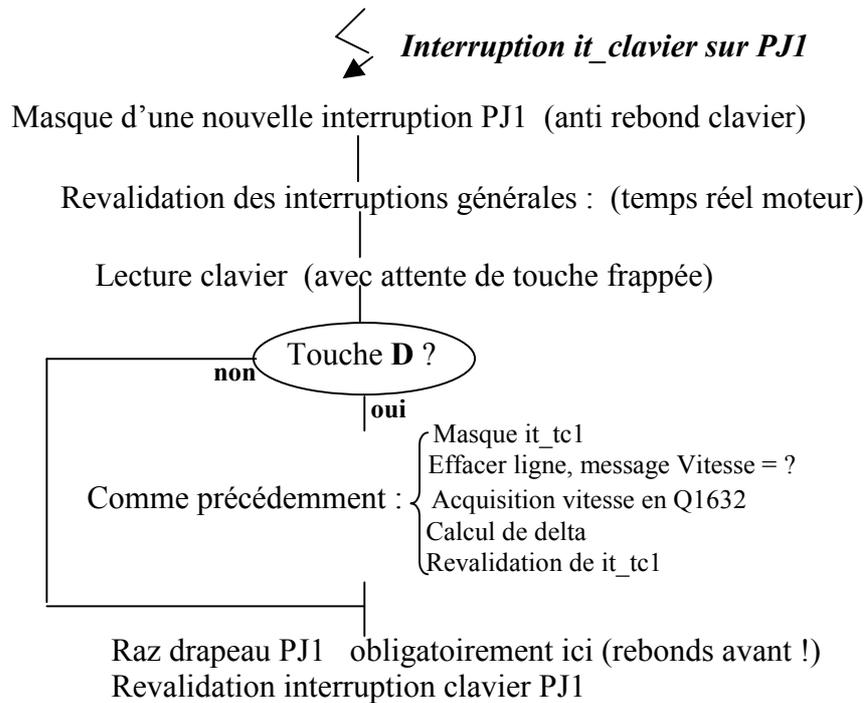
1) Modifier le programme principal comme suit :



2) Ecrire alors la fonction d'interruption PJ1

On la nommera par exemple **interrupt[?] void it_clavier (void)** et on trouvera son décalage dans le tableau des vecteurs comme d'habitude.

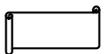
Pour respecter le temps réel sur le moteur (celui ci ne devant pas s'arrêter de tourner durant la frappe, on devra respecter l'organigramme suivant :



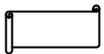
3) Essai de fonctionnement

Vérifier que le moteur n'est pas perturbé par la frappe et que tout marche OK.

4) Questions :



- Expliquer la place du Raz drapeau PJ1 qui a son importance ici !



- Expliquez le rôle de l'initialisation du port colonne à 0xEF une première fois dans le main. Et dire à quel endroit cette initialisation est refaite automatiquement (ce qui est nécessaire).

2.7. Mesure d'une tension

Sauver **etape3.c** en **etape4.c** et ouvrir maintenant le projet **etape4.prj**
 On travaille alors sur **etape4.c**

Toujours à la cadence lente de IT_TC1, on désire visualiser sur la seconde ligne de l'afficheur la tension fournie par un **potentiomètre**, son point milieu étant relié à l'entrée de conversion **AN0** du premier Convertisseur Analogique Numérique de l'HC12 .

Remarque : les fonctions d'affichage travaillant sur la ligne courante, pour afficher sur la seconde ligne on devra donc faire : changement de ligne, effacer ligne, affichage et de nouveau changement de ligne, comme ceci on ne perturbera nullement l'affichage sur la première ligne.

2.7.1. Le convertisseur CAN de l'HCS12

Voir le cours, seconde partie.

2.7.2. Premier essai

Nous utiliserons la voie **voie N°0, (000)**, la ligne d'entrée correspondante étant **AN0**. On demandera la séquence de conversion suivante : **Simple séquence, Mono canal, canal 0**.

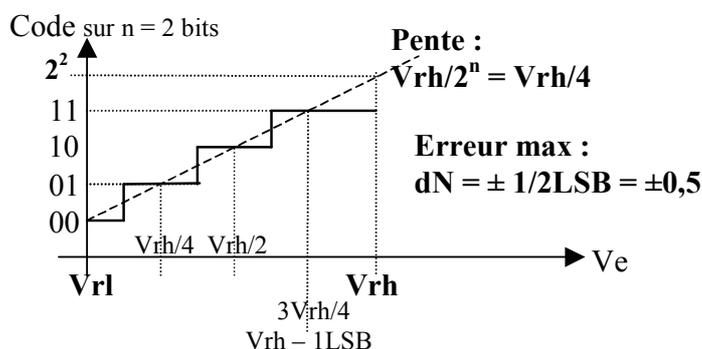
→ On décide de travailler simplement sur **8 bits non signé**, et d'écrire une simple fonction de prototype **unsigned lect_can0(void)** ;

Cahier des charges : Démarrage d'une conversion et récupération d'un code simple entier de 0 à 255 dans un non signé **8 bits**.

- Dans votre programme **etape4.c**, ajouter cette fonction.
 - Tester la fonction en l'appelant dans l'interruption d'affichage it_tc1, et en faisant afficher le code entier (fonction outinteger(int);) sur la seconde ligne de l'afficheur.
- En faisant varier le potentiomètre on doit alors voir des valeurs de 0 à 255.

2.7.3. Partie théorique : passage à la grandeur physique

➤ **Rappel : fonction de transfert d'un CAN classique, exemple sur 4 bits :**



La relation entre la tension et le code est donc $V = k \cdot \text{Code}$

Avec $k = \frac{V_{rh} - V_{rl}}{2^n}$

On rappelle que toute la précision de la mesure provient tout d'abord de la bonne stabilité des tensions de référence, et donc pour réaliser un vrai appareil de mesure précis, V_{rh} et V_{rl} devraient provenir en fait d'un circuit de référence de tension.

Ce n'est pas vraiment le cas ici puisque V_{rh} est directement la tension d'alimentation de l'H12 ! On supposera ici que cette tension est stable, et on prendra la valeur ci-dessous :

→ **On choisit ici : 8 bits non signé, $V_{rl} = 0$ et $V_{rh} = 4,800$ Volt**

➤ **Principe de la mesure et erreur.**

On désire **mesurer avec la meilleure précision possible** la tension **Ve** de 0 à **prsque Vrh**, donc couvrant la dynamique du convertisseur (ce qui évite l'ampli d'adaptation).

La limitation proviendra:

-Au départ du choix d'un **convertisseur (on se limite ici à 8 bits)**, et de la précision de ce dernier (ici **erreur max ± 1/2 lsb**).

-De la limitation à **m bits** des coefficients surtout si on travaille en virgule fixe sur de petits microcontrôleurs.

Comme la tension à mesurer **Ve** varie entre 0 et +5v , on a la relation:

$$V_e = k.N \quad N \text{ étant le code 8 bits lu du CAN}$$

- 1) **Calculer la valeur de k**, écrire **6 à 7 chiffres décimaux après la virgule**, mais il vaut mieux ensuite pour un calcul précis conserver l'écriture sous la forme d'un rapport de deux valeurs flottantes.

On doit trouver **k** voisin de 0,019..... Volt

- 2) Si on ne commet pas d'erreur sur k, $\Delta V_e = k. \Delta N$

En déduire ΔV_e , la **précision absolue théorique (en Volt) de mesure de Ve**. (le CAN donnant une valeur ± 1/2 LSB près, LSB étant le poids faible). On ne pourra jamais dépasser cette précision théorique, sauf en prenant un convertisseur de plus de 8 bits).

2.7.3.1. Calcul en virgule flottante : erreurs de calcul négligeables à priori

➤ **Avantages :**

- Le format **float** du C avec ses 23 bits pour la valeur absolue de la mantisse, donne sur celle ci une erreur max de $\pm 2^{-24} \approx \pm 5,96.10^{-8}$ soit 7 chiffres significatifs. Il est donc amplement suffisant pour coder k.

La précision initiale que permet le convertisseur n'est donc pas dégradée.

- Le calcul de k.N est immédiat, il suffira de convertir ensuite en Q16(32) pour affichage (par produit pas 65536)

➤ **Inconvénient :**

Code machine plus volumineux, et lenteur du calcul (routine virgule flottante du C lancée, sauf pour des processeurs performants qui possèdent des opérateurs virgule flottante câblés). Ce peut être parfois un handicap si on a des contraintes de vitesse.

2.7.3.2. Calcul en virgule fixe

➤ **Avantages :**

- Rapidité d'exécution. Possibilité d'optimiser encore par une programmation directe en assembleur.
- Calcul possible avec un petit microcontrôleur programmé tout en assembleur.

➤ **Inconvénients :**

Programmation un peu plus délicate, et erreurs de calcul à étudier.

On rappelle la formule générale de l'erreur de mesure :
(obtenue très simplement par le calcul différentiel sur Ve)

$\Delta V_{e\max} = k/2 + \text{Codemax. } \Delta k$ <div style="display: flex; justify-content: space-around; margin-top: 10px;"> <div style="text-align: center;"> <p>Précision maximale du convertisseur</p> </div> <div style="text-align: center;"> <p>Erreur supplémentaire de calcul</p> </div> </div>

On décide **ici** de coder **k** avec seulement **8 bits** sans trop d'erreurs si c'est possible.

Ce pourrait être utile si l'on recherchait sur un microprocesseur simple de 8 bits une grande simplicité et rapidité de calcul, évitant ainsi une routine de produit 16bits par 16 bits.

➤ **On code** tout d'abord **sans réfléchir** directement **k en Q8(8)**, soit **k88** (,---- ----)

Calculez l'erreur max de calcul, (on doit trouver environ $\pm 500\text{mV}$!).

En expliquer la cause par les zéros en tête : en calculant k88, donner le **nombre de zéros en tête** non significatifs pour son code binaire.

En déduire l'erreur max de mesure ΔV_e sur V_e (on doit trouver environ $\pm 510\text{mv}$).

Remarque importante : On rappelle que cette **erreur** est pour le **pire des cas**, et non forcément pour notre valeur particulière de k. L'erreur pourrait être nulle si k valait juste un niveau de quantification, soit pour $V_{rh} = 5$ volt, car alors $k = 5/256$, et $k88 = 5/256 * 256 = 5$ exactement !

Le pire des cas serait par exemple pour $V_{rh} = 4,5\text{v}$ (ou $5,5\text{v}$), car on aurait alors :

$k88 = 4,5(\text{ou } 5,5) / 256 * 256 = 4,5$ (ou $5,5$) et donc arrondi à 4 (ou 5), erreur $\pm 0,5$!

Mais ici on cherche à écrire un programme pour un V_{rh} général, avec la garanti d'une erreur maximale !

➤ **Remède : on chasse les zéros en tête !**

Montrer que l'idéal est de calculer V_e en 32ièmes de volts = $32 \text{ k} * N$ (On devra diviser ensuite par 32 pour un affichage en V).

On utilise donc un nouveau **coefficient : K = 32k**.

Ecrire ce nouveau **K** sous forme de rapport.

On nomme **K88** le code de ce nouveau K en non signé Q8(8), donner la ligne de C déclarant cette valeur (avec arrondi au plus près): **unsigned char K88 = ?**

En déduire la nouvelle erreur de calcul sur V_e , (on doit trouver environ $\pm 15\text{mV}$)

En déduire l'erreur max de mesure ΔV_e sur V_e (on doit trouver environ $\pm 25\text{mV}$).

C'est beaucoup mieux ! On décide de se contenter de cette précision.

Pour améliorer, il faudrait alors évidemment envisager un codage de k sur 16 bits (en Q16(16), -----), l'erreur de calcul passerait à $\pm 2\text{mV}$. Et mieux encore, coder $K = 32k$ dans ce même format. L'erreur de calcul serait alors divisée par 32, soit et négligeable alors devant la précision initiale du convertisseur.

2.7.4. Partie pratique

2.7.4.1. Calcul en effectuant le calcul de kN en virgule flottante

Modifier le programme etape4.c pour afficher sur la seconde ligne de l'afficheur à chaque interruption `it_tc1` la valeur de la tension en Volts. On utilisera bien évidemment notre fonction d'affichage `affich_q1632` et on devra donc convertir la tension en signé Q16(32).

2.7.4.2. Calcul en effectuant le calcul de kN en virgule fixe

Le coefficient sera codé **sur 8bits**, et avec l'amélioration codant plutôt **K = 32k**.

On s'aidera d'un petit dessin pour bien voir les opérations à effectuer ainsi que les décalages corrects.

Modifier le programme précédent et vérifier le fonctionnement.

Vous devez avoir un produit en C sur deux opérandes castés sur 16bits. Expliquer pourquoi ce produit se résume ici (si le compilateur optimise bien) à **seulement une seule instruction** assembleur de produit 8 bits par 8bits donne 16 bits (MUL en HC12), ce qui permettrait donc d'effectuer ce calcul même avec un tout petit microcontrôleur PIC très bon marché.

3. TEMPS PARTAGE

Travail sur un petit exécutif : temps partagé et temps réel

On rappelle ici les caractéristiques de notre temps partagé :

Cadence de changement de tâche : $T = 4\text{ms}$

Durée de commutation : $t_c = 0,18\text{ms}$

Lorsque vous ajouterez des lignes aux parties fournies, ne pas les placer n'importe où !. Respectez bien la structure globale de l'ensemble...

Dans tout ce TP on se sert de fonctions spéciales de gestion de clavier et d'affichage cristaux liquides. Ces fonctions sont détaillées dans le chapitre de cours sur le temps partagé. Par rapport aux fonctions standards décrites également dans le poly logiciel, elles sont un peu modifiées pour le multi tâche, pour permettre une écriture sur chaque ligne individuellement par des tâches différentes. Pour l'afficheur, on gère un sémaphore de prise de ressource et des curseurs indépendants. On rappelle ici leur prototype et leur cahier des charges.

➤ Fonctions d'affichage sur panneau cristaux liquides.

Ligne 0 (16 caractères)
Ligne 1 (16 caractères)

- **extern void ecrire_ligne(char ligne, char *texte);**

Ecrire ligne complète ASCII sur ligne 0 ou 1 (max 16 caractères)

- **extern void ecrire_entier(char ligne, int valeur);**

Ecrire en décimal un entier (signé 16 bits) sur ligne 0 ou 1 (le reste des espace)

- **extern void ecrire_q1632(char ligne, long q1632, char *texte);**

Ecrire en décimal un nombre signé en Q16(32) sur ligne 0 ou 1, suivi d'un texte

➤ Fonctions de gestion du clavier

- **extern char lecture_clavier(char type);**

Lire (avec attente de frappe de touche et de type de touche) un code au code au clavier.

(Type : 30hexa pour les touches B, C, D et 00hexa pour les touches numériques)

Retour : codes 0 à 9 pour les touches numériques 0 à 9

0x3B, 0x3C, 0x3D pour les touches B,C et D.

(Tous les codes de touches sont dans le tableau global : tab_touches_clavier[])

- **extern char acquerir_q1632(char ligne, long *q1632);**

Acquérir une valeur décimale signée codée en Q16(32)

Echo des touches sur ligne N° ligne

Retour -1 si dépassement (<32767 ou > 32768)


```

if(vers_pt5==0) { vers_pt5=1; PTT = PTT | 0x20; } // 1/2 periode sur PT5 = 1 pas moteur
else { vers_pt5 = 0; PTT=PTT & (0xFF-0x20); }
PTM=tab[i++]; if(tab[i]==(char)0x99)i=0;
if( (nbrpas++ % 200) == 0){ nbrtours++; ev_tour = 1; } // % donne le reste de la division
}

```

3.1.3. Travail pratique

Attention : Ne jamais oublier les **boucles sans fin dans chaque tache**, sinon les programmes plantent systématiquement !

Le Debug classique (pas à pas) est impossible en temps partagé sans un logiciel spécial non disponible ici. On peut juste mettre un point d'arrêt pour voir une valeur.

3.1.3.1. Essai du programme fourni

- Observer le contenu du projet et les différents éléments. Lancer le programme.
- Calculer le coefficient théorique de ralentissement, pour ce programme.
- Constater et évaluer à l'œil le ralentissement important des taches moteur et chrono, et comparez au coefficient théorique précédent.
- Sur la ligne numéro 0 d'affichage, le nombre de pas augmente sans arrêt, peu lisible.

3.1.3.2. Synchronisation de deux taches entre elles par « événement »

On veut que le nombre de pas affiché ne change qu'à chaque tour moteur. En utilisant la variable globale **ev_tour** modifier le programme. (ev_tour doit être mis à 1 dans avance pas à chaque tours, ce drapeau sera testé et remis à 0 dans la tache d'affichage, on peut se reporter au cours où cet exemple est décrit).

- Contrôler le fonctionnement.
- En observant la valeur changeant une fois par tour, estimer de nouveau rapidement le ralentissement de la tache moteur. Est normal ?

3.1.3.3. Mesure du ralentissement et de la régularité du moteur

On peut facilement envoyer sur un oscillo un signal qui change de niveau à chaque pas. On dispose pour cela de la sortie PT5 du PORT T qui change d'état à chaque pas.

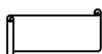
Les lignes de gestion de cette sortie PT5 sont déjà présentes dans avance pas (On peut les consulter).

- Visualiser le signal en sortie de PT5 :
 - Le signal n'est pas régulier, pourquoi ?
 - En réglant la base des temps de l'oscillo pour avoir un motif presque stable (su 3 paliers par exemple. Effectuer une **mesure rapide** (avec un **petit dessin**) permettant d'en déduire l'intervalle de temps moyen entre deux pas (durée moyenne d'un niveau du signal). En déduire le coefficient de ralentissement **k**. Comparer avec la théorie.

3.1.3.4. Amélioration 1 : pour le moteur

Introduire les **priorités** 100 pour le moteur et 90 pour les deux autres taches.

- Constater le résultat sur la vitesse du moteur. Estimer rapidement cette vitesse (en le regardant ou en regardant l'affichage des pas). Le cahier des charges moteur est-il parfaitement respecté ?
- L'affichage des pas reste apparemment synchrone des tours moteurs, expliquer.



- Et le chrono, expliquer ?
- Conclusion sur l'efficacité de cette méthode ?

3.1.3.5. Amélioration 2 : pour le chronomètre

Synchronisation sans interruption, par **Événement Timer**.

On utilise le Timer et le registre de comparaison **TC3**. Si on ne valide pas l'interruption TC3, lors de l'égalité entre le compteur TCNT et TC3, le **drapeau TC3** (dans TFLG1) passe à 1, mais le processeur ne part pas en interruption. On peut donc utiliser le **passage à 1 de ce drapeau** comme événement Timer.

On modifiera donc le processus chrono comme ci-dessous :

```
void chrono(void)
{
    long n=0;
    while(1)
    {
        if((TFLG1 & 0x08) !=0) // drapeau TC3 = événement Timer TC3
        {
            TFLG1 = 0x08; // Raz drapeau événement
            TC3 = TC3 + 100*125; // 125 pour 1 milliseconde DONC 100ms
            ecrire_q1632(1,n," s");
            n=n+65536.0/10.0;
        }
    }
}
```

On laissera les priorités précédentes 100 et 90.

- Constater le bon fonctionnement du Timer.
- Estimer en théorie le retard max de prise en compte de l'événement Timer, pour trois tâches de priorités égales d'une part, et ici pour les priorités 100 (moteur) et 90 pour les deux autres.
- Conclusion : Efficacité et limitation de cette technique. Donner **clairement cette limite** d'une **manière générale**, et **dans ce cas particulier**.
- Peut on l'utiliser pour réguler correctement le moteur pas à pas ?

3.1.3.6. Amélioration 3 : Pour le moteur ?

- On voudrait le **temps réel** pour le contrôle du **moteur pas à pas** (régularité et vitesse exacte), indépendamment de la charge du système par ailleurs.

Etant donné la cadence assez élevée de la progression des pas, une technique sans interruption, par drapeau événement Timer ne fonctionnera pas, nous venons de le voir.

On devra donc rendre la **tâche** moteur **préemptive**, sans passer par l'ordonnanceur, cadencée par une **interruption Timer**, comme c'était le cas finalement dans nos précédents TP.

→ Si vous **n'avez pas le temps de faire le paragraphe suivant**, modifier juste votre programme actuel (utiliser TC0 pour le moteur par exemple), et vérifier le fonctionnement complet (en particulier la vitesse exacte du moteur). Cette partie doit vous être déjà familière, et devrait prendre très peu de temps !

→ **Sinon, passer à la suite**. Elle consiste à développer une application à partir d'un noyau temps partagé, et avec une tâche devant rester sur interruption pour des contraintes sévères de temps réel.

3.2. Développement d'une application

Fermer le projet précédent et ouvrir **multi.prj**. Ouvrir **multi.c**

Ce fichier contient le squelette d'une application multitâche, la fonction **void avance_pas()** ainsi que la fonction d'interruption **void it_moteur(void)** (sur **TC0**) cadencée par le Timer, avec des valeurs permettant de faire tourner régulièrement le moteur à **3.5 Tour/seconde** (moteur à 200 pas par tours).

Une **variable globale flottante** : **float vitesse = 3.5** est déclarée, ainsi que la valeur de delta associée.

La fonction **avancepas()** tourne donc par préemption sans passer par l'ordonnanceur, assurant ainsi une rotation pratiquement régulière.

Aucune tâche n'est créée et la file d'attente est vide. La manipulation consistera à ajouter des tâches petit à petit, qui travailleront elles toutes en temps partagé.

- Observer le fichier fourni, repérer les différents éléments, la tâche préemptive ...
- Vérifier que le programme fourni assure bien la rotation du moteur et rien d'autre.
- Des lignes fournissant à chaque pas un signal sur PT5 sont déjà écrites, identiques à celles que vous ajoutées précédemment. Vérifier à l'oscillo la cadence exacte. Dire pourquoi on n'observe pratiquement pas de fluctuations sur le signal.

3.2.1. Première étape : Modification de la vitesse par clavier

Cahier des charges (sur ligne 0):

La **ligne 0** doit afficher la **vitesse** en tours/s.

Lors d'un appui sur la touche **D**, un message sur la **ligne 0** doit apparaître: **Vitesse ?** invitant à la frappe (**écho** sur **ligne 0**) d'une valeur numérique de la vitesse. On frappe Ret et la vitesse change.

Le clavier peut difficilement être partagé par plusieurs tâches (il n'y a d'ailleurs pas de sémaphore clavier). On imagine mal en effet comment frapper en même temps des valeurs ou commandes différentes, destinées à des tâches différentes ! (à moins peut-être d'être extra terrestre ...). Un processus unique aura donc pour charge toutes les lectures au clavier.

L'idée est de créer une tâche de frappe du clavier et qui positionne seulement des drapeaux d'événement de commande, et une autre tâche qui traite ces commandes (synchronisées sur les événements correspondants). On créera ainsi les deux tâches suivantes :

Tâche **frappe_clavier**

Attente touche commande.

Si touche = D : Message Vitesse ?

Attente frappe valeur vitesse (en Q1632)

Conversion de la vitesse en flottant : $vitesse = ?$

Drapeau Evènement vitesse mis à 1

Tâche **commandes**

Si Evènement vitesse :

Calcul du nouveau delta nécessaire.

Affichage de cette vitesse (avec l'unité Tours/s) sur la **ligne 0**.

Drapeau Evènement vitesse remis à zéro.

Astuce : On déclarera le drapeau Evènement vitesse (**ev_vitesse**) initialisé à **1 au départ**, ainsi dès le lancement du programme le delta nécessaire sera calculé et l'affichage de la vitesse s'affichera sur la ligne 0. (Ca sert les astuces !).

- Créer et faire fonctionner ce programme :
 - Au lancement le moteur doit tourner à 3.5 tours par seconde et cette valeur doit s'afficher sur la ligne 0.
 - Essayer diverses vitesses : 0,5 0,1 2 5,5 ou 6 tours par seconde.
- On veut pouvoir arrêter la moteur. En théorie il faudrait $\Delta = \infty$!
 - Comme la valeur **delta = 0** correspondrait en théorie à une vitesse infinie, on peut se servir pratiquement de ce code pour provoquer une vitesse nulle ! C'est paradoxal mais pratique ...
 - On peut remarquer dans la fonction fournie void it_moteur(void), que avance_pas() n'est appelée que si delta est différent de zéro. Donc ça marchera !
 - Vérifier le fonctionnement en frappant au clavier une vitesse nulle.

3.2.2. Seconde étape : tache messages (sur ligne 1)

On veut obtenir sur la ligne inférieure de l'afficheur (**ligne 1**) un compteur des tours moteur si la vitesse est correcte, un message Arrêt Moteur si la vitesse est nulle ($\Delta = 0$) et des messages « trop lent ou trop vite » si $\Delta > 1000$ ou < 100 respectivement. Peu importe les vitesses exactes, mais pour des moteurs à 200 pas par tours, comme $V = 625/\Delta$, cela correspondrait à 0,625 et 6,25 tours/secondes.

- Créer et faire fonctionner une tache « **messages** » (de priorité 100) affichant **en permanence** ces renseignements. (pas de gestion de drapeau ici).

Remarque : si vous êtes un peu juste en temps, n'affichez que le nombre de tours moteur, et donc ne perdez pas de temps à écrire la série de if, car la suite est importante !

3.2.3. Troisième étape : Suppression et remise de la tache message

Au moyen de la touche **B**, on veut pouvoir enlever et remettre à volonté la tache « messages » précédente (tout d'abord simplement par suppression et insertion dans la file d'attente).

- Compléter les deux taches précédemment écrites :

Tache: **frappe_clavier**

Si la touche frappée est la touche **B** (code 0x3B) faire passer un nouveau drapeau événement à 1, par exemple : **ev_messages_on_off**

Tache: **commandes**

Si ev_messages_on_off est à 1 :

Si la tache est présente, on l'extrait et on affiche une chaîne vide ligne 1, sinon on la replace dans la file d'attente.

On remet ev_messages_on_off à zéro

On utilisera les fonctions fournies :

- char chercher_dans_file(struct tache *tache) ;** Voir si une tache est en file d'attente
- void extraire(id) ;** Enlever une tache de numéro id
- void entrer(struct tache *tache) ;** Mettre en file d'attente

Se souvenir aussi que pour accéder à une donnée contenue dans la fiche d'un processus pointé par exemple par t1, on fera t1->donnée)

- Vérification du fonctionnement : Fréquemment, au moment de la frappe sur B, on peut observer un message ligne 1 de l'afficheur : « **pb semaphore** », indiquant un message de « Time out » au niveau du sémaphore de l'afficheur.

- Expliquer ce problème, et le phénomène de blocage souvent nommé « **Dead lock** » si aucun Time-out n'est programmé.
- Comprendre la gestion d'un sémaphore Semaph_afficheur et d'un Time-out, dans les fonctions de gestion de l'afficheur :

En effet, dans la fonction fournie void `ecrire_ligne(char ligne, char *texte)`, on peut voir les lignes :

```

while((*ch != 0x00) & (k++ <=16))
    {
        attente_liberation(); /* attente afficheur libre de tout process */
        semaph_afficheur = 1; /* Ce process prend cette ressource pour écrire 1
caractère */
        W_IREG(curseur); /* envoi curseur */
        Tempo40micro(); //attente_libre par simple tempo, plus simple ici, et
interromptible
        W_DREG(*ch++); /* envoi caractère pas de tempo */
        semaph_afficheur = 0;
        curseur++;
    }

```

Ainsi que la fonction associée:

```

void attente_liberation(void) // Avec message d'erreur de semaphore sur ligne 1
{
    int time_out = 0;
    while ( (time_out++ <= 5000) & (semaph_afficheur !=0)); // Attente afficheur libre
    if (time_out >=5000){ semaph_afficheur =0; ecrire_ligne(1," Pb Semaphore aff !");
        tempo(250);
        ecrire_ligne(1,"");
    }
}

```

- Programmer alors le remède (Une simple ligne à écrire !), et vérifier le fonctionnement.

3.2.4. Autre manipulation

- En faisant tourner le moteur à 5 ou 6 tours par seconde, observer le signal sortant de PT5 changeant de valeur à chaque pas. Expliquer les petites instabilités.

- As-t-on tout de même le vrai temps réel et un déterminisme sur la tâche moteur quel que soit le nombre de tâches en file d'attente.?

3.2.5. Gestion dynamique des tâches

Dans ce TP nous n'avons fait que retirer et replacer une tâche dans la file d'attente (ce qui correspond à mettre en sommeil ou activer une tâche).

Quand une tâche ne sert plus, il faudrait aussi pour récupérer de la mémoire :

- Détruire dynamiquement sa fiche, On récupère alors toute la place occupée par ses renseignements et par sa pile, donc par ses variables locales.
- Retirer totalement de la mémoire tout le code la concernant (non étudié ici).

Il est facile dynamiquement dans un processus quelconque d'effectuer l'ordre inverse : charger en mémoire le code objet d'une autre tâche, créer sa fiche, et enfin l'activer en la plaçant en file d'attente.

On peut aussi dynamiquement modifier les priorités.

Partie pratique complémentaire si vous avez encore du temps :

Dans ce petit noyau temps réel fourni, on ne peut pas supprimer totalement le code d'une tâche de la mémoire. On peut par contre détruire la fiche correspondante par la fonction fonction **void détruire(struct tache *p) ;**

- Regarder et comprendre la fonction fournie correspondante (en particulier, pourquoi trois instructions free) :

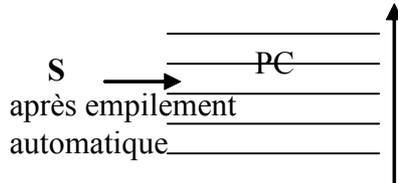
```
void detruire(struct tache *p)
{
    free( (char *) (p->SOMMET_PILE - p->taille_pile + 1));
    free(p->nom);
    free(p);
}
```

- Modifier votre programme dans la tâche commande, et vérifier le bon fonctionnement.

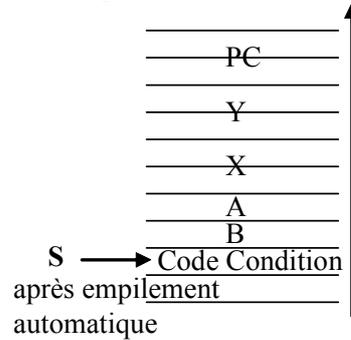
4. ANNEXES SUR LE HC12 : PILE ET VECTEURS D'INTERRUPTIONS

4.1. Evolution automatique du pointeur de pile en HC12

Après JSR :



Après interruption:



Un peu différent de l'HC11 : ici la pile fonctionne par pré-décrémentation.

Le pointeur de pile se place donc à chaque fois **sur le dernier octet empilé**.

4.2. Vecteurs d'interruption HC12

Les vecteurs d'interruptions sont en FLASH, donc reprogrammables.

Avec l'outil de développement IAR, le C programme de la même façon les vecteurs d'interruptions en Mise au point et pour l'Application finale : le **vecteur** contient **l'adresse du programme d'interruption** à lancer.

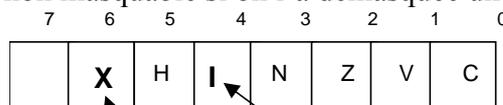
Ne pas s'occuper l'indication HPRIO (qui permettrait de modifier éventuellement l'ordre de prise en compte des interruptions).

➤ *Lignes d'interruptions Hard Ware*

IRQ Sur front

XIRQ Sur niveau

XIRQ est non masquable si on l'a démasquée une fois ! dans le code condition:



masque XIRQ

● **Masque Général des interruptions**

(Par transfert par A : tpa, anda #0%10111111, tap)

➤ *Table de JMP ?*

N'existe pas pour l'HC12, avec la carte utilisée de Axiom, et le Debugger IAR

➤ *Table de vecteurs 1 :*

Vector Address	Interrupt Source	CCR Mask	Local Enable	HPRIO Value to Elevate
\$FFFE, \$FFFF	Reset	None	None	–
\$FFFC, \$FFFD	Clock Monitor fail reset	None	PLLCTL (CME, SCME)	–
\$FFFA, \$FFFB	COP failure reset	None	COP rate select	–
\$FFF8, \$FFF9	Unimplemented instruction trap	None	None	–
\$FFF6, \$FFF7	SWI	None	None	–
\$FFF4, \$FFF5	XIRQ	X-Bit	None	–
\$FFF2, \$FFF3	IRQ	I-Bit	IRQCR (IRQEN)	\$F2
\$FFF0, \$FFF1	Real Time Interrupt	I-Bit	CRGINT (RTIE)	\$F0
\$FFEE, \$FFEF	Enhanced Capture Timer channel 0	I-Bit	TIE (C0I)	\$EE
\$FFEC, \$FFED	Enhanced Capture Timer channel 1	I-Bit	TIE (C1I)	\$EC
\$FFEA, \$FFEB	Enhanced Capture Timer channel 2	I-Bit	TIE (C2I)	\$EA
\$FFE8, \$FFE9	Enhanced Capture Timer channel 3	I-Bit	TIE (C3I)	\$E8
\$FFE6, \$FFE7	Enhanced Capture Timer channel 4	I-Bit	TIE (C4I)	\$E6
\$FFE4, \$FFE5	Enhanced Capture Timer channel 5	I-Bit	TIE (C5I)	\$E4
\$FFE2, \$FFE3	Enhanced Capture Timer channel 6	I-Bit	TIE (C6I)	\$E2
\$FFE0, \$FFE1	Enhanced Capture Timer channel 7	I-Bit	TIE (C7I)	\$E0
\$FFDE, \$FFDF	Enhanced Capture Timer overflow	I-Bit	TSRC2 (TOF)	\$DE
\$FFDC, \$FFDD	Pulse accumulator A overflow	I-Bit	PACTL (PAOVI)	\$DC
\$FFDA, \$FFDB	Pulse accumulator input edge	I-Bit	PACTL (PAI)	\$DA
\$FFD8, \$FFD9	SPI0	I-Bit	SP0CR1 (SPIE, SPTIE)	\$D8
\$FFD6, \$FFD7	SCI0	I-Bit	SC0CR2 (TIE, TCIE, RIE, ILIE)	\$D6
\$FFD4, \$FFD5	SCI1	I-Bit	SC1CR2 (TIE, TCIE, RIE, ILIE)	\$D4
\$FFD2, \$FFD3	ATD0	I-Bit	ATD0CTL2 (ASCIE)	\$D2
\$FFD0, \$FFD1	ATD1	I-Bit	ATD1CTL2 (ASCIE)	\$D0
\$FFCE, \$FFCF	Port J	I-Bit	PTJIF (PTJIE)	\$CE
\$FFCC, \$FFCD	Port H	I-Bit	PTHIF (PTHIE)	\$CC
\$FFCA, \$FFCB	Modulus Down Counter underflow	I-Bit	MCCTL (MCZI)	\$CA

Remarque :

Les vecteurs interruptions « **Enhanced Capture** » sont les mêmes que les interruptions « **Output Compare** ».

➤ *Table de vecteurs 2 :*

\$FFC8, \$FFC9	Pulse Accumulator B Overflow	I-Bit	PBCTL(PBOVI)	\$C8
\$FFC6, \$FFC7	CRG PLL lock	I-Bit	CRGINT(LOCKIE)	\$C6
\$FFC4, \$FFC5	CRG Self Clock Mode	I-Bit	CRGINT(SCMIE)	\$C4
\$FFC2, \$FFC3	BDLC	I-Bit	DLCBCR1(IE)	\$C2
\$FFC0, \$FFC1	IIC Bus	I-Bit	IBCR(IBIE)	\$C0
\$FFBE, \$FFBF	SPI1	I-Bit	SP1CR1(SPIE, SPTIE)	\$BE
\$FFBC, \$FFBD	SPI2	I-Bit	SP2CR1(SPIE, SPTIE)	\$BC
\$FFBA, \$FFBB	EEPROM	I-Bit	EECTL(CCIE, CBEIE)	\$BA
\$FFB8, \$FFB9	FLASH	I-Bit	FCTL(CCIE, CBEIE)	\$B8
\$FFB6, \$FFB7	CAN0 wake-up	I-Bit	CAN0RIER(WUPIE)	\$B6
\$FFB4, \$FFB5	CAN0 errors	I-Bit	CAN0RIER(CSCIE, OVRIE)	\$B4
\$FFB2, \$FFB3	CAN0 receive	I-Bit	CAN0RIER(RXFIE)	\$B2
\$FFB0, \$FFB1	CAN0 transmit	I-Bit	CAN0TIER(TXEIE2-TXEIE0)	\$B0
\$FFAE, \$FFAF	CAN1 wake-up	I-Bit	CAN1RIER(WUPIE)	\$AE
\$FFAC, \$FFAD	CAN1 errors	I-Bit	CAN1RIER(CSCIE, OVRIE)	\$AC
\$FFAA, \$FFAB	CAN1 receive	I-Bit	CAN1RIER(RXFIE)	\$AA
\$FFA8, \$FFA9	CAN1 transmit	I-Bit	CAN1TIER(TXEIE2-TXEIE0)	\$A8
\$FFA6, \$FFA7	CAN2 wake-up	I-Bit	CAN2RIER(WUPIE)	\$A6
\$FFA4, \$FFA5	CAN2 errors	I-Bit	CAN2RIER(CSCIE, OVRIE)	\$A4
\$FFA2, \$FFA3	CAN2 receive	I-Bit	CAN2RIER(RXFIE)	\$A2
\$FFA0, \$FFA1	CAN2 transmit	I-Bit	CAN2TIER(TXEIE2-TXEIE0)	\$A0
\$FF9E, \$FF9F	CAN3 wake-up	I-Bit	CAN3RIER(WUPIE)	\$9E
\$FF9C, \$FF9D	CAN3 errors	I-Bit	CAN3RIER(TXEIE2-TXEIE0)	\$9C
\$FF9A, \$FF9B	CAN3 receive	I-Bit	CAN3RIER(RXFIE)	\$9A
\$FF98, \$FF99	CAN3 transmit	I-Bit	CAN3TIER(TXEIE2-TXEIE0)	\$98
\$FF96, \$FF97	CAN4 wake-up	I-Bit	CAN4RIER(WUPIE)	\$96
\$FF94, \$FF95	CAN4 errors	I-Bit	CAN4RIER(CSCIE, OVRIE)	\$94
\$FF92, \$FF93	CAN4 receive	I-Bit	CAN4RIER(RXFIE)	\$92
\$FF90, \$FF91	CAN4 transmit	I-Bit	CAN4TIER(TXEIE2-TXEIE0)	\$90
\$FF8E, \$FF8F	Port P Interrupt	I-Bit	PTPIF(PTPIE)	\$8E
\$FF8C, \$FF8D	PWM Emergency Shutdown	I-Bit	PWMSDN(PWMIE)	\$8C
\$FF80 to \$FF8B	Reserved			

Remarque

→ La table des vecteurs débute donc à l'adresse : **debut_table_vecteur = \$FF80**