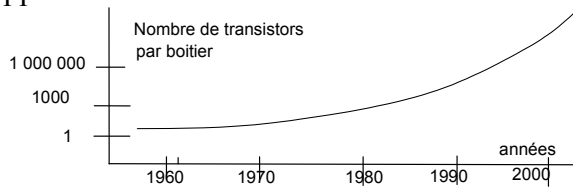
1. INTRODUCTION ET MATERIEL

De nos jours, une grande majorité d'appareils et dans tous les domaines sont commandés pas un système miniaturisé et programmé. Ils sont présent dans tout les domaines, pour ne citer que quelques exemples : l'électroménager, l'automobile, les appareils audio-visuels, les imprimantes, les photocopieuses, la plupart des appareils de mesures….

La figure ci contre montre l'évolution des circuits intégrés LSI (Large Scale Integration) en quelques années.



Les microprocesseurs de plus en plus performants, ont donnés naissance à des circuits complexes, rassemblant en un seul boîtier de nombreuses fonctions.. Les **Microcontrôleurs** se sont ainsi développés, regroupant l'unité centrale, des mémoires, des circuits d'entrées-sorties logiques et analogiques, des circuits de gestion du temps (Timer)…

Ces circuits comprennent une part de plus en plus réduite de « matériel » compensée par une partie importante de logiciel. On parle de micro-systèmes, d'électronique et **d'informatique embarquée**, ou enfouie « embedded ».

1.1. Microprocesseurs, Microcontrôleurs et les autres

Microprocesseurs 8 BITS : Bus données 8 bits. Bus adresse de 12 à 16 bits en général.

Microprocesseurs 16 BITS : Bus donnée de 16 bits, mais aussi taille mémoire adressable nettement plus importante. Nombreuses opérations de base sur 16 bits, et même sur 32 bits.

Microprocesseurs ou processeurs 16, 32 bits ou 64 bits … pour les mini-ordinateurs.

Microcontrôleurs (8 ou 16 bits ou plus) En un seul boîtier, on peut avoir le microprocesseur, des mémoires vives, des mémoires mortes (ROM et pour les très grandes série, sinon FLASH, EEPROM). Ils possèdent aussi des ports d'entrées-sorties parallèles ou séries, des temporisateurs ..., souvent des Convertisseurs AN ou NA …. Types principaux :

Type Motorola	Type Intel
Famille **68HC11** 8 bits (obsolète désormais) Famille **HC12** en 8/16 bits Famille **68330, 68360** en 16 bits (type 68000)	Famille **8051** 8 bits Famille 8051XA (Extended) 16 bits

Les PIC :	Microcontroleurs **RISC**
- Familles PIC 16Cxx - Petits PIC faible consommation, Avec CAN, CNA … etc internes (à bus 4 ou 8 bits)	« **R**educed **I**nstruction **S**et **C**ontroller » donc : nombres d'instructions limités mais surtout : - Séquenceur entièrement câblé. - Structure interne performante pour vitesse maximale d'exécution. (mémoires programme et données séparées …

Circuits spécialisés (System On chip, ou SOC) : spécifiques d'une application et réalisés souvent à la demande.

Processeurs de signaux (DSP) : Ces circuits privilégient les calculs répétitifs (à base de somme de produits) très fréquents en traitement du signal.

FPGA Circuits programmables, pouvant rassembler des circuits logiques aussi bien qu'arithmétiques pour atteindre de très grandes vitesses de traitement.

1.2. La « carte » à Microcontrôleur

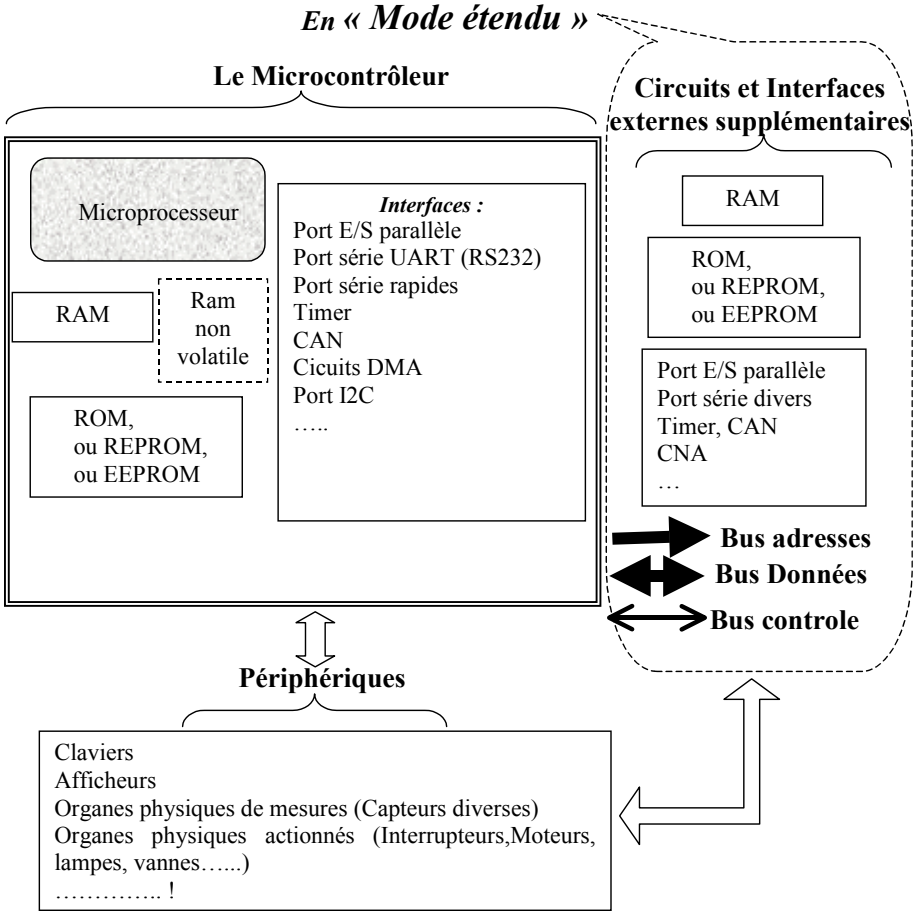
- **Le mode « Single Chip »:** Un microcontrôleur se suffit à lui même dans de nombreuses petites applications (à part des circuits d'affichage, des claviers, des circuits

de puissance pour commander les éléments extérieurs). C'est le **mode d'utilisation le plus courant** aujourd'hui.

- **Le mode « étendu » :** On peut souvent avoir accès aux bus adresse et donnée du microcontrôleur, et ainsi ajouter directement sur ce bus, de la mémoire supplémentaire, des ports d'entrées sorties (nommés ports **IO** ou **I**nput **O**utput), des circuits et interfaces pour développer des applications plus importantes.
- **Mémoire Non volatile** : conservent les données lors de mise hors tension. On distingue :

ROM (**R**ead **O**nly **M**emory) Programmée une seule fois (par le fabriquant). Ce sont en fait les vraies mémoires dites **mémoires mortes**.
REPROM Reprogrammable aisément mais effaçable seulement aux UV. Un peu anciennes !
EEPROM et **FLASH** Reprogrammable aisément, effaçable électriquement, et automatiquement en mode programmation. Très Rapides en lecture, mais algorithmes plus lents nécessaires en écriture. Les FLASH permettent des capacités plus importantes, mais on doit les programmer et les effacer par block.

- **Mémoire Vive RAM** (**R**andom **A**ccess **M**emory). Aussi rapides en lecture qu'en écriture, mais perdent leurs données à la mise hors tension. Des variantes existent (mémoires statiques, dynamiques, a écriture et lecture sur un même, cycle …). Deux types **SRAM** (statique, information stockée dans une bascule à transistor) et **DRAM** (dynamique: information stockée dans une capacité, rafraîchissement obligatoire). La DRAM permet des capacités bien plus importantes. En microcontrôleurs, la SRAM est la plus courante.
  - **Autres Mémoires non volatiles** qui peuvent aussi conserver des données lors de la coupure de courant. De faible capacité, elles conservent leurs données au moyen d'une petite batterie interne ou non. Certaines gèrent date et l'heure.



1.3. Types de structure interne

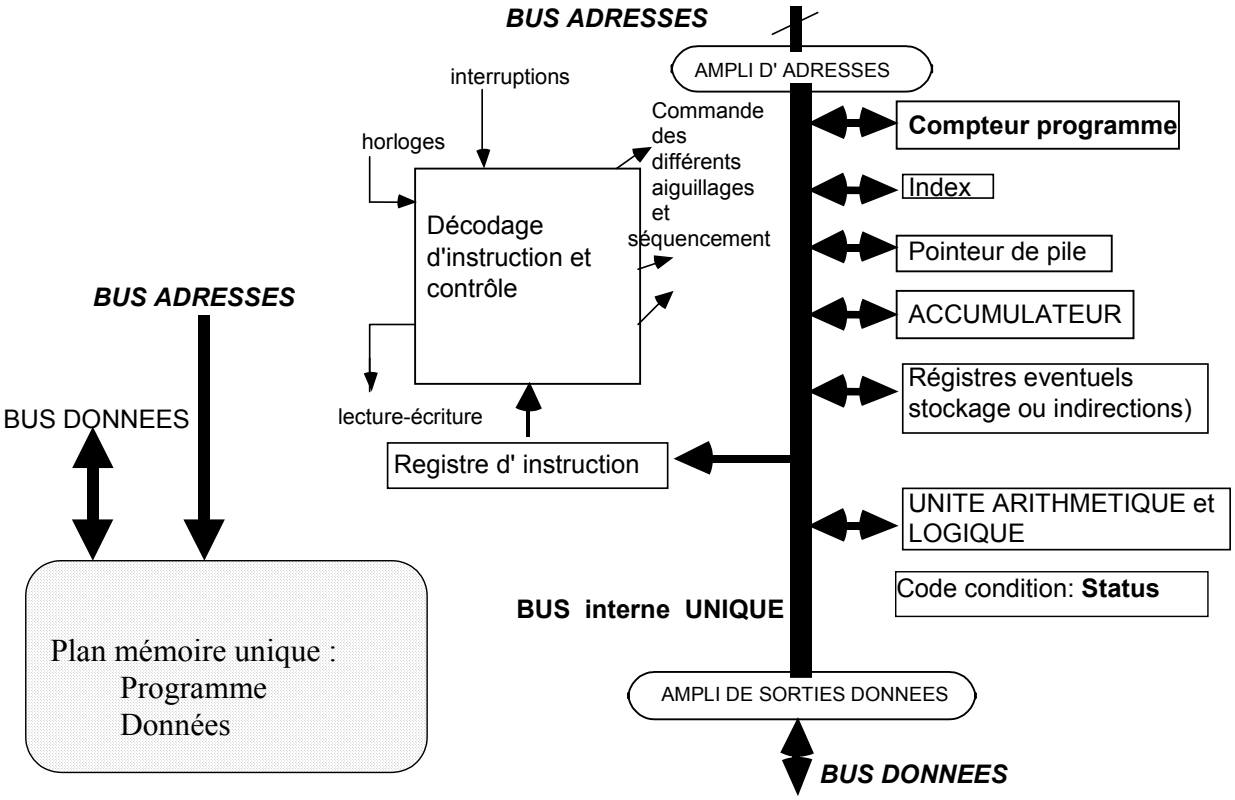
1.3.1. Structure standard, type Von Neumann

Un seul bus interne où transitent adresses et données, et un seul espace mémoire programme/donnée. Les opérations s'effectuent donc nécessairement en un **certain nombre de cycles**. Ils travaillent avec une mémoire regroupant données et programme.

Exemple: pour un adressage étendu et un microprocesseur 8 bits, il faut au minimum les cycles suivants:

Lecture du code opérateur Lecture de l'adresses haute de l'opérande Lecture de l'adresse basse	Recherche
Lecture (par exemple) de l'opérande	Exécution.

En outre que de nombreux microprocesseurs ayant cette structure demandent davantage de cycles: ceci provient de l'unité de contrôle du séquencement qui n'est pas optimisée.



Exemples : Motorola type **HC11,HC12**, type **68xxx**

1.3.2. Structure Harward

Deux bus internes séparés, adresses et données. Deux espaces mémoire distincts: espace **mémoire données** et espace **mémoire programme**.

Pendant la phase d'exécution d'une commande, la recherche de la suivante peut s'effectuer, ceci entraîne un gain de temps considérable, un grand nombre d'**instructions** peuvent alors s'exécuter en **un seul cycle machine** (On parle de Structure **Pipe Line**).

Exemple : type 80C51, microcontroleurs PIC , processeurs de signaux DSP …

1.4. Le mode Single Chip d'un microcontrôleur

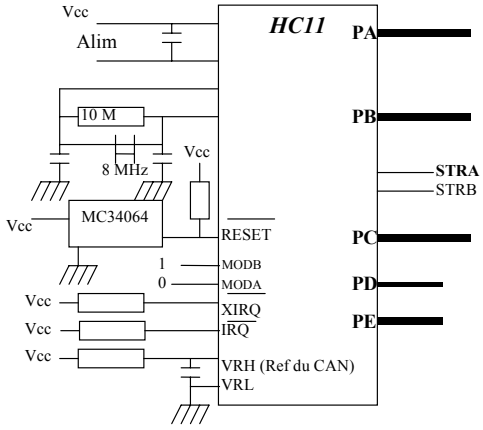
C'est le mode de fonctionnement le plus utilisé actuellement, étant donné le nombre croissant de ports sur ces composants: ports séries, parallèles, lignes d'interruption, mesure de temps, génération de signaux, entrées analogiques, bus série lent et rapides, bus spécialisées.

1.4.1. Mode single chip d'un microcontrôleur

Le câblage est des plus simple, tous les ports du microcontrôleur sont accessibles.

Ici cas du HC11 obsolète, mais le but est de voir ici le câblage toujours très simple !

Ici les signaux MODB et MODA câblés à 1 et 0 assurent le démarrage dans ce mode à la mise sous tension.



1.4.2. Circuits d'IO sur les ports du microcontrôleur.

Les microcontrôleurs modernes possédant un grand nombre de ports, on peut en réserver pour le câblage éléments **d'entrée sortie** (IO) très divers tels que : claviers, afficheurs, capteurs numériques divers (température, humidité, accélération ….). Plusieurs possibilités :

1) Circuits fonctionnant à partir de données parallèles et de signaux de contrôle (ils pourraient se câbler directement sur le bus, à la manière d'une mémoire).

On peut les câbler directement sur un ou plusieurs ports parallèles.

Pour économiser des lignes parallèles, et aussi le câblage, on peut également utiliser un bus série (série rapide) et un circuit de conversion série parallèle.

Les **signaux de gestion** ne sont évidemment pas les signaux du bus (bus donnée, signaux de contrôle), ils doivent être **gérés par logiciel**. Les échanges sont donc forcément ralentis, nécessitant plusieurs instructions, mais cela n'est pas forcément gênant vu la vitesse de travail des microcontrôleurs modernes (F_BUS pouvant aller à 40MHz voir bien plus).

Il faut écrire des **fonctions d'IO** spécifiques, que l'on peut nommer « **driver** ».

2) Circuits gérés par bus spécifique : série rapide, bus philips IIC …, bus CAN…

On les câble évidemment sur ces bus !

3) Circuits gérés par bus série non standard

On les câble sur un ou plusieurs bits d'un port parallèle.

Certains bus parallèles (bus série rapide, bus IIC, bus CAN) permettent même le câblage de plusieurs circuits sur le même bus.

1.5. Le mode étendu d'un microcontrôleur

1.5.1. Rappel du mode étendu

De nombreux microcontrôleurs peuvent travailler dans ce mode.

Un câblage doit être respecté pour démarrer dans ce mode à la mise sous tension. Pour certains microcontrôleurs, des changement de mode sont possible par logiciel.

En acceptant de perdre quelques ports parallèles toutefois, on peut alors disposer du bus complet: adresses, données, signaux de contrôle, comme c'était le cas pour un

microprocesseur, et ainsi ajouter à loisir mémoires, circuits d'entrée sortie…., en respectant évidemment le plan mémoire complet pour éviter les conflits d'adresse !

L'intérêt est d'une part d'ajouter si besoin est de la mémoire, mais aussi de conserver libre encore un certain nombre de ports parallèles et série du microconroleur.
Utile surtout pour les plus gros systèmes.

Remarque :

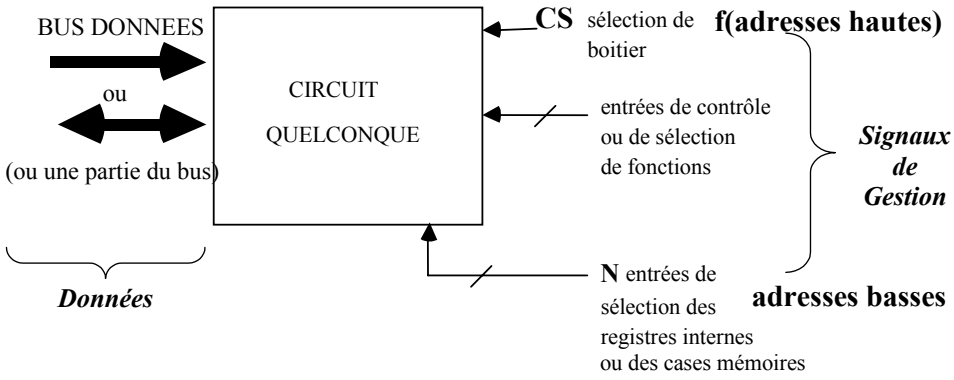
Par soucis d'économie de broches, de nombreux microcontrôleurs ne fournissent pas directement le bus adresse et donnée complet, mais un bus réduit multiplexé. Exemple pour l'HC12, un bus complet demanderait 32 bits (16 bits d'adresses et 16 bits de donnée pour le mode 16 bits), et ferait ainsi perdre 4 ports parallèles ! il fournit en réalité seulement 16 bits multipléxés adresses/données (AD15…AD0). Un registre latch (16 bascules D) permet de prélever les adresses envoyées juste avant les données lors de chaque cycle d'écriture ou de lecture, pour ainsi reconstituer le bus complet. L'HC12 fournit le signal nécessaire pour cela : $\overline{AS}$ (Adress Strobe).

1.5.2. Câblage de circuits d'entrée sortie sur le bus

→ **Cette partie ne sera pas étudiée en détail**, et un document peut être fourni sur demande. Le mode « Single Chip » ou « Microcontroleur » est en effet de plus en plus utilisé, et une étude matérielle plus élaborée dépasserait le cadre de ce cours. On rappellera ici seulement quelques principes généraux.

- ***Tous ces circuits comprennent :***

  Des entrées de **sélection de boîtier** ("chip select" en abrégé CS).
  Des entrées de **contrôle** pour assurer diverses fonctions (écriture ou lecture par exemple).
  Des entrées de **sélection des différents registres** ou **positions** internes.



Ces composants sont ainsi « placés » dans le plan mémoire général du processeur, et on accède à leur registre interne au moyen des instructions classiques du processeur, comme des accès mémoires.

- Il faut donc étudier la gestion de l'espace mémoire, ainsi que les signaux de contrôle (signaux de sélection, signaux de validation, horloges…) assurant les cycles d'écritures et de lecture mémoire d'un microprocesseur.

Pour effectuer une opération quelconque, il faut :

-**PREPARER** le circuit en envoyant des adresses, des commandes, certaines entrées de sélection, des données dans le cas d'une écriture.

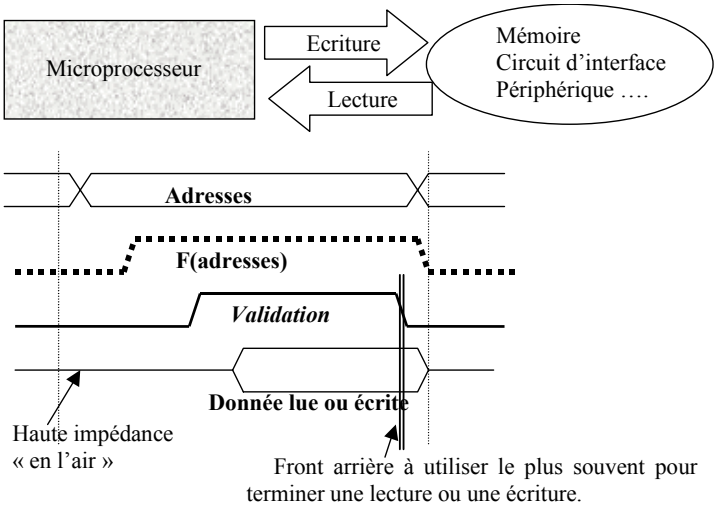
-**Avant de VALIDER** par **un « Signal de validation de l'échange », spécifique au type de processeur, et généré par** celui ci.

Préparation et validation se font à des instants précis du cycle du microprocesseur.

➢ ***Chronogramme général d'écriture ou de lecture***

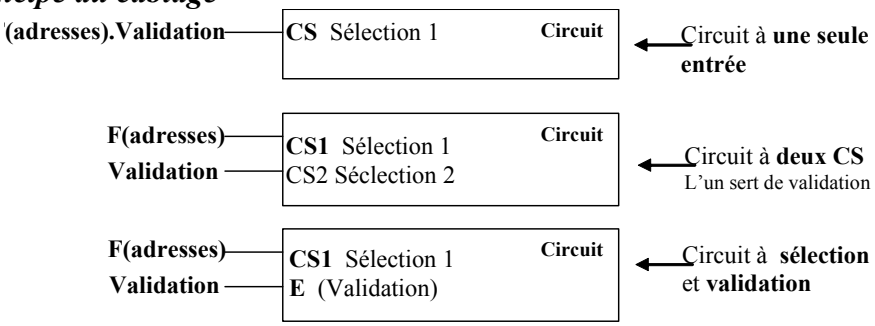
Une fonction d'adresses : **F(adresses) sélectionne** le composant, et permet sa place correcte dans tout le domaine adressable du processeur.

Les signaux F(adresses) et validation pourraient être des signaux actifs au niveau bas.



Remarque : certains processeurs ont un plan mémoire pour les périphériques distinct du plan mémoire de base, et des signaux distincts de validation.

➢ ***Principe du câblage***

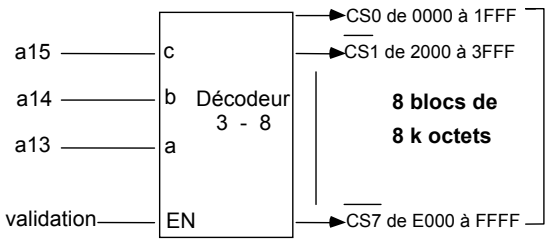


➢ ***Exemple de décodage d'adresse***

Partage d'un domaine de base de 16 bits d'adresse en 8 blocs.

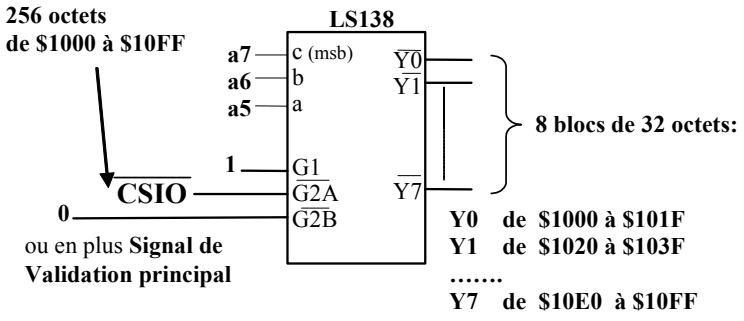
La validation peut se mettre en aval.

(A la place du décodeur, on peut utiliser évidemment un circuit réalisant la fonction logique de décodage).



Les sorties des décodeurs courants sont souvent actives à 0 .

Partage d'un domaine réduit déjà décodé (ici de \$1000 à \$10FF) en 8 sous blocs.



1.6. Problèmes de puissance

Par faute de place dans ce poly, se reporter à l'annexe du poly de TP.

2. INTRODUCTION A LA PROGRAMMATION

2.1. Choix du ou des langages de programmation

Assembleur

- C'est le langage de base, **spécifique** du processeur. En fait, il s'agit directement du code machine du processeur écrit sous une forme tout de même plus aisée à manipuler !

C

- langage évolué très performant permettant une liaison aisé avec l'assembleur, et parfois même une programmation très proche de celui-ci si nécessaire (usage de pointeurs...). **Langage portable** avec pas ou peu de modifications vers d'autres processeurs.

	Assembleur	**Langage évolué (C surtout)**
Avantages	- Permet d'optimiser au maximum un programme en temps d'exécution.	-Grande facilité et souplesse de programmation. -**Très nombreux types d'opérandes** (entiers, nombres en virgule fixe, en virgule flottante, structures, tableaux, chaînes de caractères…) - Grand nombre de **fonctions mathématiques** disponibles. -**Gestion très simple** des **interruptions.** -**Portabilité** vers d'autres processeurs (mis à part les routines travaillant sur les périphériques internes).
Inconvénients	-Spécifique au processeur -Long et pénible à développer -Nécessité de disposer ou d'écrire des sous programmes ou macros pour travailler sur des opérandes au delà de la taille de base ou en virgule flottante.	-Malgré les performances des compilateurs pour microprocesseurs actuels, le code n'est pas toujours optimisé en vitesse de traitement. -Taille du code et temps d'exécution évidemment très augmentés si on travaille systématiquement en virgule flottante sur des microcontroleurs ne possédant pas d'unité arithmétique virgule flottante, et c'est le cas de la plupart d'entre eux.

Conclusion :

Actuellement pour développer une application industrielle sur microcontrôleur :

- On écrit entièrement en **C** les parties de programmes **sans grandes contraintes de vitesses ni contraintes de taille de code** (Souvent 90 à 99.9% du programme !).

- Les parties à **contraintes de vitesse** (pour assurer le "temps réel" (surtout les algorithmes contenant des boucles de calcul), ou certaines fonctions dont on veut **optimiser la taille du code**, **peuvent** être mises sous la forme de **fonctions C écrites** en **assembleur.**

Un programme entièrement en assembleur pourrait être encore un peu plus rapide, mais nécessiterait des efforts de programmation disproportionnés avec le gain de temps encore réalisable....

Centaines **instructions très spécifiques du processeur** restent encore **en assembleur** ou du moins sont placées dans une fonction C écrite en assembleur.(manipulation de certains masques d'interruption par exemple).

Des sous programme déjà existant en assembleur, après quelques modifications parfois très simples, peuvent être réutilisés pour créer une fonction C.

C et assembleur sont des langages très liées, constantes, variables et fonctions pouvant être déclarées et utilisées à partir d'un langage comme de l'autre.

Il faut évidemment posséder la documentation technique du compilateur expliquant comment travaille le compilateur, pour utiliser conjointement les deux langages.

2.2. Programmation structurée

Dans tous langages, afin d'éviter une perte de temps, et pour créer du logiciel réutilisable pour d'autres applications ou par d'autres personnes, la programmation structurée est nécessaire.

1) **Un logiciel complet devra comprendre:**

- Un programme principal
- De nombreux **sous programmes** ou **fonctions**
- Un cahier des charges clair pour chaque fonctions :
  - Les variables en entrée, les variables en sortie
  - Les variables pouvant être détruites (aucune si possible !)
  - Les précautions d'emploi: interruption autorisées ou non, masquées ou non en cours de traitement; l'utilisation de ce programme à tout moment ...
  - Eventuellement l'encombrement mémoire, la durée d'exécution ...

L'idéal est de réaliser des programmes ou sous programmes utilisables sans précautions particulières et à tout moment .

2) ***Très important :*** Si des **valeurs** à priori **figées** sont **susceptibles de changer** en fonction par exemple de la carte utilisée (adresses de ports, constantes diverses, taille d'un tableau, de la pile…), il faut regrouper leur déclaration et leur initialisation **à un seul endroit**, **et dans des fichiers bien précis**, ceci pour faciliter toute modification ultérieure. Sinon c'est la pagaille si on veut reprendre par la suite un programme !

3) ***Minimisation du temps d'exécution***

La multiplication des sous programmes et fonctions ainsi que la sauvegarde et restitution de variables va bien évidemment à l'encontre de la rapidité de traitement. Pour certains cas particulier, un excès de structuration des programmes est nuisible, il faudra dans ce cas par exemple éviter les nombreuses fonctions courtes d'exécution avec transferts de paramètres.

La recherche de rapidité amène souvent à réduire la polyvalence d'un logiciel.

2.3. Chaîne de compilation

2.3.1. Généralités

On trouve sur le marché des petits logiciels pour développer du microprocesseur, comprenant juste un assembleur. Ce sont surtout des produits « école » et de découverte des microprocesseurs, ils sont peu onéreux, mais très réduits en possibilité (souvent travail sur un seul fichier, évidemment pas de programmation, en langage évolué…).

Nous ne considérerons que des **produits** destinés à développer des **applications industrielles, destinées à être implantées sur des cartes à microprocesseur.**

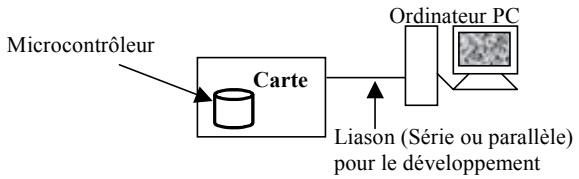
→ **Durant la phase de mise au point**, nous pouvons avoir deux configurations :

N°1) Avec Debugger

Configuration avec Carte et son microcontrôleur, liaison (série, ou parallèle, ou USB) entre un ordinateur de développement et la carte.

Sur le PC, un programme **Debugger** permet la mise au point.

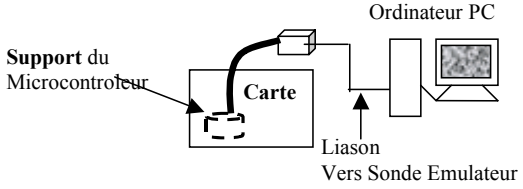
Sur la carte, un programme **Moniteur** (le plus souvent) assure l'interface de la liaison.



Avantage : moins onéreux.

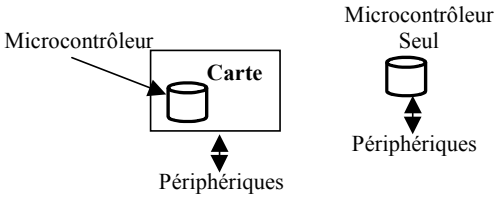
N°2) Avec Emulateur

Configuration avec Emulateur : une sonde spéciale relie l'ordinateur de développement directement au Support du processeur de la carte, et se substitue à lui.



Avantages: mise au point plus aisé en vrai temps reél. Mais plus onéreux !

→Pour l'application finale :
Deux possibilités :

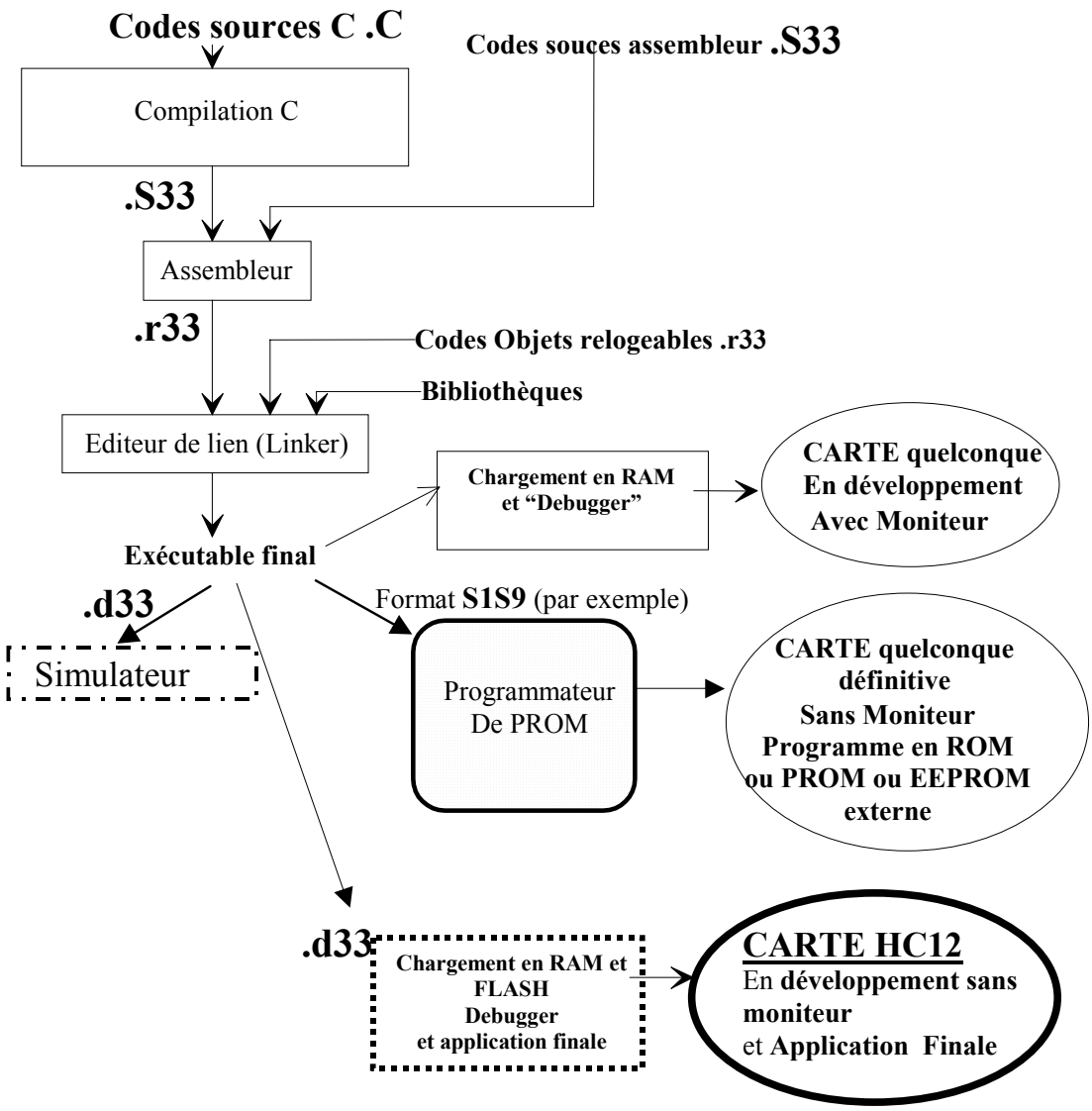


→ On doit donc avoir présent à l'esprit que le code objet définitif résultat de la compilation doit :

- **Tourner entièrement en mémoire vive** (RAM) (Sauf si il appelle des routines déjà en mémoire morte …) pour une application en **développement** (mode Debug) avec téléchargement par exemple du code sur la carte de travail.
- **Tourner en mémoire morte** (ROM ou PROM ou FLASH ou EEPROM) pour **l'application définitive**. Le programme doit aussi démarrer tout seul à la mise sous tension de la carte.

2.3.2. Outil de développement

Un « **outil de développement** » performant comprend les outils suivants (Extensions des fichiers correspondant aux produits de chez **IAR Systems**) :



Note : certains microcontroleurs (comme l'HC12) on une possibilité de mise en point par un port spécial, nommé **BDM** (Back Ground Debug Mode), avec sonde spéciale, le code pouvant même résider en FLASH.

- **Le terme général "Compilation"** est en fait souvent employé pour désigner la succession des tâches à exécuter avant d'arriver au code exécutable final.

La **compilation** proprement dite n'est que la <u>première phase</u> de ce processus, aboutissant à un premier code brut en assembleur.

L'optimiseur réduit le code dans des proportions non négligeables, en anticipant certaines opérations, en éliminant certaines variables intermédiaires en mémoire et les opérations sur les constantes, en optimisant les boucles..., mais avec parfois des résultats inattendus !!!. Des options diverses d'optimisation existent: optimisation **en temps d'exécution** ou **en taille de code** généré.

- **Sections, ou segments**

Dans un code généré par un compilateur, ou un assembleur…, on peut distinguer des parties bien distinctes, les quatre principales étant :

Le **Code**, exécutable par le processeur

Les **Données** (variables, constantes, chaînes de caractères ….

Les **vecteurs d'interruptions ….** Au minimum le Reset !

La pile : Nécessaire pour les sauvegardes automatiques lors des départs en sous programme (compteur programme), lors des départs en interruptions (au minimum le compteur programme et le status). Les autres rôles de la pile seront vus ultérieurement.

Ces parties portent le nom de « **segments »**, ou de « **sections»**

Les sections ont leurs caractéristiques propres : implantation au niveau de la carte à des endroits différents, rôle différent ….

On distingue deux grands types de section :

- Les sections de **type CODE**, destinées à être en **MEMOIRE NON VOLATILE** dans l'application définitive
- Les sections de **type DATA** (donnée), destinées à être toujours en mémoire Vive, **RAM**.

- **Code absolu**

C'est du code destiné à **être placé** et à **ne tourner qu'à une adresse bien précise**.

On ne connaît à priori que ce code dans les outils simple « école ».

– Pour le **code objet intermédiaire** (avant édition de lien), le **manque de souplesse** d'un code absolu est évident pour de plus grosses applications, (on modifie souvent le programme et même le matériel). Des conflits sont possibles entre des zones d'adresses….
– Un **code objet absolu** est par contre **très courant pour le code final** des applications microprocesseurs, car celui ci est destiné à être implanté sur une carte définitive une fois pour toute.

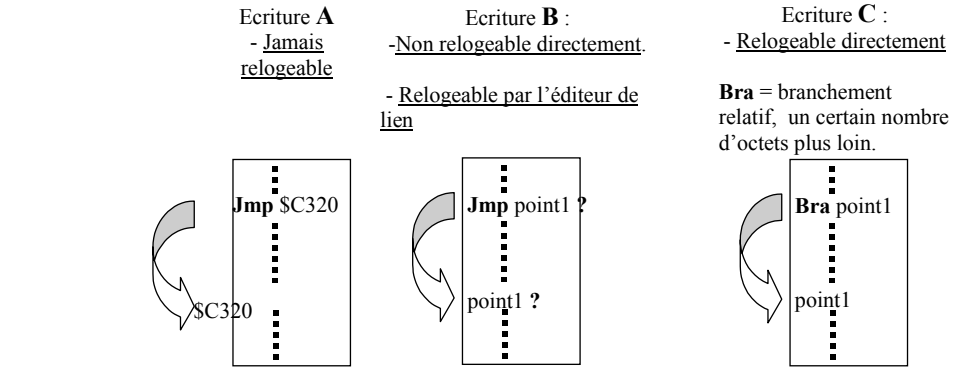
- **Code "Relogeable" ou "translatable"**

C'est du code pouvant être **placé** en mémoire et **s'exécuter à n'importe quelle adresse**.

Il faut distinguer :

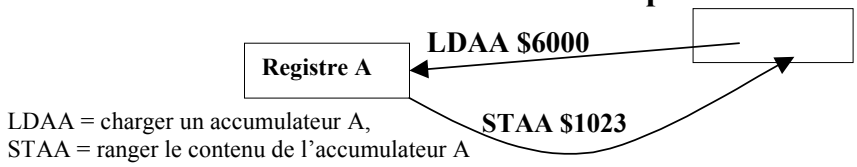
– Le code relogeable après modification de quelques valeurs à l'édition de lien.
– Le code relogeable directement, sans modification.

Il est évident que certaines écritures ou instructions donnent un code dépendant le l'emplacement mémoire : exemples simples avec instructions de saut ou de transfert.



Ecriture **D** : - Directement relogeable pour des opérandes à des adresses fixes.

RAM ou ROM ou registre : Opérande à une adresse fixe



LDAA = charger un accumulateur A,
STAA = ranger le contenu de l'accumulateur A

Ecriture A : l'adresse est écrite en « dur » dans le code, jamais relogeable.

Ecriture B dans une section déclarée absolue : elle fournit à l'assemblage une écriture comme en A, l'éditeur de lien ne pourra évidemment plus rien modifier.

Ecriture B dans une section destinée à être relogée permet à l'assembleur de laisser une sorte de point d'interrogation, **l'édition de lien complétera** le moment venu.

Ecriture C est directement relogeable, le saut doit s'effectuer un certain nombre d'octets plus loin, ce nombre étant toujours le même quelle que soit l'adresse de point1.

Cas D : pour des opérandes (ou des registres de circuits périphériques) à des adresses fixes, le code est directement relogeable avec l'adresse écrite dans le code exécutable, puisqu'elle ne change jamais.

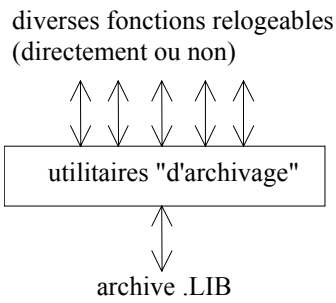
Conclusion :

- Un code objet intermédiaire (destiné à l'éditeur de lien) peut contenir des instructions d'apparence non relogeable (écriture B).
- Mais attention : Un code objet final relogeable, par contre, ne peut pas contenir de telles instructions ! Donc si on désire créer un code final directement relogeable, il faudra les interdire aussi dans tous les codes intermédiaires. Cela est possible, il existe en effet pour certains microcontrôleurs des instructions ou modes d'adressages spéciaux pour cela.

- **Autres codes objets, Bibliothèques ou librairies**

Les fichiers contenant des fonctions définitives, ne devant plus être retouchées, n'ont plus besoin d'être assemblés à chaque fois, et peuvent être pris en compte seulement au niveau de l'édition de lien. On peut les conserver en relogeables.

On peut aussi les regrouper en bibliothèques .LIB (des utilitaires d'archivages existent). Avantage : Une fonction d'une bibliothèque n'est extraite et placée en mémoire que si celle ci est utilisé dans le programme, donc code plus court.



Des fonctions définitives, peuvent aussi être déjà présentes en mémoire morte sur la carte. On doit les prendre en compte en déclarant leurs adresses réelles, ces codes sont évidemment non relogeables … !

On distingue :

Les **bibliothèques ou librairies personnelles**, complétées au fur et à mesure des besoin du programmeur (avec outil Archiveur).

Les **bibliothèques ou librairies** du fabricant permettant une programmation aisée en C. On y trouve des fonctions :

- mathématiques comme sin(), cos(), log(), etc. dont le prototype est décrit dans le fichier « math.h »
- d'entrées/sorties comme getchar(), printf(), sscanf(), etc., prototype dans « stdio.h » utilisable seulement quand l'affichage s'effectue sur la console du PC.
- de manipulation de caractères comme isalnum(), islower(), tolower(), etc, prototype dans « ctype.h »,
- de manipulation de chaînes de caractères comme len(), strcat(), strcpy() etc., prototype dans « string.h »,
- d'utilitaires généraux comme abs(), atoi(), calloc(), free(), rand(), etc., prototype dans « stdlib.h ».
- On y trouve aussi le Cstartup du C nécessaire au bon démarrage du C.

- **L'édition de lien**

Deux rôles :

1) A partir de différents codes objets obtenus par le compilateur et l'assembleur, et les codes de bibliothèques désirés, il fournit un **code objet final** complet. (Il à recalculé les adresses qui ne peuvent pas être connues au moment de l'assemblage séparé de chaque fichier).

2) Permettre de placer en mémoire correctement et d'une manière souple les différentes parties de codes, les **« sections »** (en zones d'adresses réservées pour le code, pour les variables non initialisées, initialisées, pour la pile …) **selon le plan mémoire de la carte** utilisée.

L'éditeur travaille à partir de code relogeable, avec possibilité si nécessaire de quelques parties non relogeables, attention cependant aux conflits alors possibles …. !

Les fichiers (relogeables, librairies) à prendre en compte, les correspondances entre sections et adresses d'implantation sur la carte, quelques options du fichier de sortie, … sont indiqués dans son « **fichier de commande** » dont la syntaxe est évidemment bien spécifique à chaque produit.

- **Le code objet binaire final :**

Destiné à être envoyé vers une carte, ou vers un programmateur extérieur, en plus du code objet exécutable, il contient tous les renseignements destinés au programme de chargement pour l'implantation mémoire des différentes parties du programme. Des options sont définies au moment de l'édition de lien.

Ce code objet final peut être lui même:

- **Absolu :** Cas le plus fréquent, pour des applications microprocesseurs, il ne peut alors tourner qu'à l'adresse fixée lors de l'édition de lien. (On referait une édition de lien pour le placer autre part en mémoire).

- **Relogeable :** Nécessaire sur les plus gros systèmes, ou sur des processeurs performants, sur les ordinateurs. Exemple : fichier exécutable .exe qui se charge et s'exécute sur double click. Les microcontrôleurs simples ne permettent pas facilement la création de code final de ce type, ce qui est nullement un inconvénient !

D'autre part, différentes ***options supplémentaires*** sont possibles au niveau du type de code généré :

- Obtention d'un code (dans un **format natif** spécifique du compilateur-debugger utilisé pour le téléchargement, plus ou moins standard) contenant des informations de mise au point, « Debug » . C'est un mélange de code binaire exécutable, d'information de téléchargement et de mise au point.

- Obtention d'un code également dans ce même format natif en vue d'un téléchargement direct en FLASH par exemple, mais sans informations de mise au point. En final sur l'application ne figurera que son code binaire, donc plus court, pour l'application définitive.

- Le code peut contenir des routines d'entrée sortie sur le clavier et l'écran de l'outil de développement. (Pratique en mise au point).

- Code destiné à être envoyé à un programmateur de mémoire, pour l'application définitive. Un fichier est alors crée, contenant tout le code objet, mais sous la forme de fichier texte, en **binaire codé ASCII**. Ce fichier est visible sur tout éditeur de texte, transportable sur ligne série RS232 (ne contenant pas de caractères de contrôle qui bloquerait la ligne). Le format classique est le « Motorola S1, S9 ». Ce sont des formats reconnus par les cartes « cibles standard, et par les programmateur de PROM …

- Obtention d'un code non exécutable à lui tout seul, mais relogeable, à garder, pour être incorporé en tant que code objet intermédiaire d'une autre application, ou à mettre en bibliothèque.

- **Téléchargement**

C'est l'opération qui consiste simplement à envoyer le code binaire final généré par l'éditeur de lien :

- Vers la carte "cibles" en **mode Debug** (pour la mise au point).
- Vers la carte (mémoires FLASH) ou un programmateur extérieur pour l **'application définitive**, avec évidemment le code juste nécessaire à l'application.

- **Projet**

Un outil de développement permet souvent de travailler sous forme de projet. Un fichier projet (extension fréquente **.PRJ**) spécifie les différents programmes sources en C et en assembleur, ainsi que les bibliothèques déjà compilées sur lesquelles on travaille (ou leur chemin). Diverses options y figurent également (niveau d'optimisation, type de fichiers de sortie, simulateur ou Debugger, …)

Le compilateur, l'assembleur, ainsi que l'éditeur de lien, opèrent alors sur les indications de ce projet, avec un interface graphique pour l'utilisateur.

- **Code réentrant**

Un code (sous programme ou fonction) réentrant peut être interrompu et réutilisé dans une routine d'interruption, le premier calcul se terminant ultérieurement. Un programme interruptible et réentrant (multiplication 32 bits par 32 bits par exemple) pourra donc être interrompu à tout instant et réutilisé sans précautions particulière, d'où l'intérêt pour ce type de programme.

- Les systèmes multitâches ne peuvent tourner sans sous programmes réentrants.
- De même les codes récursifs (fonction qui s'appelle elle même)
- Il faudra remplir les deux conditions suivantes, et nous en reparlerons :

-Tous les **registres** utilisés du processeur sont **sauvegardés** dans une pile lors de l'interruption (cela se fait parfois automatiquement, mais pas toujours)

-Toutes les **variables supplémentaires** (dites "**locales**") éventuellement nécessaires pour les calculs sont **dans une pile,** et **différentes pour chaque appel à la fonction**.

-Les compilateurs C fournissent des codes réentrants pour des fonctions avec paramètres et variables locales.

Nous reverrons ceci ultérieurement.

3. PROGRAMMATION EN ASSEMBLEUR

- Le **code objet** est le code en binaire pure qui est directement compris par un processeur. Pour faciliter la programmation, le premier langage, le plus proche du processeur est le langage Assembleur, spécifique d'un processeur donné.
- Le langage assembleur est un langage source qui permet :
  De remplacer les valeurs numériques (adresses ou données) par des **symboles**.
  De remplacer les codes d'instructions par des **mnémonique**
  D'effectuer une fois pour toute à l'assemblage certains calculs.

Il délivre un **listing** avec des messages d'erreurs et des informations.

Un programme en langage assembleur est saisi à l'aide d'un **Editeur** de texte et non un traitement de texte ! qui rajoute toute une série de code de mise en page). C'est le **programme source.**

- La traduction du programme source en **code objet** est assurée par le **programme ASSEMBLEUR**.

Les **directives d'assemblage** dirigent le code vers tel ou telle section.

Section absolue : pour le code Absolu

L'adresse est définie lors de l'assemblage : possible pour des petits programmes. A utiliser le moins possible, très pratique par contre pour la section des vecteurs d'interruption d'un processeur.

Section relative : pour le code Relogeable

Adresse définie à l'édition de lien: bien plus souple. On travaille pratiquement presque toujours ainsi.

- L'**ÉDITEUR DE LIEN (LINK)** permet de rassembler plusieurs codes objets si nécessaires, et aussi d'implanter correctement et d'une manière très souple le code selon la carte utilisée.

Son ***fichier de configuration*** spécifie les différentes « **sections »** à répartir :

Section de programme (instructions exécutables)
Section de variables non initialisées
Section de variables ou constantes initialisées
Section de la pile
…….

Important :

Selon les différents types de processeurs et de constructeurs, il existe d'**importantes variantes de syntaxe**, surtout pour les directives d'assemblage et pour le fichier de commande de l'édition de lien. Mais les lignes générales sont les mêmes.

3.1. Résumé du langage assembleur

3.1.1. Les bases

3.1.1.1. Caractères reconnus par l'assembleur

L'assembleur reconnaît généralement la majorité des caractères ASCII, les caractères codés de $20 à $7F en particulier :

- l'alphabet majuscule de **A à Z** et minuscule de **a à z**
- les chiffres de **0 à 9**
- les quatre opérateurs arithmétiques **+ - * /**
- le caractère "espace" (SP)
- les caractères spéciaux suivants, utilisés comme préfixes

# (dièse)	: indique l'adressage **immédiat**
$ (dollar)	: indique un nombre **hexadécimal**
% (pour cent)	: indique un nombre **binaire**
' '	: indique un ou plusieurs **caractère** ASCII
" "	: **chaînes de caractères** (avec $00 de fin de chaîne)

3.1.1.2. Format général d'une ligne

En langage assembleur on écrit une instruction par ligne. Une ligne source est divisée en quatre champs séparés entre eux par au moins un caractère espace (**SP**).

Elle **commence** toujours en **première colonne** !

Elle se termine par un caractère "retour chariot" ou « **C**arriage **R**eturn » **CR**.

champ étiquette	champ opération	champ opérande	commentaires

3.1.1.2.1. CHAMP ETIQUETTE

Une étiquette est un symbole.

Le caractère **espace** à la place du premier caractère d'une étiquette indique qu'il n'y a pas d'étiquette. On entre ensuite dans le champ opération.

Le caractère ***** à la place du premier caractère du champ étiquette transforme toute la ligne en champ **commentaire**.

Une ***étiquette*** est utilisée dans les cas suivants :

- Pour repérer un **début** de programme ou sous programme
- pour repérer une **ligne destination** d'une instruction de branchement ou de saut.
- avec la directive EQU, pour déclarer qu'un symbole est une valeur ou l'adresse d'une valeur.
- pour repérer des zone mémoire définie par les directives d'assemblages écrivant des valeurs en mémoire (FCB, FCC, FCS, FDB), ou réservant de la place en mémoire (RMB). Voir plus loin ….

3.1.1.2.2. CHAMP OPERATION

Le champ opération peut contenir :

- la mnémonique d'une **instruction** exécutable
- la mnémonique d'une **directive d'assemblage**
- l'appel d'une **macro** instruction

3.1.1.2.3. CHAMP OPERANDE

L'assembleur reconnaît dans le champ opérande : **nombres, symboles,** et **expressions**

Le contenu du champ opérande précise en outre le mode d'adressage des instructions exécutables.

Les nombres :

Nombre décimal : NOMBRE
Nombre hexadécimal : $NOMBRE
Nombre binaire : % NOMBRE

Les symboles :

Peuvent être des **variables** ou des **adresses**.

En outre, le **symbole spécial *** (astérisque) représente le « pointeur courant d'assemblage ». (Lors de l'assemblage ce pointeur s'incrémente régulièrement, et permet ainsi de générer du code ou des valeurs dans les zones mémoires concernées).

Les expressions :

Une expression est une combinaison de symboles et/ou de nombres séparés par des opérateurs +, -, * et /.

L'assembleur évalue les expressions algébriquement de la gauche vers la droite, en l'absence de parenthèses. Un résultat fractionnaire, total ou partiel sera arrondi à une valeur entière par l'assembleur.

Exemples:

MOT EQU $1000 le symbole mot équivaut à $1000 (valeur ou adresse)
(L'opérande MOT + 8 représente: $1000 + 8 = $1008)
VALEUR EQU 10*7/3 donne 23 en décimal
Pour écrire 0,257 sur 8 bits en ,-------- non signé : val equ 257*256/1000
Pour écrire 0,257 sur 16 bits en -,------- -------- signé : val equ 257*32768/1000
(se reporter à la partie arithmétique pour les formats non entiers)

Attention, ces calculs ne sont effectués qu'une seule fois lors de l'assemblage du programme, et en aucun cas lors de l'exécution du programme...

3.1.1.2.4. CHAMP COMMENTAIRES

En outre, une ligne de programme source qui commence par le caractère * devient une ligne complète de commentaire.

Il est inutile de rappeler l'intérêt d'un commentaire pertinent pour la relecture d'un programme !

3.1.2. Les Modes d'adressage

Le mode d'adressage d'une instruction est le plus souvent précisé par le format du champ opérande.

3.1.2.1. Adressage d'accumulateur

Instruction travaillant sur un accumulateur, sans autres opérandes : remise à zéro, complément à 2 …

Ex en HC11,HC12	Ex en type 68000
CLRA mis à zéro de l'accumulateur A (en HC11)	**CLR.W** mis à 0 des 16 bits LSB du registre D1

3.1.2.2. Adressage implicite ou inhérent

Il n'y a pas d'ambiguïté, la mnémonique suffit, le champ opérande est vide: instruction de multiplication entre deux registres (cas de processeurs simples ou le produit s'effectue toujours entre ceux ci). Ex : **MUL** en HC11,HC12 produit entre les accu A et B

3.1.2.3. Adressage étendu (ou absolu)

Le champ opérande est l'adresse de l'opérande.

Ex en HC11,HC12	Ex en type 68000
LDAA $500 Chargement(LOAD) de l'accu A avec le contenu de l'adresse $500 **LDD $500** Chargement(LOAD) de l'accumulateur D (en fait A et B pris ensemble) par la valeur de 2 octets située en $500,$501 (octet haut en premier en Motorola)	MOVE.B $500,D0 Chargement dans le registre D0 d'une valeur 8 bits, situé à $500 MOVE.W $500,D0 MOVE.L $500,D0 Chargement dans le registre D0 de valeurs de 16 et 32 bits, situées à partir de l'adresse $500

3.1.2.4. Adressage direct

Ce terme, un peu spécifique de motorola, est en fait un adressage étendu court pour la petite zone d'adresses entre 00 et FF. L'écriture est identique, seul le code objet un peu plus court permet une exécution plus rapide. On s'en sert en fait peu.

3.1.2.5. Adressage immédiat

Il est repéré par le caractère # comme premier caractère du champ opérande. Le champ opérande prend alors un des formats suivants. ***Le champ opérande est la valeur à charger :***

Ex en HC11,HC12	Ex en type 68000
LDAA #10 Chargement (LOAD) de l'accu A avec 10 (décimal) **LDX #$1000** Chargement de l'accu X avec l'adresse hexa $1000 (gestion de tableaux, pointeurs ….)	**MOVE.L #tab,A0** Chargement de l'adresse du début du tableau tab dans le registre A0

3.1.2.6. Adressage relatif et relatif/pc

Le champ opérande est le décalage par rapport au compteur programme pour accéder à l'opérande ou pour se brancher.

Si ce déplacement signé ne tient pas sur 8 bits (>127 ou <-128) on peut avoir un message d'erreur. Il faut alors utiliser, si elles existent, des instructions de déplacement plus loin , ou sinon l'adressage étendu.

Ex : **BSR calcul1** saut au sous programme calcul1 (adressage relatif)

JSR calcul1 en adressage étendu fait la même chose, et permet des sauts plus éloignés, mais le code n'est alors plus relogeable sans intervention de l'éditeur de lien! (il fait référence à une adresse réelle dans le code lui même).

MOVE.W VALEUR(PC) en type 68000: accès à VALEUR par décalage par rapport au PC (pour créer un code directement relogeable). Ce mode n'existe pas pour de nombreux petits microprocesseurs.

3.1.2.7. Adressage indexé, ou indirect par registre

Un registre contient l'adresse de l'opérande. Des décalages fixes ou variables sont parfois possibles. Application : **gestion de tableaux**, simulant des variables indicées .

Ex en HC11,HC12	Ex en 68000
LDX #tab X pointe le début du tableau tab **LDAA 0,X** Accès au premier élément du tableau	**MOVE.L #tab,A0** A0 pointe le début du tableau tab **MOVE.W (A0)** accès au premier élément du tableau

***Indexé par rapport à S*:** certains processeurs autorisent l'indexé sur le pointeur de pile S, ce qui est très pratique pour aller chercher très rapidement des valeurs dans la pile.

3.1.2.8. Modes d'adressages plus complexes, selon les processeurs

Indexés avec décalage accumulateurs (HC12)
Indexés indirects
Auto incrémentations ou décrémentations en indexé (HC12) ou indirect par registre.
Adressages sur 8, 16 ou 32 bits (HC12).
Etc ……

3.1.3. Les Directives d'assemblage

- Pour déclarer des **sections relogeables** ou **absolues.**
- Pour **définir étiquettes** et **variables**.
- Pour **réserver de la place** en mémoire...

Tous ces renseignements étant nécessaires au niveau de l'assembleur ou au niveau de l'édition de lien.

Certaines notations sont employées par la plupart des assembleurs, d'autres comme par exemple les déclarations de section et la syntaxe du fichier de commande de l'édition de lien sont très spécifique à un produit.
Pour celles ci , nous prendrons la **syntaxe** du compilateur **IAR Systems.**

3.1.3.1. Déclaration de sections : ORG RSEG …

♦ **Section absolue** (donc non relogeable)
Déclaration d'origine : **ORG $1000**

Elle permet de générer du code ou des données à l'adresse indiquée, et non à partir de 0.
Une directive **ASEG** signifiant section absolue, est parfois utilisée et doit précéder la directive ORG (IAR Systems).

♦ **Section relogeable :**
ou **RSEG nom** (Relocatable segment, IAR Systems)

Les adresses de ces sections sont définies à l'édition de lien.

3.1.3.2. Mise d'un label sur une adresse fixe : EQU

Pratique pour déclarer **l'adresse de base et les registres internes d'un circuit périphérique,** interne ou externe au microcontrôleur.
Exemple un circuit d'entrée sortie parallèle 8 bits, programmable au moyen des 3 registres : PORT (registre d'entrée sortie proprement dit), DDR (registre de direction), CTRL (registre de contrôle), et placé à partir de $6000 :

Label égal une adresse :	**Label = décalage par rapport à l'adresse de base :**
base equ $6000 PORT equ base DDR equ base+1 CTRL	Base equ $6000 PORT equ 0 DDR equ 1 CTRL equ 2
Accès par l'adresse : **En HC11,HC12 :** ldaa PORT PORT dans A	**Accès par un pointeur sur la base du circuit** **En HC11,12 :** ldx #base ldaa PORT,X PORT dans A

- Il vaut mieux écrire une seule fois une adresse (comme ici $6000), ainsi si on change un jour l'adresse de base du circuit, l'adresse de tous ses registres internes change également.
- L'accès par un label représentant une adresse est plus simple, et plus évident …
- L'accès par label = décalage est cependant un plus rapide d'exécution une fois que le pointeur est positionné sur le circuit (l'adresse n'est pas à relire en mémoire à chaque fois).

Cette façon de faire est pratique surtout pour des processeurs (tels que les 68xxx), ayant de nombreux registres pouvant travailler en mode indirection avec décalage, ou en indexé, l'un restant chargé en permanence à l'adresse de base du circuit.

3.1.3.3. Mise d'un label sur une adresse en code relogeable

- Si on veut mettre un label en début d'uns section par exemple :

```
        RSEG SECTION        Nom de la section relogeable
Debut_section   equ   *
```

Le * **signifie** adresse courante lors de l'assemblage, puis lors de l'édition de lien.

- Remplacement d'une valeur par un label : EQU

```
DEUX   equ   2
```

Le symbole DEUX est remplacé automatiquement par le **caractère** 2. Cette valeur n'est pas en mémoire ! .

3.1.3.4. Déclaration de variables non initialisées

3.1.3.4.1. DIRECTIVE EQU ?

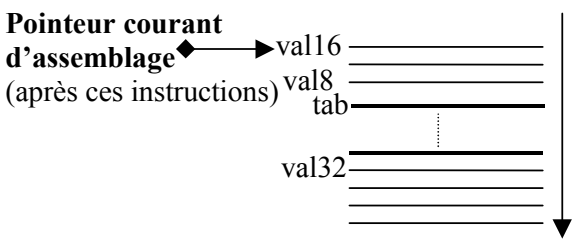
Soit a déclarer et positionner à partir de $D000, dans l'ordre, une valeur 16 bits, une valeur 8 bits, un tableau de taille maximale taille_max égale à 100, une valeur 32 bits.

Idée (non relogeable) :

```
taille_max   equ   100
val16        equ   $D000          valeur 16 bits
val8         equ   val16+2              valeur 8 bits, en fait on voit ici que val16 est sur 16 bits.
tab  equ     val8+1               début du tableau "tab"  on voit ici que val8 fait 8 bits …
val32        equ   tab+taille_max       on voit ici la taille max possible pour le tableau.
```

- ***MAIS ATTENTION DANGER …. !***

Avec cette technique, le « pointeur d'assemblage » n'avance pas, et il n'est même pas défini dans le premier cas. Dans le second cas, si on place ensuite une directive qui écrit du code ou des valeurs en mémoire, ce code ou ces valeurs seront placées au début de la section, et donc écraseront les zones réservées !



Donc ne **jamais** se servir de directives **equ** pour déclarer des variables, il faut leur réserver de la place par la directive suivante d'allocation de mémoire.

3.1.3.4.2. DIRECTIVE RMB

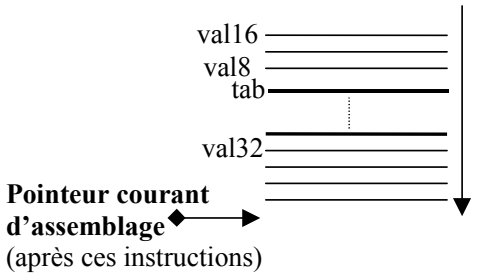
On utilise l'instruction: **Rmb n** (reserve memory byte): réservation de n octets à l'adresse courante d'assemblage.

Possible dans une section absolue, ou relogeable.

Exemple : On peut utiliser la directive Rmb pour déclarer comme précédemment val16, val8, tab, et val32 :

```
                ORG    $D000   (ou un nom de section :   RSEG  VARIABLES)
Taille_max  equ  100
val16           rmb    2            on définit les tailles ici sur la même ligne !
val8            rmb    1
tab             rmb    taille_max
val32           rmb    4
```

Remarque :
Ici, le « pointeur d'assemblage » avance à chaque instructions, et si on ajoute des lignes générant code ou valeurs, tout se placera à la suite, donc pas de problème, il faut donc toujours utiliser RMB et non EQU pour déclarer des variables !



3.1.3.5. Ecritures de valeurs en mémoire FCB, FDB

Ces directives placent des valeurs (8 ou 16 bits) en mémoire dans la **section courante** d'assemblage (On trouve dans d'autres systèmes de développement des directives équivalentes : DC.B DC.W ….)

Exemple : on veut placer en mémoire des valeurs, en leurs associant un label :

- **val16**, **val8**, et **val32** initialisées à $1000, 12, $F000101A respectivement
- **caract** et **mot1** valant le caractère **F** et la chaîne **bonjour** respectivement.

```
   ORG      …..     (ou un nom de section :   RSEG …….)
val16               FDB  $1000                FDB  force double byte
val8                FCB   12                  FCB  force  constant byte
val32               FDB   $F000,$101A
caract              FCB   'F'
mot1                FCB  'bonjour'            sans caractère fin de chaine $00
                    ou  FCB  "bonjour "       avec caractère fin de chaine $00
```

La distinction entre constantes et variables initialisées sera vue ultérieurement.

3.1.3.6. Déclaration d'importation ou d'exportation de symbole

Pour travailler sur plusieurs fichiers :

- **Extern** ou **global** (selon les assembleurs)

Pour dire qu'un symbole est défini par ailleurs, dans un autre fichier

- **Public** ou **def** ou **global** (selon les assembleurs)

Pour dire qu'un symbole doit être exporté pour pouvoir être connu dans un autre fichier (par directive extern ou global)

3.1.3.7. Autres directives d'assemblage

Les notations dépendent de l'assembleur utilisé …

- **rept** nombre et **endr**

répétition de nombre fois les lignes entre ces deux directives.

Exemple : on veut placer en mémoire 80 fois la valeur 0 sur 16 bits, on fera :

```
rept  80
fdb   0
endr
```

- **set**

déclaration et initialisation de variables d'assemblage

Exemple : écriture en mémoire des valeurs de 0 à 99

```
i   set     0
    rept    100
    fcb     i
i   set     i+1
    endr
```

3.1.4. #define et Assemblage conditionnel

Remarque : les notations peuvent être différentes pour d'autres assembleur, mais souvent ce sont les mêmes qu'en C.

Les # peuvent souvent s'omettre en assembleur, ou même parfois ne doivent pas se mettre !. Le #include doit s'écrire parfois .include

Le **#define** avec valeur numérique peut être remplacé par un **set** vu précédemment, mais pas compatible avec le C.

3.1.4.1. #define

Permet un peu comme equ de dire qu'un nom est défini ou qu'un nom peut être remplacé par un autre nom ou d'autres caractères, pour faciliter la compréhension. Mais la notation est ici identique au C comme on le verra. Ce que l'on définit ainsi n'est **donc pas en mémoire** !

On peut avoir quelques constantes qu'il est inutile de mettre en mémoire, mais qui doivent être utilisées dans plusieurs fichiers du projet (aussi bien fichiers C que fichiers assembleurs).

Exemple :

la fréquence choisie F_BUS de travail du microcontrôleur HC12 (valeurs possibles 2,4,8,16,20,24 MHz) peut être définie par **#define F_BUS 2**

On désire que cette valeur soit connue dans tous les fichiers (elle conditionne en effet la programmation de nombreux registres). Les directives extern et public sont sans objet, puisqu'il ne s'agit pas de variables ou constantes en mémoire !

Solution :

- On écrit un fichier d'entête par exemple **ini_carte.h**

Remarque : les directives #ifnfef (si non défini) et #endif (fin de if) sont nécessaires car ce fichier d'en tête étant à priori inclus dans plusieurs autres fichiers, un message d'erreur de multiples définition interviendrait !

```
#ifndef  F_BUS
#define  F_BUS 2
#endif
```

- On inclut ce fichier dans tous les fichiers assembleurs (et C comme nous le verrons) du projet, par **#include ″ini_carte.h″.**

Fonctionnement: Partout ou est écrit F_BUS, et dans tout le projet, l'assembleur ou le compilateur remplace cette écriture par 2.

3.1.4.2. Assemblage conditionnel

Une même source peut pour diverses applications être assemble de plusieurs façons selon par exemple des options choisies ou non, selon qu'il s'agit d'une version réduite ou complète….

Si on définit un symbole dans un seul fichier, le **#ifndef** …. **#endif** n'est pas nécessaire.

Obligatoire par contre si on définit le symbole dans un fichier.h à inclure dans plusieurs autres, comme précédemment.

Les # sont souvent optionnels ou en assembleur, sauf pour les #define.

3.1.4.2.1. COMPARAISON A VALEUR NUMERIQUE

On peut définir et donner une équivalence pour un symbole VERSION par :

#define VERSION 1 ou **VERSION set 1**

Et dans le code on écrira :

(Attention : pas d'espace autour du = et valeur Numérique nécessaire)

souvent optionnel ou écriture sans #nécessaire…

```
#if VERSION=1
    lignes d'assembleur
#else
    autres lignes d'assembleur
#endif
```

3.1.4.2.2. SI UN SYMBOLE EST DEFINI OU NON :

On peut juste définir un symbole : VERSION_1 ou VERSION_2 mais sans donner d'équivalence.

#define VERSION_1

Et dans le code on écrira :

On peut aussi utiliser des #else

```
#ifdef VERSION_1
    lignes d'assembleur
#endif
#ifdef VERSION_2
    autres lignes d'assembleur
#endif
```

3.1.5. Macro

3.1.5.1. Définitions

Quand une suite d'instructions est utilisée à plusieurs reprises, avec possibilité de changer à chaque appel quelques paramètres, on peut utiliser les « **macro instructions** »

C'est une façon élégante de créer des fonctions quand on ne travaille pas dans un environnement "langage évolué".

L'assemblage conditionnel permettant à un même programme assembleur d'être assemblé de différentes façons pour s'adapter par exemple à une carte ou un type de processeur peut évidemment être utilisé dans les macro.

Les Macros et l'assemblage conditionnel évitent la répétition de lignes d'instructions, et permettent par exemple de réaliser des fonctions en assembleur. On peut ainsi facilement générer les lignes de code permettant l'envoi de paramètres à un sous programme (nous verrons cela ultérieurement).

Attention :

- Les paramètres ***sont*** *passés* à la macro **une fois pour toute lors de l'assemblage du programme**, et nullement à l'exécution !, et *sous forme de* ***chaînes de caractères***.
- Il y a donc **génération de code objet à chaque appel**.

Ces paramètres passés sont :

-Des **modes d'adressages différents**: **immédiat**, **étendu**, **relatif ....,** (attention à l'existence de tous les modes dans la programmation de la macro, si un mode n'existe pas, on aura un message d'erreur à l'assemblage).

-Des **caractères**, des **valeurs** ou des **mots d'aiguillage**, permettant des assemblages totalement différents.

3.1.5.2. Exemple avec : if else endif

Soit à réaliser une macro, que l'on nommera **affiche** qui affiche soit des valeurs numériques, soit des chaînes de caractères. On écrira ici des notations sans #.

On choisit les paramètres :

-Le premier paramètre (notation **/1**) sera par exemple val ou mot, il permet d'aiguiller l'assemblage différemment.

-Le second (/**2**) le renseignement concernant l'adresse la valeur ou la chaîne à afficher. (attention toutefois aux modes interdits de certaines instructions).

-Le troisième (/**3**) la taille en bits (8, 16 ou 32) de la valeur à afficher.

- **Description de la Macro:**

```
affiche  macro
                    if  /1="val"
                    conversion et affichage de /2 dans le format /3
                    endif
                    if  /1="mot"
                    affichage de la chaîne commencant à l'adresse /2
                    endif
endm
```

(le **if** porte sur un **test de la chaine** /1, et est une **directive** d'assemblage conditionnel. Ce n'est nullement une instruction assembleur !)

- ***Utilisation de la Macro:***

Affiche(val, valeur, 16) affiche de la valeur 16 bits à l'adresse valeur

Affiche(mot, #message1) affiche le message1 débutant à l'adresse message1.

(Remarque, pour afficher une chaîne, soit elle se termine par le caractère NULL, soit on définit sa longueur que l'on passerait en paramètre /3).

Lors de chacun de ces appels, le code assembleur généré (et le code objet) est différent: certains modes d'adressages différent suivant /2 et /3, le code diffère totalement suivant /1 !!.

L'intérêt de ces Macros est évident: moins d'erreurs de programmation, création d'un langage évolué personnel propre à ses besoins.

3.1.5.3. Remarques

L'usage des Macro en assembleur est moins répandu de nos jours, depuis la généralisation de la programmation des microprocesseurs en langage évolué.

Cependant le principe de programmation par Macro existe aussi à d'autres niveaux :

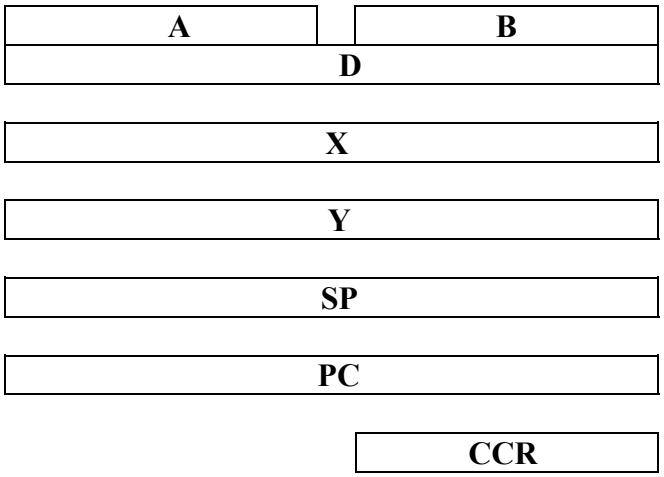
- On peut écrire des Macro en C
- On se sert de Macro dans de nombreux logiciels (traitement de texte, tableur …). Le principe restant un peu le même : se créer des fonctions ou automatiser une suite de taches.

3.2. L'Assembleur type HC12, résumé

Se reporter à la doc en Annexe

3.2.1. Registres de base

- Accumulateurs 8 bits **A et B** (formant ensemble l'accu **D** de 16 bits)
- Registres d'index **X et Y** de 16 bits
- Registre pointeur de pile **SP** de 16 bits
- Compteur de Programme **PC** de 16 bits (ou compteur ordinal)
- Code Condition Register **CCR** de 8 bits



accumulateur 8 bits A et B
ou accumulateur D sur 16 bits

registre d'index X

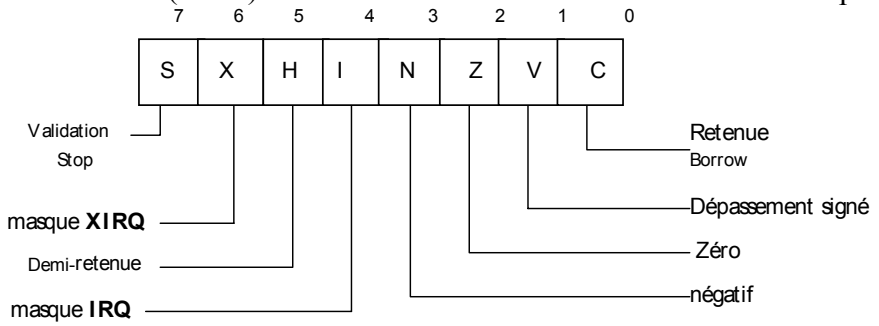
registre d'index Y

pointeur de pile système

compteur ordinal

registre de code condition

Registre de code condition (CCR) :
Le registre code condition (8 bits) définit à tout instant l'état des indicateurs du processeur :



Ligne **IRQ** interruption masquable
Ligne **XIRQ** interruption non masquable (après démasquage initial)

Dans ce qui suit, la plus part des exemples travaillent sur **l'accu A.**
On trouvera dans la doc complète les mêmes instructions travaillant sur **l'accu B.**

→ Travail sur des oparandes de 16 bits :
Certaines instructions (écritures ou lectures mémoires, opérations arithmétiques simples) peuvent travailler sur « **l'accu D »** qui est la réunion des accus A et B (A étant alors l'octet de fort poids), ou même sur les registres X ou Y Ceci permet ainsi de **travailler** directement sur des **opérandes 16 bits**.

Important : En **Motorola**, les opérandes de plus de 8 bits sont toujours placés en mémoire **OCTET HAUT EN PREMIER** (Ordre croissant des adresses). C'est l'inverse en Intel.

3.2.2. Modes d'adressages

On trouve seulement les modes de base :

Immédiat LDAA #..
Etendu LDAA valeur
Indexé LDAA d,x (avec décalage d) **sur X et Y,** et **S** (**en HC12**).
LDA 0,X (manipulation d'un octet)
LDD 0,X (manipulation de deux octets en X et X+1)
LDA 0,SP sur le pointeur de pile S (en HC12 seulement)
Relatif : banchements BRA …

D'autres modes existent, non décrits ici.

Deux autres nouveautés de l'HC12 par rapport à l'HC11 :

- Auto incrémentation des pointeurs X ou Y, avant (pré) ou après (post) l'adressage.

On peut donc remplacer : LDAA 0,X *Par : LDAA +1,X*
(CompatibleHC11) INX

Et aussi : LDD 0,X *Par : LDD +2,X*
INX
INX *Bien plus rapide pour boucles et tableaux.*

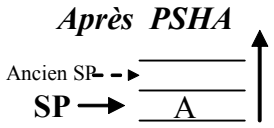
- Calcul d'adresse sans adressage, Instructions **LEA** : par exemple INX INX peuvent se remplacer par LEAX 2,X

3.2.3. Jeu d'instruction (résumé)

Remarque *: Contrairement à certains processeurs (ancien 68HC11 et bien autres) l'HC12 peut effectuer directement des opération mémoire à mémoire.*

3.2.3.1. Instructions de chargement de stockage et de transfert

Les deux instructions spéciales **PSHA** et **PULA** agissent sur la **Pile** (**Stack** en anglais). L'empilement s'effectue par pré-décrémentation puis écriture au niveau du pointeur de pile. (Remarque, c'était l'inverse en HC11… !).



Mnémonique	**Opération** (**M** = mémoire)
LDAA	M → A
STAA	A → M
CLRA	0 → A
TAB ou TBA	A → B ou B → A
PSHA	SP –1 → SP et A → Stack
PULA	Stack → A et SP +1 → SP
TPA ou TAP	CCR → A ou A → CCR
TSX (ou TSY)	SP + 1 → X **X pointe le dernier empilé**
TXS (ou TYS)	X-1 → SP
XGDX	**Echange** X et D
XGDY	**Echange** Y et D

3.2.3.2. Instructions de transfert mémoire ←→mémoire

MOVB source,destination Pour un octet.
MOVW source,destination Pour un mot de 16 bits.

3.2.3.3. Instructions arithmétiques de base

- Les instructions de comparaisons (TSTA, CBA, CMPA) effectuent une différence entre deux registres (A et B) ou entre un registre et une case mémoire (A et M ou A et 0). Elles n'affectent pas le contenu des opérandes (registres ou mémoire) mais positionnent les bits Z, N, V, C du CCR qui pourront être utilisés à la suite de ses instructions.
- NEGA effectue le **complément à 2** du registre A : par exemple 54h qui vaut 84d, à pour complément à 2 : 0 – 54h ou alors 100h – 54h = ACh = -84d.
- COMA effectue le **complément à 1** du registre A, elle est classée parmi les instructions de type logique : $A \leftarrow (\overline{A})$.

Mnémonique	Opération
ABA	A + B → A
ADCA	A + M + Carry → A
ADDA	A + M → A
CBA	Test B - A
CMPA	Test A – M
DECA	A – 1 → A
INCA	A + 1 → A
NEGA	0 – A → A
SUBA	A – M → A
SBCA	A – M - C → A
TSTA	Test A – 0

3.2.3.4. Multiplications et divisions

Structure câblée directement à l'intérieur de la puce.

♦ **Multiplication non signée** seulement

MUL	A * B → D

♦ **Division non signée**:

La division peut être entière ou fractionnaire.
Les opérandes sont sur 16 bits, on obtient un quotient **Q** un reste **R**

FDIV de D/X	Q → X , R → D
IDIV de D/X	Q → X , R → D

- ***Division entière*** **IDIV**

$$N/D = Q + R/D$$

Q est le quotient, R est le reste de la division.

Exemple. N = 14A2h divisé par D = 01CBh, soit N = 5282d divisé par D = 459d, donne : $N/D = 5282/459 = Q + R/D = 11+233/459$

***Remarque** : IDIV retourne Q = FFFFh dans le cas d'une division par zéro (D = 0).*

- ***Division fractionnaire*** (ou en virgule fixe) **FDIV**.

Il faut que Numérateur < Dénominateur et donc Q < 1 interprété en Q16(16) (16 bits fractionnaires). Les valeurs limites pour le quotient Q sont :

Q_{min} = 0001h = 2^{-16} = 0.00001525

Q_{max} = FFFFh = 1 - 2^{-16} = 0.99998475

***Remarque**: FDIV retourne Q = FFFFh dans le cas d'une division par zéro (D = 0) ou dans le cas ou le numérateur N est supérieur ou égal au dénominateur D.*

Exemple. Soit N = 42784 et D = 52669 le quotient vaut $CFF4 donc en Q16(16) : 52236/65536 = 0,81231689

- ***Cas général N/D = Q + R/D***

Avec ces deux divisions on peut facilement effectuer toute division de deux entiers (ou même position de la virgule …) 16 bits non signés. Une première division par IDIV de N/D fournit le quotient Q entier. Une deuxième division par FDIV du reste R par le diviseur D donne la partie fractionnaire sur 16 bits. D'ou un résultat possible sur 32 bits, 16 bits entiers et 16 bits fractionnaires

3.2.3.5. Instructions logiques

L'instruction BITA effectue le « ET » logique entre le registre A et la mémoire M sans affecter le contenu du registre A.

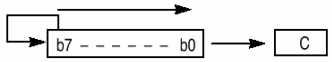
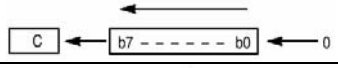
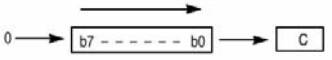
Mnémonique	Opération
ANDA	$A \cdot M \rightarrow A$
BITA	$A \cdot M$
COMA	$\overline{(A)} \rightarrow A$
EORA	$A \oplus M \rightarrow A$
ORAA	$A + M \rightarrow A$

3.2.3.6. Décalages et rotations

Ces instructions utilisent toutes la Carry, bit C du CCR en conjonction avec un registre (ou une mémoire). On peut les diviser en deux catégories :

Instructions de type logique : RORA, ROLA, LSRA, LSLA (opérandes non signés)

Instructions de type arithmétique : ASLA, ASRA (opérandes signés)

Mnémonique	Opération
ASLA	
ASRA	
LSLA	
LSRA	
ROLA	Permutation circulaire à gauche du registre et de la Carry
RORA	Permutation circulaire à droite du registre et de la Carry

3.2.3.7. Instructions intervenant directement sur la mémoire

On peut citer les instructions suivantes :

CLR, DEC, INC, NEG, TEST, COM, ASR, ASL, LSL, LSR, ROL, ROR, etc.

Utiliser ces instructions peut être pratique car on ne modifie pas le contenu d'un registre interne du microcontrôleur.

3.2.3.8. Instructions de branchement et d'interruption

Mnémonique	Opération
JMP	Saut *inconditionnel* (en adressage étendu)
BRA	Branchement *inconditionnel* (en adressage relatif)
BEQ, BNE, BPL, BPI, BGT, BLE ….	Tous les branchements *conditionnels* classiques (en adressage relatif), sur des opérandes quelconques, signés ou non signé. Le test porte toujours sur les bits du Code condition de l'instruction précédente, voir le tableau en annexe.
JSR	Saut à un **sous-programme** (en adressage étendu)
BSR	Branchement à un sous-programme (en adressage relatif)
RTS	Retour de sous-programme
CLI	Validation des interruptions générales (masque I à 0)
SEI	Masquage de toutes interruptions (masque à 1)
RTI	Retour d'**interruption**
SWI	Interruption logicielle
WAI	Attente d'interruption

On trouve aussi des instructions de décrémentation et branchement
Pour optimiser l'écriture ders boucles : **DBNE …**

3.2.3.9. Instructions agissant sur le Code Condition

Quelques instructions spécifiques agissent individuellement sur certains bits du CCR :
CLI et **SEI** vus ci-dessus.
CLC met à zéro la Carry C
SEC la met à un

CLV met à zéro le bit overflow V
SEV le met à un

TPA et **TAP** (transfert du CCR dans A et réciproquement) permet par exemple de modifier directement des bits dans le CCR, en passant par A.

3.2.4. Test et manipulations de bits en assembleur

On a souvent besoin sur un octet (ou un mot de 16 bits) :
De tester la valeur d'un bit pour effectuer ensuite telle ou telle action
De manipuler un bit sans modifier les autres.

3.2.4.1. Tests de bits

1) Méthode

On **isole toujours le bit** à tester par un **ET logique** (MASQUE) avec un opérande ayant un bit à 1 à l'endroit utile.

On se sert des instructions classiques : le ET logique : **AND**
Puis **BEQ BNE**, **BMI BPL** pour juste tester le poids fort

CMP comparaison (à la différence d'une soustraction seuls les bits du code condition évolue, le résultat est mis nulle part).

2) Exemples

a) **Attente** que le bit 4 du « **status** » d'un circuit passe à **1** (ou à **0)**
(b7 b6 b5 **b4** b3 b2 b1 b0)

```
STATUS  equ  ....
BOUCLE  LDAA   STATUS
        ANDA   #%000 10000
        BEQ (ou BNE)   BOUCLE   [ BEQ attente tant que 0, BNE attente tant que ≠0 ]
```

b) Attente du bit 4 **ou** du bit 0 (b7 b6 b5 **b4** b3 b2 b1 **b0**)

```
BOUCLE   LDAA   STATUS
         ANDA   #%000 10001
         BEQ   BOUCLE
```

c) Attente des deux bits 4 **et** 0 : (b7 b6 b5 **b4** b3 b2 b1 **b0**)

```
BOUCLE    LDAA   STATUS
          ANDA   #%00010001
          CMPA   #%00010001
          BEQ   BOUCLE
```

d) Test du bit le plus à gauche, façon plus rapide : (**b7** b6 b5 b4 b3 b2 b1 b0)

```
   LDAA   STATUS
   BMI    OUI          (en le considérant comme un bit de signe )
```

3.2.4.2. Manipulation de bits

1) Méthode générale

Ceci est valable pour de **braves variables** en mémoires ou pour des registres standards fonctionnant en écriture et lecture.

Remarque : pour certains microcontroleur, il existe aussi des instrcutions type **BCLR** et **BSET** (non étudiées ici).

Attention, ceci n'est pas valable pour certains drapeaux (d'interruption ou autres) certaines se remettent parfois à 0 en écrivant 1, ou en lisant ou écrivant à une autre adresse ! Voir la Doc technique et adapter ! !

2) Exemples

a) **Mise à 1** du bit 4 (valeur sur 8 bits)

```
        LDAA   valeur
        ORA    #$10   (ou #%00010000)
                                  STAA   valeur
```

b) **Mise à 0** du bit 4

```
        LDAA   valeur
        ANDA    #%EF  (ou #%11101111)
        STAA   valeur
```

c) **Changement d'état** du bit 4

```
        LDAA   valeur
        EORA    #$10   ou exclusif
        STAA   valeur
```

3.2.5. Comparaisons de nombres en assembleur

Instructions de comparaisons du type : **CMPxx** (Compare)

- **CMPA**, CMPB sur 8 bits
- **CMPD ou CPX** … pour **deux** octets
- Pour des valeurs sur n octets, on effectue une soustraction en commençant par l'octet bas et avec la soustraction avec retenue.

Ce n'est que sur la **dernière opération** que tous les **indicateurs** seront **significatifs** pour des branchements conditionnels ultérieurs.

- Dans tous les cas, les instructions de branchement dépendent **de l'interprétation du nombre :** ***signé*** (mode complément vrai) ***ou non signé.***
  En **signé, N et V** et Z sont significatifs
  En **non signé** seulement **C** et Z

Voir ce que testent les instructions dans les tableaux d'instructions, et se reporter à la partie arithmétique binaire du cours).

3.2.6. Instructions performantes spéciales de traitement de signal

On trouve sur ce processeur HC12 des instructions telles que :

Produits 16 bits par 16 bits at accumulation sur 32 bits (EMACS)
Divisions 32 bits par 16 bits
Instructions de calcul de **minimum**, de **maximum**, de **moyenne,** de **pondérations …**
Etc, etc ….

Ce processeur permet donc du traitement du signal avec des instructions performantes, non étudiées ici. Se reporter aux tableaux d'instructions et à la notice complète du produit si nécessaire….

3.2.7. Vecteurs d'interruptions du HC12

On distingue les interruptions externes (IRQ, XIRQ, RESET) et les interruptions de tous les périphériques internes.

Se reporter à la doc en annexe, aux chapitres sur les divers ports de l'HC12, ainsi qu'au chapitre : « vecteurs d'interruptions, développement et application définitive ».

3.2.8. Format d'instruction, temps d'exécution

3.2.8.1. Etude par les tableaux fournis par le constructeur

Toute une série de tableaux fournis par le constructeur renseignent sur :

- Le codage complet de l'instruction (fonction de l'instruction évidemment, mais aussi du mode d'adressage)
- Le nombre de cycles machine (fréquence F_BUS = fréquence du signal E) correspondant
- Ce qui circule pour chaque cycle sur le bus.

Nous ne donnons pas tous les tableaux, mais seulement de quelques instructions de base.

Sans posséder une architecture Hardward, le HC12 est structuré pour une vitesse d'exécution maximale.

Bien qu'ayant les mêmes registres et accumulateurs A et B de 8 bits que l'HC11, L'HC12 a tout de même une structure d'un 16 bits si on observe ses bus donnée : bus internes de données de 16 bits, et en mode étendu, possibilité de câblage d'un bus donnée également de 16 bits.

L'HC12 **lit ainsi son code objet 16 bits par 16 bits**, et son compteur programme s'auto incrémente de 2. Il se débrouille intérieurement pour distinguer les codes opérateurs des instructions (sur 1 ou 2 octets) des opérandes.

Abréviations utilisées : PC Compteurs programme
RTN Adresse retour après saut à un sous programme
Pour les codes machine : **hh ll** Octets hauts et bas d'une adresse
ee ff Octets hauts et bas d'un Offset

Types de cycle demandant chacun un **seul cycle d'horloge E** :

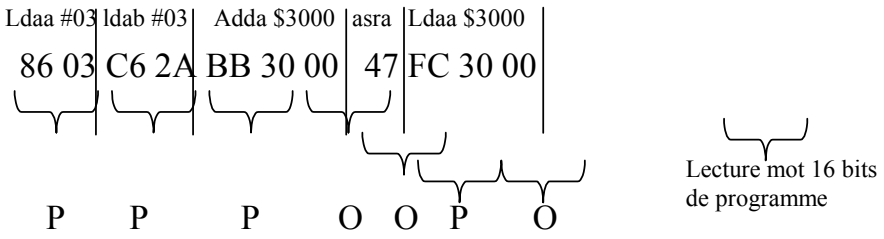
f	Cycle interne sans accès bus
r et **w**	Accès donnée 8 bits
R et **W**	Accès donnée 16 bits
P	Lecture de 2 octets de code objet, auto incrémention du PC de 2.
O	Lecture ou non de code objet, si un seul octet utile. Recadrage.

Remarque : les cycles d'accès à 16 bits demandent évidemment 2 cycles de E pour les accès en mode étendu en câblage 8 bits, ce qui n'est pas le cas normal !

Si l'instruction N à un nombre d'octet pair, le compteur programme tombe automatiquement sur l'instruction suivante. Cycle noté **P**.

Si l'instruction N à un nombre d'octets impair, un **cycle O** de lecture 16 bits pour un seul octet utile (ou parfois sans lecture du tout car il peut avoir déjà lu cet octet lors de l'instruction précédente) s'effectue avec auto incrémentation de 1 seulement de son compteur programme.

Exemples de cycles de lecture du code objet : Cycles P et O



Sur un simple microcontrôleur 8 bits, il est évident que tous les accès 16 bits demandent 2 cycles machine. On constate donc la rapidité plus importante de l'HC12 sur l'HC11 à F_BUS identique : rien que pour lire ce code (sans compter l'exécution où l'on gagne également) il faudrait 11 cycles pour l'HC11, l'HC12 se contente de 7.

D'autre part la F_BUS de l'HC12 peut monter si on le souhaite à 40 MHz, avec le même quartz de 8 MHz (PLL interne) !

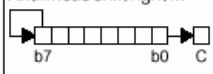
Pour bien comprendre, il est intéressant de trouver le **nombre minimum théorique de cycles nécessaires pour une instruction**, et comparer au ***nombre réel*** fourni par le ***constructeur***. Le processeur parfois n'optimise pas à 100% et perd quelques cycles.

L'étude du nombre de cycle minimum théorique nécessaire, est basée sur le bon sens, et est valable un peu pour tous microcontrôleur classique non Harward, et à priori dans les lectures, sans mélanger dans un même cycle code opérateur et octets de données ou d'adresses.

On compare ensuite le cas particulier de l'HC12.

Etude sur quelques instructions. On remarquera que l'ordre indiqué pour les accès n'est pas l'ordre réel, bizarre …. !

ASLA

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
ASR *opr16a* ASR *oprx0_xysppc* ASR *oprx9,xysppc* ASR *oprx16,xysppc* ASR [D,*xysppc*] ASR [*oprx16,xysppc*] ASRA ASRB	Arithmetic shift right M  Arithmetic shift right A Arithmetic shift right B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	77 hh ll 67 xb 67 xb ff 67 xb ee ff 67 xb 67 xb ee ff 47 57	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	– – – – Δ Δ Δ Δ

Durée d'exécution minimale théorique :

Opérations élémentaires	Nombre d'octets	Nombre de cycles micro 8 bits	Nombre de cycles Micro 16 bits
Lecture Code opérateur	1	1	1
		Total : 1	Total : 1

Durée d'exécution réel pour HC12 :

Opérations élémentaires pour : ASLA : 48	Abréviations **O**	Nombre de cycles micro 16 bits HC12
Lecture Code opérateur $48	O	1
		Total : 1

ADDA valeur8 adressage étendu (EXT), adresse 16 bits (opr16a), donnée 8 bits

Code objet 3 octets : **BB hh ll** BB code opérateur

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
ADCB #*opr8i* ADCB *opr8a* ADCB *opr16a* ADCB *oprx0_xysppc* ADCB *oprx9,xysppc* ADCB *oprx16,xysppc* ADCB [D,*xysppc*] ADCB [*oprx16,xysppc*]	Add with carry to B; (B)+(M)+C⇒B or (B)+imm+C⇒B	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C9 ii D9 dd F9 hh ll E9 xb E9 xb ff E9 xb ee ff E9 xb E9 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – Δ – Δ Δ Δ Δ

ADDA #*opr8i*
ADDA *opr8a*
ADDA *opr16a*
ADDA *oprx0_xysppc*
ADDA *oprx9,xysppc*
ADDA *oprx16,xysppc*
ADDA [D,*xysppc*]
ADDA [*oprx16,xysppc*]
	Add to A; (A)+(M)⇒A
	or (A)+imm⇒A
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			8B ii
			9B dd
			BB hh ll
			AB xb
			AB xb ff
			AB xb ee ff
			AB xb
			AB xb ee ff
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					– – Δ – Δ Δ Δ Δ

Durée d'exécution minimale théorique :

Opérations élémentaires	Nombre d'octets	Nombre de cycles micro 8 bits	Nombre de cycles Micro 16 bits
Lecture Code opérateur	1	1	1
Lire adresse opérande	2	2	1
Lire opérande et addition à A	1	1	1
		Total : 4	Total : 3

Durée d'exécution pour HC12 :

Opérations élémentaires pour : ADDA (EXT) : BB hh ll	Abréviations **rPO**	Nombre de cycles micro 16 bits HC12
Lecture Code opérateur FC et adresse opérande hh	P	1
Lire adresse opérande ll et recadrage PC	O	1
Lire opérande et addition à A	r	1
		Total : 3

- **Ldd valeur16** adressage étendu (EXT), adresse 16 bits (opr16a), donnée 16 bits
- **Ldx, Ldy** idem

Code objet 3 octets : **FC hh ll** FC code opérateur

LDAA *#opr8i*
LDAA *opr8a*
LDAA *opr16a*
LDAA *oprx0_xysppc*
LDAA *oprx9,xysppc*
LDAA *oprx16,xysppc*
LDAA [D,*xysppc*]
LDAA [*oprx16,xysppc*]
	Load A
	(M)⇒A
	or imm⇒A
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			86 ii
			96 dd
			B6 hh ll
			A6 xb
			A6 xb ff
			A6 xb ee ff
			A6 xb
			A6 xb ee ff
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					– – – – Δ Δ 0 –

LDD *#opr16i*
LDD *opr8a*
LDD *opr16a*
LDD *oprx0_xysppc*
LDD *oprx9,xysppc*
LDD *oprx16,xysppc*
LDD [D,*xysppc*]
LDD [*oprx16,xysppc*]
	Load D
	(M:M+1)⇒A:B
	or imm⇒A:B
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			CC jj kk
			DC dd
			FC hh ll
			EC xb
			EC xb ff
			EC xb ee ff
			EC xb
			EC xb ee ff
				PO
				RPf
				RPO
				RPf
				RPO
				fRPP
				fIfRPf
				fIPRPf
					– – – – Δ Δ 0 –

Durée d'exécution minimale théorique :

Opérations élémentaires pour : LDD (EXT) : FC hh ll	Nombre d'octets	Nombre de cycles micro 8 bits	Nombre de cycles Micro 16 bits
Lecture Code opérateur : FC	1	1	1
Lire adresse opérande : hh ll	2	2	1
Lire opérande 16 bits et addition à D	2	2	1
		Total : 5	Total : 3

Durée d'exécution pour HC12 :

Opérations élémentaires pour : LDD (EXT) : FC hh ll	Abréviations **RPO**	Nombre de cycles micro 16 bits HC12
Lecture Code opérateur FC et adresse opérande hh	P	1
Lire adresse opérande ll et recadrage	O	1
Lire opérande 16 bits et addition à A	R	1
		Total : 3 !

- **DEX, DEY, INX, INY, et DECA,DECB,INCA,INCB**

DEX	Decrement X; (X)–1⇒X	INH	09	O
DEY	Decrement Y; (Y)–1⇒Y	INH	03	O

LDD 2,x+

Charger dans D le mot de 16 bits pointé par X, auto incrémentation de 2.
Code objet 2 octets seulement: **EC 31**

Durée d'exécution minimale théorique :

Opérations élémentaires pour : LDD 2,X+ EC 31	Nombre d'octets	Nombre de cycles micro 8 bits	Nombre de cycles Micro 16 bits
Lecture Code opérateur et valeur d'auto incrémentation : EC 31	2	2	1
Lire opérande 16 bits, chargement dans D, auto incrémentation X de 2.	2	2 ou 4 si l'auto incrémentation n'existe pas (instructions en plus)	1 ou 2 si l'auto incrémentation n'existe pas
		Total : 4 ou 6	**Total : 2 ou 3**

Durée d'exécution pour HC12 :

Opérations élémentaires pour : LDD 2,X+ EC 31	Abréviations **RPf**	Nombre de cycles micro 16 bits HC12
Lecture Code opérateur et valeur d'auto incrémentation : EC 31	P	1
Lire opérande 16 bits, chargement dans D, auto incrémentation X de 2.	R	1
Travail interne, **non optimisé au max** !	f	1
		Total : 3

- **JSR sousprogramme**

Code objet 3 octets : **16 hh ll** 16 code opérateur

JSR *opr8a* JSR *opr16a* JSR *oprx0_xysppc* JSR *oprx9,xysppc* JSR *oprx16,xysppc* JSR [D,*xysppc*] JSR [*oprx16,xysppc*]	Jump to subroutine (SP)–2⇒SP RTN_H:RTN_L⇒M_{SP}:M_{SP+1} Subroutine address⇒PC	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	17 dd 16 hh ll 15 xb 15 xb ff 15 xb ee ff 15 xb 15 xb ee ff	SPPP SPPP PPPS PPPS fPPPS fIfPPPS fIfPPPS	- - - - - - - -

Durée d'exécution minimale théorique :

Opérations élémentaires pour : JSR (EXT) : 16 hh ll	Nombre d'octets	Nombre de cycles micro 8 bits	Nombre de cycles Micro 16 bits
Lecture Code opérateur : 16	1	1	1
Lire adresse opérande : hh ll	2	2	1
Empiler ancien compteur programme	2	2	1
Charger nouveau compteur programme	2	0 (déjà lu)	0
		Total : 5	Total : 3

Durée d'exécution pour HC12 :

Opérations élémentaires pour : JSR (EXT) : 16 hh ll	Abréviations **SPPP**	Nombre de cycles micro 16 bits HC12
Lecture Code opérateur 16 et adresse opérande hh	P	1
Lire adresse opérande ll (noté P, O de recadrage sans objet)	P	1
Empilement compteur programme pour retour	S	1
Chargement nouveau PC	P ?	1
		Total : 4

Branchements

BEQ *rel8*	Branch if equal; if Z=1, then (PC)+2+rel⇒PC	REL	27 rr	PPP (branch) P (no branch)	- - - - - - - -
BNE *rel8*	Branch if not equal to 0; if Z=0, then (PC)+2+rel⇒PC	REL	26 rr	PPP (branch) P (no branch)	- - - - - - - -

Attention : on remarque 3 cycles si branchement (boucle d'attente). Un seul cycle si on saute en fin de boucle.

Transferts direct mémoire ← → mémoire : movb #3c,$3000

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C

MOVB #*oprx8*, *opr16a*
MOVB #*oprx8i*, *oprx0_xysppc*
MOVB *opr16a*, *opr16a*
MOVB *opr16a*, *oprx0_xysppc*
MOVB *oprx0_xysppc*, *opr16a*
MOVB *oprx0_xysppc*, *oprx0_xysppc*
	Move byte
	Memory-to-memory 8-bit byte-move
	(M_1)⇒M_2
	First operand specifies byte to move
		IMM-EXT
		IMM-IDX
		EXT-EXT
		EXT-IDX
		IDX-EXT
		IDX-IDX
			18 0B ii hh ll
			18 08 xb ii
			18 0C hh ll hh ll
			18 09 xb hh ll
			18 0D xb hh ll
			18 0A xb xb
				OPwP
				OPwO
				OrPwPO
				OPrPw
				OrPwP
				OrPwO
					- - - - - - - -

MOVW #*oprx16*, *opr16a*
MOVW #*oprx16i*, *oprx0_xysppc*
MOVW *opr16a*, *opr16a*
MOVW *opr16a*, *oprx0_xysppc*
MOVW *oprx0_xysppc*, *opr16a*
MOVW *oprx0_xysppc*, *oprx0_xysppc*
	Move word
	Memory-to-memory 16-bit word-move
	(M_1:M_1+1)⇒M_2:M_2+1
	First operand specifies word to move
		IMM-EXT
		IMM-IDX
		EXT-EXT
		EXT-IDX
		IDX-EXT
		IDX-IDX
			18 03 jj kk hh ll
			18 00 xb jj kk
			18 04 hh ll hh ll
			18 01 xb hh ll
			18 05 xb hh ll
			18 02 xb xb
				OPWPO
				OPPW
				ORPWPO
				OPRPW
				ORPWP
				ORPWO
					- - - - - - - -

Durée d'exécution minimale théorique :

Opérations élémentaires pour : Movb #3c,$3000: 180B 3C 3000	Nombre d'octets	Nombre de cycles micro 8 bits	Nombre de cycles Micro 16 bits
Lecture Code opérateur : 180B	2	2	1
Lire opérandes : 3C hh ll	3	3	2
Ecrire (ici 3C) en mémoire	1	1	1
		Total : 6	Total : 4

Durée d'exécution pour HC12 :

Opérations élémentaires pour : Movb #3c,$3000: 180B 3C 3000	Abréviations **OPwP**	Nombre de cycles micro 16 bits HC12
Lecture Codes opérateurs 180B	P	1
Lire donnée immédiate, 8 bits utiles	O	1
Lire adresse d'écriture	P	1
Ecriture octet	w	1
		Total : 4

3.2.8.2. Mesure de temps d'exécution

On peut parfois **mesurer** des temps d'exécution sur **l'outil de développement** utilisé (Debugger ou simulateur), ligne par ligne, entre deux ponts d'arrêts, par fonctions …

Sinon, **en temps réel**, on mesure des temps d'exécution en faisant évoluer une ligne quelconque d'un port parallèle : passage à 1 en début de zone étudiée, à 0 ensuite, et on **mesure à l'oscilloscope**. Prévoir pour cela un programme répétitif si possible, ou utiliser la pause pour figer le signal sur les oscilloscopes numériques modernes.

3.3. Programme de démonstration en assembleur HC12 : 4_leds

Le problème des vecteurs d'interruptions seront vus ultérieurement. Nous étudierons par contre le problème des variables initialisées en vue d'une mise en ROM de l'application.

3.3.1. Plan mémoire du microcontrôleur utilisé

→plan mémoire HC12 et choix pour la mise au point :

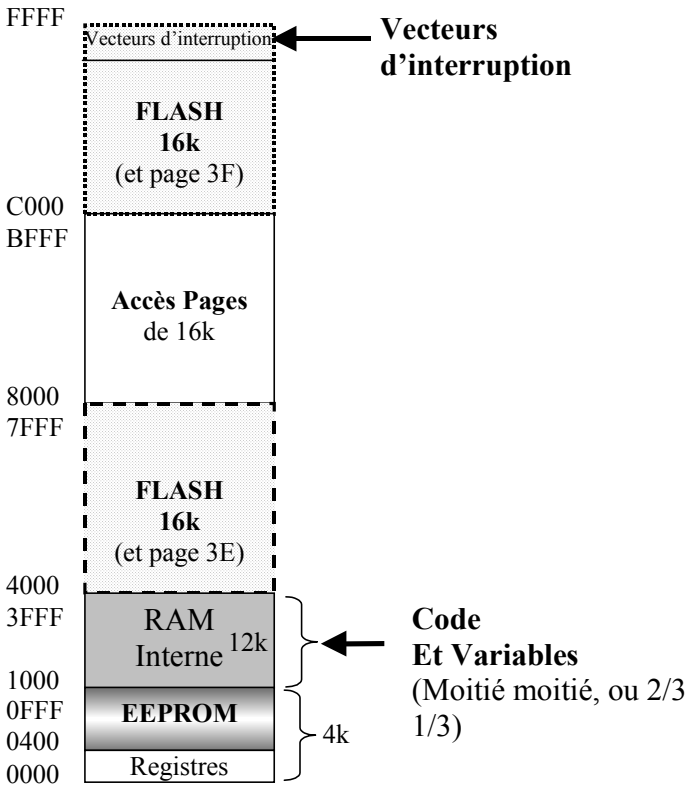
En mode mise au point, tout est placé en RAM.

L'HC12 possède une **RAM interne** de **$1000** à **$3FFF.**

On choisit par exemple:
$1000 à $2FFF : code
$3000 à $3FFF : données

L'HC12 travaillant toujours en décrémentant son pointeur de pile, la pile sera placée en haut de la RAM partie donnée, donc son sommet sera à $3FFF.

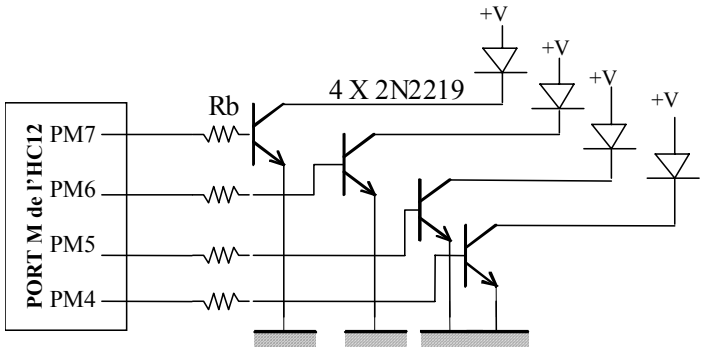
Remarque :, certains systèmes permettent une mise au point en FLASH, mais téléchargement plus long. A éviter si on peut.



3.3.2. Application rudimentaire étudiée en HC12

Sur les 4 bits de fort poids du PORT parallèle M de l'HC12, sont câblées 4 diodes électroluminescentes. Ce port se programme au moyen de deux registres internes à l'HC12 :

PTM adresse $250
DDRM adresse $252
(sens, 1 pour sortie).



On étudie un programme allumant à tour de rôle et à une cadence donnée chaque diode LED. Il faut donc envoyer les codes : $10, $20, $40, $80, $10, $20, $40, $80 ….

- La cadence est assurée par un simple sous programme assembleur **tempo**, réalisant un certain nombre de boucles. Ce sous programme n'est pas étudié ici.

Son cahier des charges, pour la cadence du HC12 à 2MHz:

- Valeur de l'attente en millisecondes dans registre X
- Aucun registre modifié

- Les codes sont placés dans un tableau **Tab** de 4 valeurs. La taille du tableau est figée, on utilise un label **Taille** déclaré par une directive EQU en assembleur.

3.3.3. Assembleur non relogeable, un seul fichier

Etudions ici un programme très simple, écrit avec un outil de développement rudimentaire, sans éditeur de lien, travaillant sur un seul fichier et en code absolu (C'est le cas des Kits simples d'initiation aux microcontroleurs).

1) ***Fichier assembleur : 4_leds.s33***
(extension .S33 propre à l'assembleur IAR Systems)
Le code de tempo est forcément dans ce même fichier.

```
        PTM             equ     $250
        DDRM            equ     $252
        Taille          equ     4

                        ORG     $3000           place des variables en absolu
        Tab     fcb     $10,$20,$40,$80
                        ORG     $3FFF     ou $FF     option expliquée plus loin
        Sommet_pile equ *                        option expliquée plus loin

                        ORG     $1000           Code exécutable
        debut   LDS     #sommet_pile            option expliquée plus loin
                        movb    #$F0,DDRM       Plus rapide, en HC11 : LDAA #$F0 et STAA
DDRM                                            4 bits MSB du PORTM en sortie
        encore  ldy     #Tab            pointeur
                        ldab    #taille         compteur
        bou     ldaa    1,y +           Indexé, auto incrémentation de Y
                                                En HC11, il fallait ldaa 0,y et iny !
                        staa    PTM             envoi au moteur
                        ldx     #10     sous programme temporisation de X = 10 millisecondes
                        jsr     tempo
                        decb
                        bne     bou
                        bra     encore

        tempo           …       sous programme de temporisation de X millisecondes
                        …       par Timer ou par boucles, sans interruption, non étudié ici
                        …       Ne modifiant pas les registres
                        rts
```

2) Et le pointeur de pile ?

Le programme l'utilise forcément lors du saut au sous programme tempo (sauvegarde automatique du compteur programme, pour pouvoir revenir ….)

Les systèmes de développements utilisent forcément déjà une zone de 'pile moniteur' (obligatoire pour l'usage de sous programme), et donc un pointeur de pile déjà initialisé. Deux possibilités selon les systèmes :

- On laisser le pointeur de pile dans la zone moniteur, donc rien à ajouter il est déjà initialisé par le moniteur.
- On change cette zone, on doit alors ajouter l'instruction **LDS #sommet_pile** , avec la pile réservée dans une zone RAM de donnée (le HC12 empile par décrémentation).

3) Et les vecteurs d'interruptions ?

L'outil de développement se sert obligatoirement d'interruptions, au minimum l'interruption RESET (qui est en ROM et qui permet le démarrage de la carte sur le Moniteur), et une interruption pour reprendre la main après lancement d'un programme utilisateur. Rien n'est à modifier ici car ce programme utilisateur ne se sert pas d'interruptions.

4) Inconvénients ?

Le code assembleur dépend de la carte d'application (Sections déclarées 'absolues')
Un seul fichier ne permet pas de travailler aisément sur des programmes plus importants !

3.3.4. Ecriture en Sections Relogeables, et plusieurs fichiers avec éditeur de lien

On décide de réaliser maintenant un code situé sur plusieurs fichiers, et entièrement relogeable au niveau de l'éditeur de lien (pile comprise), on utilise donc des « sections » relogeables:
Avantages immédiats : plusieurs fichiers et implantation physique définie uniquement à l'édition de lien.

Section de type CODE	de nom **CODE**	**Code exécutable**
Section de type DATA	de nom **VARIABLES**	**Variables**
Section de type DATA	de nom **STACK**	**Pile Utilisateur**

On utilisera les directive RSEG (Section relogeable) ASEG (Section absolue) de l'outil IAR Systems, et ses notations.
En assembleur seul, les noms de sections CODE, VARIABLES, STACK sont quelconques et choisis par le programmeur.

3.3.4.1. Les trois fichiers assembleurs :

1) Fichier ***initialisations.s33***

Ce fichier peut contenir :
Dans la section STACK :La déclaration d'un label sommet_pile sur adresse courante.
Dans la section CODE :
L'initialisation du pointeur de pile sur ce label.
Le branchement vers le début de notre programme.

Dans UNE SECTION ABSOLUE le vecteur Reset en $FFFE sur le label correct (Même si ceci semble inutile pour la mise au point, cela serait nécessaire pour une mise en mémoire morte future, et souvent la mise au point Debug s'initialise en voyant ce vecteur initialisé correctement).

```
                Extern  debut           symbole connu ailleurs (importation)
                RSEG    STACK
Sommet_pile     equ     *
                RSEG    CODE
Ini             lds     #sommet_pile
                Jmp     debut   debut du programme proprement dit, dans un autre fichier.
                ASEG
```

```
                ORG   $FFFE
                Fdb     ini     utile pour la mise en ROM et parfois pour le Debug,
                                (dépend de l'outil de développement)
                END             ( IAR System : fin de texte à assembler)
```

2) Fichier ***sousprog.s33***
Dans la section CODE : entre autres notre sous programme tempo.

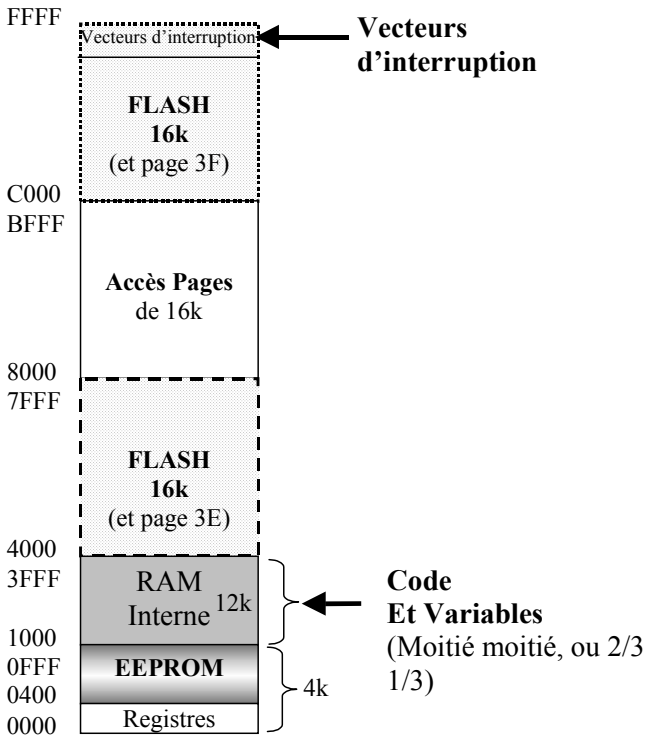
```
                Public tempo
                RSEG CODE
tempo …         sous programme de temporisation de X millisecondes
                …       par Timer ou par boucles, sans interruption, non étudié ici
                …       Ne modifiant pas les registres !
                rts
```

3) Fichier **principal.s33**
Dans la section VARIABLES le tableau de 4 valeurs
Dans la section CODE l'exécutable

```
PTMequ      $250
DDRM        equ     $252
            Extern  tempo       routine dans un autre fichier, ou en librairie.
            Public  debut
Taille      equ     4
            RSEG    VARIABLES
Tab         fcb     $10,$20,$40,$80     données initialisées
            RSEG    CODE
debut       movb    #$F0,DDRM   Plus rapide, en HC11, il fallait écrire LDAA #$F0 et STAA DDRM
                                4 bits de poids faible programmés en sortie
Encore      LDY     #Tab        pointeur Y en début de tableau
            LDAB    #taille     B comme compteur
Bou  LDAA 1,Y+      indéxé, auto incrémentation
            STAA    PTM
            LDX     #200
            JSR     Tempo       tempo de 200ms valeur en millisecondes dans Y
            DECB
            BNE     Bou
            BRA     encore
```

3.3.4.2. Edition de lien en mode mise au point

➢ ***Carte utilisée et plan mémoire***



➢ ***Fichier de commande possible pour l'éditeur de lien, en mise au point***

Voici un exemple simple de fichier de commande pour cette carte (Outils IAR Systems)

On retrouve dans ce fichier les noms de section déjà choisis par l'utilisateur dans les programmes assembleurs (CODE, VARIABLES, STACK …), et on remarque les sortes de directives les dirigeant vers telle ou telle zone mémoire.

```
-!  Type de microprocesseur  -!
-c68hc12
-!              SECTIONS de type  CODE  de 1000 à 2FFF        -!
-Z(CODE)CODE =1000-2FFF

-!              SECTIONS de type  DATA   de 3000 à 3FFF        -!
-Z(DATA)VARIABLES=3000-3EFF     // libre ensuite 256 octets de pile
-Z(DATA) PILE=3FFF
-!     Pile de 3F00 à 3FFF   adresses RAM interne,    PILE = Sommet de la pile      -!
```

-Z(CODE) signifie Sections de type CODE
-Z(DATA) signifie Sections de type DATA

3.3.5. Le problème des variables initialisées en vue de la mise en ROM

Les valeurs 10,20,40,80 du tableau précédent Tab sont actuellement dans une zone de variables et sont téléchargées à chaque chargement du code complet sur la carte.

Ces valeurs doivent être évidemment en ROM pour l'application définitive, et d'ailleurs un « Warning » lors de l'édition de lien nous indiquerait que pour l'instant cette application n'est pas mettable en ROM 'non promable '

3.3.5.1. Travail avec des constantes

Si ces valeurs ne sont jamais modifiée par le programme. Ce sont alors des constantes !

- ***Modification du programme***, tab et ses valeurs sont placées dans une section de nom CONSTANTES.

Dans principal.s33 :

```
                    RSEG   CONSTANTES
Tab                 fcb    $10,$20,$40,$80
```

- ***Modification du fichier de link*** :

```
-! Type de microprocesseur -!
-c68hc12
-!          SECTIONS de type CODE de 1000 à 2FFF      en RAM en développement -!
-Z(CODE)CODE,CONSTANTES = 1000-2FFF

-!          SECTIONS de type DATA  de 3000 à 3EFF      -!
-Z(DATA)VARIABLES=3000-3FFF
-Z(DATA)PILE=3FFF
-!          PILE = Sommet de la pile     -!
```

3.3.5.2. Travail avec des variables initialisées

Nécessaire si plus tard on doit par programme modifier ces valeurs.

- Il faut modifier principal.s33 en réservant de la place pour tab, et en déclarant dans la section CONSTANTES les valeurs initiales de tab dans un tableau déclaré « public » ini_tab.

Dans principal.s07 :

```
                    Public tab, initab
                    RSEG   CONSTANTES
initab              fcb    $10,$20,$40,$80
                    RSEG   VARIABLES
tab                 rmb    4
```

- Il faut compléter initialisation.s07 par des instructions de recopie de ini_tab vers tab (et directives extern). Nouveau fichier :

```
        Extern  debut,tab, ini_tab          symboles connu ailleurs (importation)
                RSEG   STACK
Sommet_pile  equ  *
                RSEG   CODE
Ini             lds    #sommet_pile
                Jsr    inivar
                Jmp    debut
Inivar  ldaa    ini_tab
                staa   tab
                ldaa   ini_tab+1
                staa   tab+1                (ou boucle de recopie)
                ldaa   ini_tab+2
                staa   tab+2
                ldaa   ini_tab+3
                staa   tab+3
                rts
                ASEG
                ORG    $FFFE
 Fdb            ini
                END
```

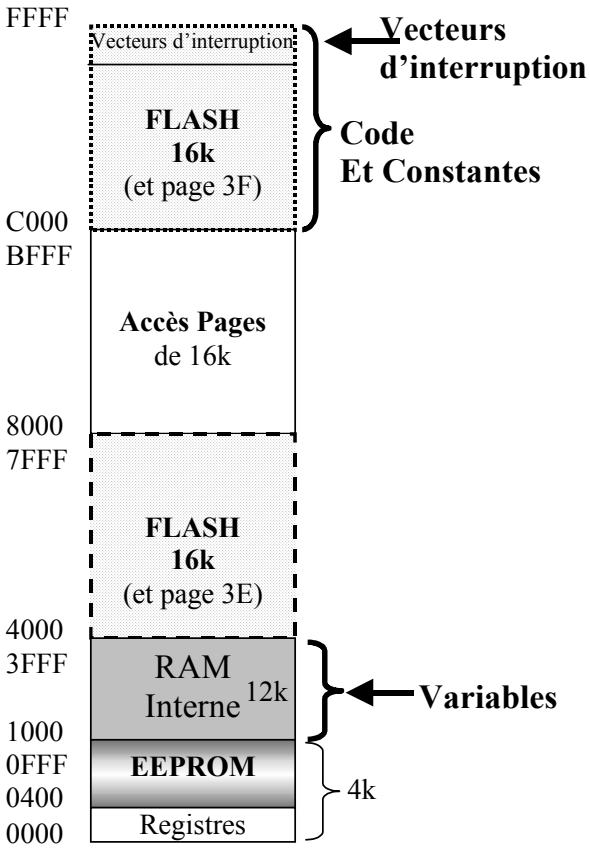
Remarque : un langage performant tel que le C gère tout ceci automatiquement. !

3.3.5.3. Fichier de lien pour la mise en ROM

Il doit correspondre à la carte ci-contre.

Le code exécutable est placé en mémoire morte (FLASH ici)

Les variables sont en RAM




```
-!  Type de microprocesseur  -!
-c68hc12
-!              SECTIONS de type  CODE  de C000 à CFFF       en FLASH application finale  -!
-Z(CODE)CODE,CONSTANTES = C000-CFFF

-!              SECTIONS de type  DATA   de 3000 à 3EFF        -!
-Z(DATA)VARIABLES=3000-3FFF
-Z(DATA)PILE=3FFF
-!          PILE = Sommet de la pile      -!
```

3.3.6. vecteurs d'interruption autres que le simple Reset ?

Nous verrons ceci ultérieurement lors de l'étude des vecteurs d'interruption pour le développement et pour l'application finale, ou en effet un autre problème se posera.

Il faut pouvoir, en phase de mise au point, modifier ces vecteurs. Si ils sont en FLASH, pas trop de problème, on peut les modifier lors des téléchargement. Si ils sont en ROM ou PROM ce ne sera pas possible. Il faudra ruser.

3.4. Astuces de gestion de variables indicées en assembleur

On cherchera surtout dans ce chapitre à optimiser en assembleur la vitesse d'exécution dans les calculs répétitifs.

Des astuces de travail au moyen d'un seul pointeur sont décrites, ce qui permet l'emploi de microprocesseurs simples.

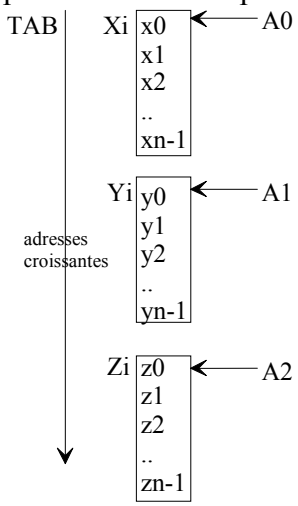
Ces techniques peuvent servir lors de la programmation en assembleur de fonction C en vue d'optimisation de la vitesse.
Le but étant de **garder les pointeurs dans des registres**.

3.4.1. Variables à un seul indice

Soit 3 variables indicées de 32 bits xi, yi, zi (avec i de 0 à n-1).
Une table de 4.n octets définit chaque variable.
On cherche à calculer dans une boucle sur i : **zi = f(xi, yi)**

➢ ***Disposition 1***

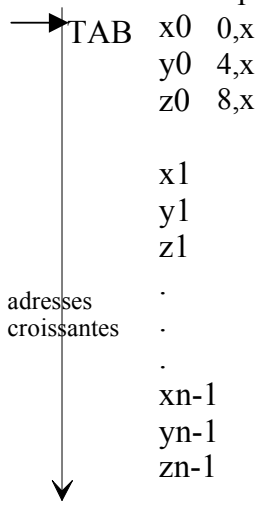
Tous les Xi, puis tous les Yi à partir de TAB



Il faut trois pointeurs (ici A0, A1, A1) si on veut travailler sur les trois variables d'une façon répétitive.
L'HC11 (ou HC13) n'a que des pointeurs X et Y. Il en faut donc un en mémoire.
En 68000 pas de problème.

➢ ***Disposition 2 : plus astucieux***

xi,yi,zi sont consécutives à partir de TAB.



Dans une boucle, un seul pointeur suffit en effet pour accéder à x, y et z (par les décalages 0,4,8)
X +12 donne l'indice suivant.
Pratique pour des processeurs a peu de pointeurs (comme l'HC11).

3.4.2. Variables à plusieurs indices, calcul matriciel

Voyons le cas général manipulant des nombres de différentes tailles, et optons pour l'utilisation **d'un seul pointeur** (pour des microprocesseurs plus simples, HC11 par exemple c'est nécessaire) . Soit une structure figée à taille fixe, tel que le produit matriciel suivant:

a,b variables 4 octets
ki constantes 2 octets
xi 4 octets (après arrondi)

$$\begin{bmatrix} x1 \\ x2 \\ x3 \end{bmatrix} = \begin{bmatrix} k1 & k2 \\ k3 & k4 \\ k5 & k6 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

On utilisera alors un sous programme pour réaliser $xi = k_j.a + k_k.b$

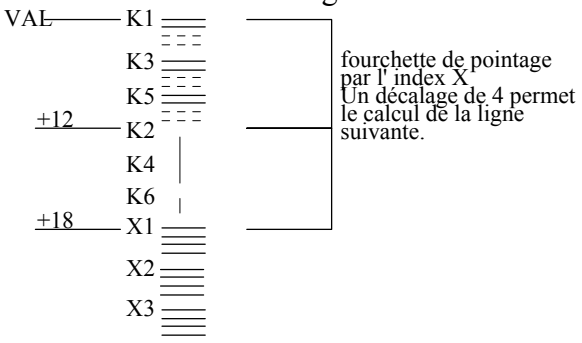
La disposition suivante astucieuse permet de travailler avec un seul registre X

Les sorties se faisant sur 4 octets, on réservera 4 octets pour les ki, deux n'étant pas utilisés (traits pointillés).

On travaille sur « fourchette » dessinée.

On décale la fourchette de 4 (X=X+4) pour passer à l'indice suivant. Simple !

Les ki sont obligatoirement des variables initialisées (et non des constantes !) (leur valeur initiale pourra être en ROM), en effet elles doivent être en RAM à coté des xi.



4. PROGRAMMATION EN C, INTRODUCTION

4.1. Avantage :

Le C permet de développer de longs programmes d'une façon bien moins laborieuse qu'en assembleur, en particulier :

Presque **tous types d'opérandes,** et **fonctions mathématiques.**
Gestion très **simple** des **interruptions**.

Quelques difficultés cependant subsistent !

- **Le langage C ne sait pas travailler à priori sur des nombres en virgule fixe quelconques**, il raisonne soit en **entier**, ou en **flottant**.

- **Si on travaille en virgule fixe seulement par soucis de vitesse d'exécution ou de taille de code, o**n retrouvera, à des variantes près, les problèmes inhérents aux opérations virgule fixe :
  - **Cadrage** des opérandes
  - Gestion de la **dynamique** et de **la précision**.
  - **Travailler sur des entiers** tout en **raisonnant en fractionnaires**.

- Dans le cas de travail en **flottant**, attention aux **conversions** en **virgule fixe**, nécessaires au niveau des entrées sorties de valeurs numériques (sur Clavier, afficheurs, CNA, CAN..)

- Certaines **instructions spécifiques** ne peuvent être écrites en langage C, on doit toujours incorporer quelques lignes d'assembleur.

Outils logiciels:

Programmation: Compilateur C (avec optimiseurs de différents niveaux)
Assembleur
Editeur de lien

Mise au point -**Emulateur** avec sonde se mettant à la place de processeur d'une carte. Performant, mise au point sans ralentissement du processeur. Assez onéreux.

-**Simulateur** (tournant sans carte DSP), très lent, et pas d'entrée sortie réelles.

-Système avec ordinateur et **téléchargement** dans une **carte** avec **moniteur**. Bon compromis prix facilité de programmation de mise au pont et de mise en ROM par la suite.

-Carte **sans Moniteur** pour certains types de microcontrôleurs. Pratique mais peut nécessiter une sonde spéciale de liaison assez onéreuse.

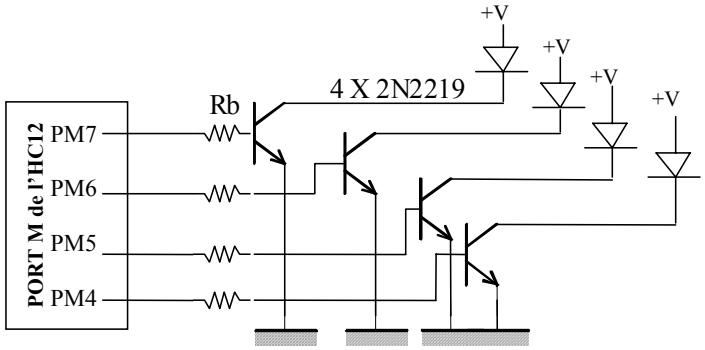
4.2. Programme de démonstration en C : 4_leds_c

Note : Les notations et fichiers générés correspondent au compilateur **IAR Systems.**
Le problème des **vecteurs d'interruption** sera vu **ultérieurement**.

4.2.1. Application étudiée

Il s'agit exactement du même exemple de démonstration que celui étudié précédemment en assembleur : sur les 4 bits de forts poids du PORT parallèle M de l'HC12, sont câblées 4 diodes électroluminescentes.

Le programme doit allumant à tour de rôle et à une cadence donnée chaque diode LED. Il faut donc envoyer les codes : $10, $20, $40, $80, $10, $20, $40, $80 …



PTM adresse $250
DDRM adresse $252 (sens, 1 pour sortie).

- La cadence est assurée cette fois ci par une fonction :

void **tempo(unsigned int millisecondes) ;**

Elle a été forcément écrite en assembleur, pour maîtriser les timings, et est spécifique de la carte (cette fonction sera étudiée ultérieurement dans le chapitre sur la génération d'intervalle de temps). Elle est donc situés dans un fichier séparé.

- Les codes sont placés dans un tableau **Tab** de 4 valeurs.
- Si vous désirez comprendre dès à présent les directives #define déclarant les registres, vous pouvez vous reporter au chapitre traitant du C pour microcontrôleur, mais ce n'est pas obligatoire pour l'instant.

Sachez seulement que pour écrire dans un registre, PTM par exemple, on écrit alors simplement : **PTM = valeur ;** Les labels des ports s'utilisent comme des noms de variables.

4.2.2. Programme en C : fichier 4_leds_c.c

```
#define   PTM      *(char*)0x250
#define   DDRM     *(char*)0x252
#define taille 4
extern void tempo(int);
char tab[] = {0x10,0x20,0x40,0x80};
void main(void)
{
char k;                 // Variable Locale
// valid_it();      /*  cli si necessaire reprendre la main, pas avec BDM de l'HC12  */
DDRM = 0xF0;
while(1)          // Boucle sans fin de l'application
{
for(k=0; k < 4;k++){        PTM = tab[k];
                            tempo(250);
                    }
}
}
```

4.2.3. Fichier de projet .prj

Ce fichier (Syntaxe propre à chaque compilateur et non lisible clairement), contient en fait tous les renseignements sur le travail en cours :

- Options de compilation et options matériel de Debug
- Chemin des librairies, des include … du compilateur
- Fichiers (C et assembleurs) à prendre en compte dans le projet
- Nom du fichier d'édition de lien
- Etc …..

→ On a dans cet exemple au départ seulement **deux fichiers**: **c.c**

Routines_mini_c.s33

4.2.4. Listing assembleur généré par le compilateur seul: 4_leds_c.lst en provenance de 4_leds_c.c

Le C génère automatiquement du code dans des noms de sections prédéfinies.

Citons juste ici les principales sections équivalentes à celles que nous avons utilisées en assembleur précédemment :

Section de type :	Nom précédent, en assembleur :	Nom attribué par le C
CODE	CODE	CODE et RCODE
CODE	CONSTANTES	CDATA1
CODE	INTVECT	INTVECT
DATA	CSTACK (la pile)	CSTACK (la pile)
DATA	VARIABLES	IDATA1

Examinons le fichier (fichier texte en caractères ASCII) de listing **4_leds_c.lst** dont une grande partie est fournie ci dessous :

- Dans la section CODE (de type CODE), on voit chaque ligne de C, et le développement assembleur correspondant (instructions exécutables). Ce code est peu optimisé par rapport à ce que nous avons écrit en assembleur, mais ici les options d'optimisation ne figurent pas forcément ni les astuces d'écriture en C permettant un code plus court.

- On voit également les sections IDATA1 (de type DATA) et CDATA1 (de type CODE) permettant de déclarer le tableau avec les valeurs initiales.

- Du code doit donc exister quelque part pour initialiser ces 4 variables. On ne le voit pas dans ce fichier, il se trouvera en effet dans le « boot du C » qui doit être exécuté avant le main(). Nous le vérifierons par la suite.

Tout ceci ressemble en fait à ce que nous avons écrit en assembleur seul !

```
      Public  main,tab
      Extern  tempo

RSEG      CODE
  9       void main(void)
  10      {
\ 000000  34            PSHX
\ 000001  36            PSHA
  11      char k;
  13      DDRM = 0xF0;
\ 000002  180BF002      MOVB    #-16,594
\         52
```

tab est **globale→ public**
tempo est dans un **autre fichier**

Section de type CODE

Code exécutable début du main

Création variable locale k (dans la pile)
(A est empilé, mais c'est juste le décalage du pointeur de pile qui crée k)

```
\                ?0001:
 14        while(1)
 15        {
 16        for(k=0; k < 4;k++){  PTM = tab[k];
\  000007  6980              CLR     0,SP
\               ?0004:
\  000009  8604              LDAA    #4
\  00000B  A180              CMPA    0,SP
\  00000D  2FF8              BLE     ?0001
\  00000F  A680              LDAA    0,SP
\  000011  B705              SEX     A,X
\  000013  A6E2....          LDAA    `tab`,X
\  000017  7A0250            STAA    592
 17                    tempo(250);
\  00001A  CC00FA            LDD     #250
\  00001D  16....            JSR     tempo
\  000020  6280              INC     0,SP
 18                          }
 19        }
\  000022  20E5              BRA     ?0004
 20        }
\                RSEG        IDATA1
\               `tab`:
\                       RMB     4

\                       RSEG    CDATA1
\                       FCB     16
\                       FCB     ' '
\                       FCB     '@'
\                       FCB     128
```

Les labels **?0001, ?0004** n'ont pas à être connus à l'extérieur, le C leur attribut un **nom automatiquement**.

Section de type DATA

Réservation de 4 octets pour tab (**variables globales initialisées**).

Section de type CODE

Valeurs initiales (constantes) de tab (mélange de code ASCII et de valeur décimale, bizarre …)

→ Et k ?

Cette variable déclarée en **locale n'existe pas avant exécution**, donc mentionnée nulle part sur ce fichier. Le label k n'existe pas non plus. La variable est crée dans la pile au moment de l'exécution par **PSHA** : A est empilé mais ca ne sert à rien ! c'est juste le décalage du pointeur de pile qui crée la place pour k.

→ Si tab était en statique du main :

On aurait:

```
        RSEG  IDATA1
tab     Rmb   4 (Et sa déclaration Public disparaît)
        RSEG CONST
        FCB 1,2,4,8
```

Déclaré par:

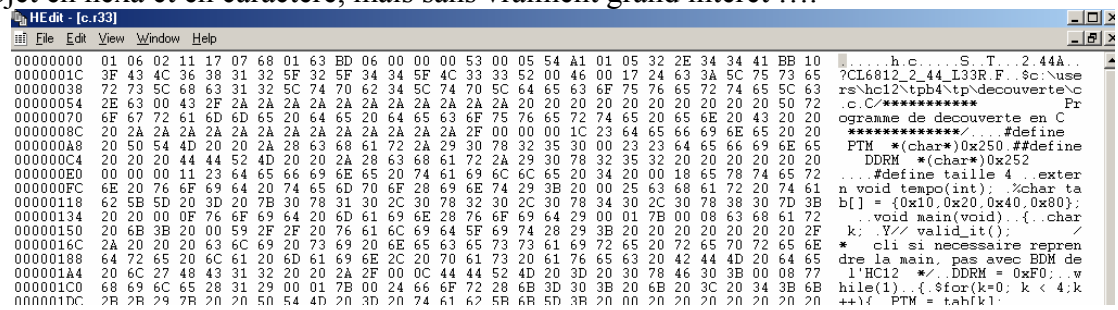
```
Main()
  {
  static char tab[] = {1,2,4,8};
```

Certains compilateurs remplaceraient le nom tab par un nom automatique.

4.2.5. Code relogeable correspondant : 4_leds_c.r33

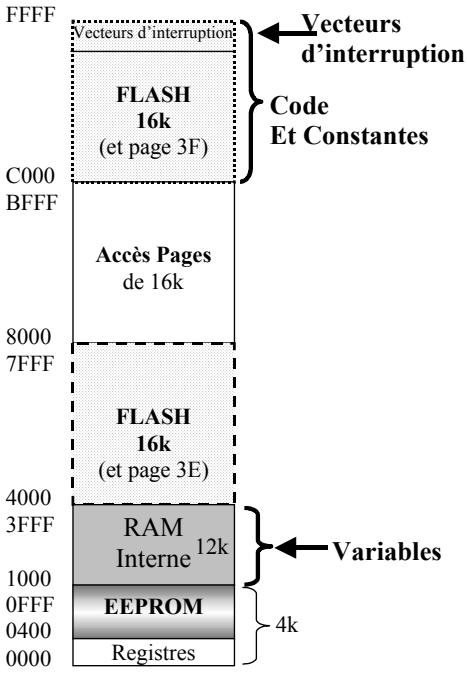
A ce listing, correspond évidemment un objet relogeable C.r33 que le link prendra en compte. (dans /Debug/obj). Un éditeur simple de texte fournit a peu près ceci !

C'est un mélange de quelques renseignements destinés au link (en caractères ASCII) et de code objet. Donc évidemment peu lisible ainsi. Un éditeur hexadécimal permet de lire ce code objet en hexa et en caractère, mais sans vraiment grand intérêt ….



4.2.6. Fichier de commande du link, en mise au point

- Rappel du plan mémoire en mise au point :



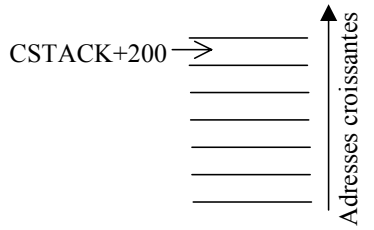
- Fichier de commande d'édition de lien :

```
// 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 interrrupt 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)
cl6812
```

On retrouve -Z(CODE) ou –P(CODE) signifiant : Sections de type CODE
-Z(DATA) signifiant : Sections de type DATA
(Ne pas s'occuper de la distinction entre -Z et -P, elle pourrait s'omettre).

La **librairie du C** est le fichier **cl6812** (situé dans le chemin des librairies du compilateur)

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



Seules les sections en gras sont utilisées pour cet exemple simple. Les autres seront étudiées ultérieurement.

On ne voit pas ici les fichiers pris en compte par l'éditeur de lien (Ceux ci figurent dans le fichier projet.prj)

4.2.7. Le fichier « cartographie » 4_leds_c.map fourni par l'éditeur de lien.

Ce fichier est fourni par l'édition de lien. Il renseigne sur l'emplacement mémoire :

- des **variables globales**,
- des **différentes sections**,
- des **différents sous programmes**.

Ne figurent pas les variables locales (qui sont dynamiques et sont crées dans la pile à l'exécution).

Ne figurent pas forcément les variables statiques (qui n'ont pas à être connues de partout, le C leur donne un autre nom dans le listing assembleur .lst).

Toutes les sections n'ayant pas encore été étudiées, ne pas s'étendre trop dessus, on pourra cependant regarder l'emplacement mémoire :

- Du vrai point d'entrée du programme (Program Entry)
- Du main
- De quelques sous programmes (tempo, __low_level_init , exit : voir plus loin)
- De la variable globale tab.

On voit également les **différents fichiers (**ici **trois fichiers !)** avec tous les relogeables fichiers.r33 correspondant pris en compte par l'éditeur de lien. On parle aussi de « Program Module ».

Au départ, on avait seulement deux fichiers, le troisième provient d'une librairie C, et correspond au **CSTARTUP** nécessaire pour initialiser correctement l'environnement du C.

Nom de Program Module	Provenant du fichier	Relogeable correspondant
c	Notre programme **c.c**	**c.r33**
routines_mini_c	**routines_mini_c.c**	**routines_mini_c.r33**
CSTARTUP	Librairie du **C**	**cl6812.r33**

On voit bien les adresses réellement occupées par les différents éléments.

Tous ces points sont marqués en **gras** dans le texte suivant :

```
Program entry at : 1004  Relocatable, from module : CSTARTUP
          ****************************************
          *            MODULE MAP                *
          ****************************************
 FILE NAME : c:\users\hc12\tpb4\tp\decouverte\Debug\Obj\c.r33
 PROGRAM MODULE, NAME : c

 SEGMENTS IN THE MODULE
 ======================
<CODE,RCODE> 1 (was CODE)
 Relative segment, address: 1063 - 1086 (0x24 bytes), align: 0
 Segment part 0. ROOT.
       ENTRY              ADDRESS       REF BY
       =====              =======       ======
       main               1063          Segment part 9 (CSTARTUP)
         calls direct
       LOCAL              ADDRESS
       =====              =======
       ?0001              106A
       ?0004              106C
  ---------------------------------------------------------------------------
IDATA1
 Relative segment, address: 3000 - 3003 (0x4 bytes), align: 0
 Segment part 14. ROOT.     Intra module refs:   main
       ENTRY              ADDRESS       REF BY
       =====              =======       ======
       tab                3000
  ---------------------------------------------------------------------------
CDATA1
 Relative segment, address: 1000 - 1003 (0x4 bytes), align: 0
 Segment part 15. ROOT.
  ********************************************************

 FILE NAME
 c:\users\hc12\tpb4\tp\decouverte\Debug\Obj\routines_mini_c.r33
 PROGRAM MODULE, NAME : routines_mini_c

 SEGMENTS IN THE MODULE
 ======================
<CODE,RCODE> 1 (was CODE)
 Relative segment, address: 1087 - 109F (0x19 bytes), align: 0
 Segment part 0. ROOT.
       ENTRY              ADDRESS       REF BY
       =====              =======       ======
       tempo              108D          main (c)
       valid_it           1087
       inhib_it           108A
       LOCAL              ADDRESS
       =====              =======
       bou6               1092
       bou7               1095
 ********************************************************
```

@@@@@@@@

Fichier 1
Notre programme C

@@@@@@@@

Fichier 2
Nos fonctions

@@@@@@@@

FILE NAME
C:\USERS\HC12\HC12_TOOLS\WBENCH\6812\LIB**cl6812.r33**
PROGRAM MODULE, NAME : CSTARTUP

SEGMENTS IN THE MODULE
==========

CSTACK
Relative segment, address: 3004, align: 0
Segment part 0. ROOT.

. etc ……..

INTVEC
Common segment, address: **FF80 - FFFF** (0x80 bytes), align: 0
Segment part 10. ROOT.

LIBRARY MODULE, NAME : **lowinit**

SEGMENTS IN THE MODULE
==========

<CODE,RCODE> 1 (was CODE)
Relative segment, address: 10A0 - 10A2 (0x3 bytes), align: 0
Segment part 0. ROOT.

ENTRY	ADDRESS	REF BY
__low_level_init	**10A0**	Segment part 9 (CSTARTUP)

. etc ……..

Fichier 3
Librairies du C
CSTARTUP

@@@@@@@@

****
* SEGMENTS IN ADDRESS ORDER *
****

SEGMENT	START	END	SIZE		
CDATA0	1000				
CDATA1	1000 - 1003		**4**		
CCSTR	1004				
<CODE,RCODE> 1	**1004 - 10A5**		**A2**		
DATA1	3000				
IDATA1	**3000 - 3003**		**4**		
UDATA1	3004				
ECSTR	3004				
WCSTR	3004				
TEMP	3004				
CSTACK	**3004 - 3203**		**200**		
INTVEC	**FF80 - FFFF**		**80**		

Cartographie résumée

@@@@@@@@

****
* END OF CROSS REFERENCE *
****

294 bytes of CODE memory
516 bytes of DATA memory
Errors: none

Taille des deux zones de type CODE et de type DATA

4.2.8. Code objet final, et vrai point d'entrée du programme

4.2.8.1. Code natif (IAR) téléchargeable sur la carte : C .d33

Ce code (que l'on pourrait voir dans **/Debug/exe**) fourni par l'édition de lien, contient le code objet complet qui sera téléchargé, il ressemble quand à l'allure au précédent, avec en plus (option mise dans l'édition de lien) des informations de Debug mais aussi les routines du terminal io si utilisé.
Ce code est compris par le debugger IAR
On peut citer **d'autres codes : Ieee-695 Coff S1S9** qui seraient compris par d'autres Debugger.
On aurait pu supprimer les informations de Debug pour le rendre le plus court possible. Ces informations de toute façon disparaissent dans l'application définitive.
L'examen de ce code présente en fait peu d'intérêt ….., mais le fichier correspondant existe.

4.2.8.2. Le vrai point d'entrée du programme

Il est nommé en début de fichier.map : **Program entry at : 1004**
Donc situé à une adresse différente du main.
→ Ce code avant le main ne se voit clairement sur aucun fichier généré, il se trouve en code objet relogeable dans une librairie du C (CSTARTUP), et est incorporé par l'éditeur de lien. C'est une sorte de « boot » du C. Avec un outil de développement et en mode assembleur, on pourrait examiner pas à pas les toutes premières instruction du programme (**vrai point d'entrée**) et on trouverait en résumé :
« Boot du C »:

```
→ LDS   #……  initialisation du pointeur de pile (obligatoire évidemment)
  JSR    autres initialisations  de bas niveau éventuelle possibles.
         ( ici   __low_level_init)
  JSR    ….     Vers une routine d'initialisation :
                      Recopie des valeurs initiales (1,2,4,8) vers le tableau tab et
                      Autres initialisations éventuellement (environnement du C)
  JSR    main Notre  routine main()
  JSR    exit          Retour au moniteur éventuellement
```

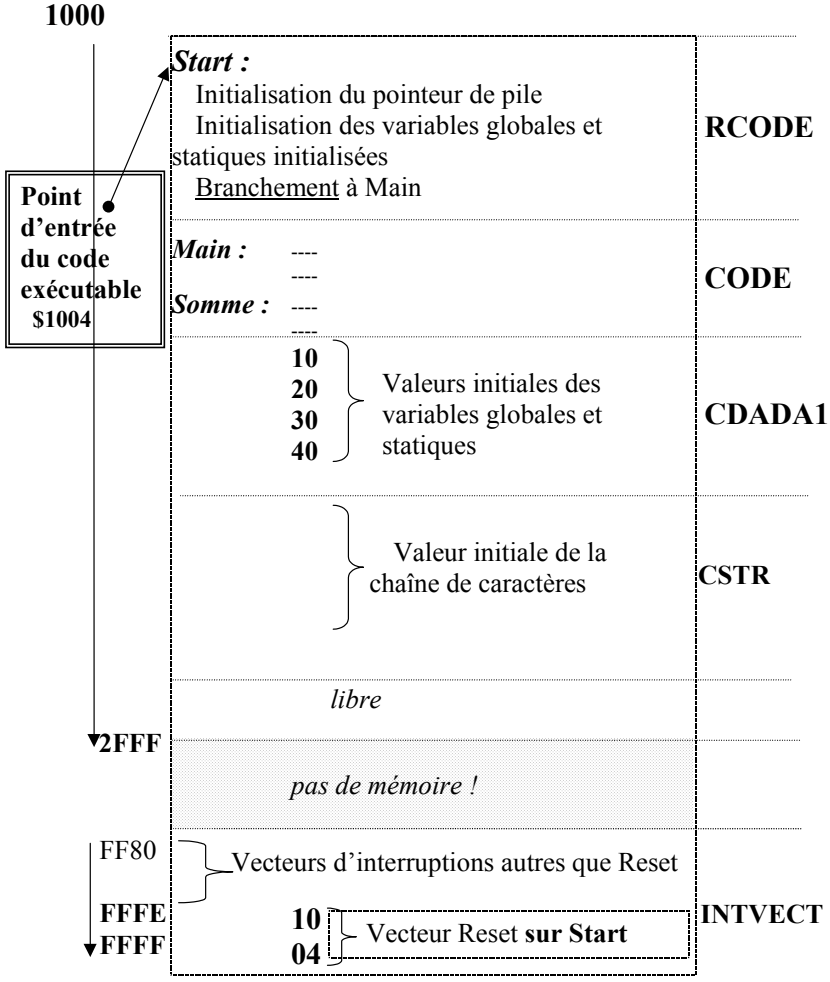
Dans le cas présent, la routine _low_level_init existe mais ne fait rien.

→ On peut consulter la source du CSTARTUP fournie par le constructeur du compilateur. On peut aussi la modifier si nécessaire. On peut aussi modifier seulement la routine _low_level_init qui par défaut ne fait rien pour initialiser par exemples certaines configuration d'un système qui s'effectuerait alors avant le main.

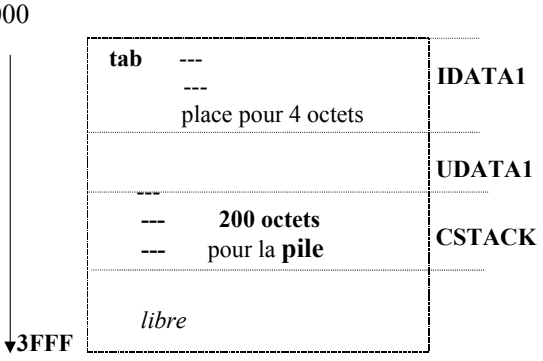
4.2.9. Schéma de la cartographie générée

On peut finalement remarquer une disposition en fait assez similaire à ce que nous avons programmé en assembleur seul ! Voici donc en résumé tous les éléments placés en mémoire :

➢ **Sections de type CODE (En ROM sur l'application définitive)**



➢ **Sections de type DATA (En RAM sur l'application définitive)**



→ ***Remarque :*** La **variable k**, comme nous l'avons vu est **inexistante pour l'instant**. Le programme main() n'est pas en cous d'exécution !

4.2.10. Et la future mise en ROM ?

Aucun problème, une simple modification d'adresse au niveau de l'éditeur de lien et le tour sera joué… Le compilateur a déjà prévu des sections pour le code, les variables, les constantes, les valeurs initiales des variables, etc … La recopie des valeurs initiales des variables que nous devions effectuer en assembleur est faite automatiquement.

5. C POUR MICROCONTROLEURS

5.1. Le langage C « minimum »

On trouvera ici une description minimale du langage C, permettant de programmer des applications microprocesseurs et non de gros logiciels informatiques.

Les **fichiers.c** sont évidemment du code source en C.

Les **fichiers.h** sont des fichiers d'entête, contenant des lignes non exécutables, elles renseignent le compilateur sur les fonctions externes, les variables externes, des définitions de labels …. Il y a des **.h** se rapportant à la **librairie du compilateur** C (Exemple math.h, string.h, stdlib.h ….) et des **.h personnels**.

5.1.1. Structure générale d'un programme

```
#include < >          // pour inclure des fichiers d'entête de la librairie du compilateur
#include " "          // pour inclure des entête personnelles dans le répertoire courant
#define ...../* pour définir des valeurs sous forme d'équivalence de chaînes de caractères,
équivaut à equ en assembleur */
→ Prototypes des fonctions internes
→ extern .....       /* Déclarations de variables ou prototypes de fonctions externes */
→ Déclaration de variables globales

-----------------------------------------------------------------------------------
   main()
   {
   Déclaration de variables locales au main          Programme Principal
   Instructions C
   }
-----------------------------------------------------------------------------------
   .. Fonction1(..., ...,)
   {
   Déclarations de variables locales à la fonction1  Fonction1
   Instructions
   }
-----------------------------------------------------------------------------------
   .. Fonction2(..., ...,)
    {
   Déclarations de variables locales à la fonction2  Fonction2
   Instructions
    }
```

Des fonctions ou variables peuvent être évidemment dans d'autres fichiers. (Rôle de la déclaration extern)

{ } ensemble d'instruction **;** termine toute instruction

/* bloc commentaires ***/** et **//** ligne commentaires

5.1.2. Types de variables

- ***Variables Locales***
  - Déclarées dans une fonction, (main() y compris) et sont invisibles en dehors.
  - N'ont d'existence que durant l'exécution de ces fonctions. Elles sont donc générées et initialisées si c'est le cas, à chaque appel de la fonction.

Mise en garde très importante :

Les **variables locales** n'existant que lors de l'exécution, **aucun contrôle de place disponible dans la pile** n'est effectuée, des disfonctionnements brutaux et aléatoires peuvent alors survenir !

Remède : Estimer si la zone réservée pour la pile (indication dans le fichier d'éditiobn de lien) est suffisante, l'augmenter si nécessaire.

Sinon, déclarer des variables statiques : en effet la place disponible sera alors examinée lors de l'édition de lien avec un message d'erreur éventuel.

- ***Variables globales***
  - Déclarées à l'extérieur du main() et de toute fonction
  - Existent toujours en mémoire. Si initialisées, au démarrage du programme seulement.
  - Visibles de partout, conservent leur valeur en permanence sauf modification par le programme.

N'**utiliser** les variables **globales qu'à bon escient**, pour des raisons éventuellement de rapidité (pas de passage de paramètres aux fonctions), pour des données susceptibles d'être modifiées ou prises en comptes dans des taches d'interruptions, ou pour d'autres raisons plus spécifiques. Un excès de variables globales nuit à la clarté du programme, à la création de fonctions bibliothèques …

Utiliser des **noms bien explicites** : pas de variable k en global !

Et attention à une variable locale qui aurait par hasard le même nom qu'une locale …. des confusions peuvent survenir.

- ***Variables statiques***
  - Déclarées dans une fonction (par **static** type … ;), et donc invisibles en dehors.
  - Existent toujours en mémoire. Si initialisées, ne le sont qu'au démarrage du programme seulement. Conservent donc leur valeur en dehors de la fonction.

  Pratiques pour gérer des drapeaux (par exemple pour initialiser un circuit au premier appel à une fonction), pour incrémenter ou modifier une valeur à chaque appel à une fonction (compteur déclaré et initialisée puis incrémenté dans une fonction) Sans elles, des variables globales seraient nécessaires.

- ***Variables allouées dynamiquement***

Leur déclaration et la réservation de place se fait dynamiquement en cours d'exécution. On peut aussi dès que ces variables ne sont plus utiles libérer la place. On s'en sert peu pour de petites applications, mais bien pratique pour les autres !

- ***Constantes***

Exemple : const float pi = 3.1416 ;

Leur valeur n'est pas modifiable par le C. Existent avant l'exécution du programme

- ***Comparaison avec les variables déjà connues en assembleur***

Type de variables en C	Exemples de déclaration en C	Déclarations assembleur correspondante
Globales	/* En dehors d'une fonction */ Int **val16** ; /* 16 bits non initialisée */ Main() { }	**public** val16 **val16** rmb

Statiques	/* Dans une fonction en statique */ calcul() { **static** char **tab[4]** = {0,1,2,3} /*initialisé*/ … }	• pas de directive public **tab** rmb 4
Constantes	const char **coeff** = 0x2C ;	Public(selon le cas) **coeff** **coeff** fcb $2C

locales	/* Dans une fonction */
	calcul()
	{
	char **k** ;
	…
	}
		Pas d'instruction de déclaration directe
		Il s'agit de variables dynamiques, placées dans la pile, sans existence en dehors de la fonction !

Crées par **malloc**	Char *chaine ; chaine = (char*)malloc(100) ; /* chaine de 100 caractères */ /* pour libérer */ free chaine ;	**Pas d'instruction de déclaration directe** C'est de l'allocation dynamique, que l'on peut désallouer à volonté !

5.1.2.1. Valeurs numériques:

On prend le cas du C standard pour processeurs 8 et 16 bits, il y a des variantes selon les compilateurs et processeurs, par exemple int sur 32 bits au lieu de 16.

Char	entier 8 bits signé (ou non signé)	de -128 à 127 ou caractère
int	entier 16 bits signé	de -32768 à 32767
long	entier 32 bits signé	de -2 147 483 648 à 2 147 483 647
unsigned char	entier 8 bits non signé	de 0 à 255 ou caractère
unsigned int	entier 6 bits non signé	de 0 à 65535
unsigned long	entier 32 bits non signé	de 0 à 4 294 967 295

float	flottant simple précision Sur 4 octets : 23 bits mantisse, signe, 8 bits exposant. Dynamique : de $0,5.2^{-64}$ à presque 1.2^{63} ($10^{18 \text{ à } 19}$) Précision : $\pm 2^{-24}$ ou $5,9.10^{-8}$	Attention: sauf pour de gros processeurs, il faut une bibliothèque de calcul virgule flottante ! Les temps d'exécution sont bien plus long (qq ms) A ne pas utiliser dans les calculs à contraintes de temps d'exécution.
Double	Flottant double précision	

Les char sont signés ou non signés (option possible au niveau du complateur).

Dans certains cas, pour spécifier des int sur 16 bits et non sur 32 bits, on peut écrire short int.

5.1.2.2. Tableau ou chaîne de caractères

Représenté en fait par un **pointeur** sur le début de tableau ou de la chaîne.

Accès aux éléments d'un tableau par: ***pointeur** se promenant
tableau [k];

Une chaîne de caractères n'est en fait qu'un tableau de char, il doit se terminer par l'octet 00. Il n'y a pas de variable chaîne de caractères.

5.1.3. Déclarations

- ***Variables simples:***

int valeur1, valeur2;	non initialisées
int valeur1=123; /* décimal */ char n=-0x2F; /* Héxadécimal */ char c = 'f'; /* le caractère f */	initialisées

- ***pointeurs:***

Un pointeur contient une adresse, permettant d'accéder à toutes information en mémoire. Les pointeurs peuvent être volontairement des registres du processeur quand on programme en assembleur, le compilateur les place en mémoire quand on les déclare en C, (sauf option ou déclaration spéciale).

char *pt;	pt sera un pointeur vers un char	***non initialisé !***
int *tab;	tab sera un pointeur vers un int	***non initialisé !***

Un pointeur peut adresser une valeur isolée, un tableau, ou une chaîne de caractères, un objet, il peut aussi permettre d'adresser un registre physique d'un composant (port d'entrée sortie, CAN, CNA, afficheurs .

Attention danger! Une déclaration type ***truc** laisse juste la place en mémoire pour le pointeur, mais celui ci n'est pas initialisé. Il pointe donc sur n'importe quoi, donc danger !!

Après déclaration d'un pointeur il faut donc:

- Initialiser le pointeur
- Réserver assez de place pour la donnée, le tableau ou la chaîne pointée!

(Sinon aucun message d'erreur à la compilation et n'importe quoi à l'exécution ou erreur de calculs, ou plantage !). Le C ne pardonne rien ….. !

- ***Structures***

Pour rassembler sous un même nom un ensemble de valeurs, de renseignements, créer des fiches. Exemple une fiche contenant des renseignements sur des personnes :

- <u>On définit la structure</u> (en dehors de toute fonction)

```
Struct personne
{char nom[50] ;          /* nom 50 caractères max, sans allocation dynamique */
 char    age ;
 char  telephone[20] ;  /* 20 caractères max pour le N° téléphone */
} ;
```

- <u>On peut déclarer un élément de ce type nouveau</u>

Struct personne **personne1** ;
On accède aux éléments internes par : **personne1.nom**
personne1.age

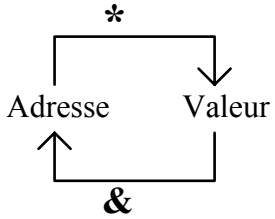
- <u>On peut déclarer un pointeur vers un élément de ce type nouveau</u>

Struct personne ***pers** ;
pers = (struct personne ***)malloc(sizeof(struct personne) ;**
/* réservation de place dynamiquement, nécessaire ! */
On accède aux éléments internes par **pers->nom**
pers->age

- ***Exemples:***

int valeur; int *ptvaleur; ptvaleur=&valeur;	Déclaration de **valeur** de type int **Déclaration** d'un **pointeur** ptvaleur sur un int **Initialisation** du **pointeur** sur l'adresse de valeur
char *chaine = ″chaine de caractères″ ; char chaîne[20]; strcpy(chaîne, ″chaine de caractères″); (on pourrait aussi faire de l'allocation dynamique)	chaîne de caractère initialisée; chaîne de caractère non initialisée; initialisation en exécution : Attention, à la taille, il faut réserver une place >= nombre de caractères + le$00 de fin de chaîne !!
int tab[5] = { 1,2,3,4,5}; int tab[20];	tableau initialisé tableau de 20 int non initialisé
int *tab; tab = (int*)malloc(20*sizeof(int)) ; free(tab) ;	pointeur sur un int, et tableau dynamique non initialisé de 20 int libération de la place allouée à ce tableau
#define base_port 0x6000 char ***port1 ;** /* ports 8 bits */ char ***port2;** port1 =(char *)base_port; port2 = port1+1 ; /* +1 taille pointée */ *Utilisation :* ***port1 = 0x34 ;** /* sortie de $34 sur le port*/	Déclaration de **pointeurs sur des registres** de **circuits physiques** On peut aussi définir une **structur**e définissant un circuit, et placer un pointeur
#define base_port 0x6000 #define port1 *(char*)base_port #define port2 ***(char**)base_port+1 ; *Utilisation :* **port1 = 0x34** ; /*port1 est directement la donnée pointée */	**Déclaration de registres physiques sans pointeur en mémoire**. Solution préférable et la plus simple !

Rappel important, résumant tout !



5.1.4. Changement de type (Cast)

```
char val8, *pt8;                    int val16, *pt16;
val8 = (char)val16;      /* prise des 8 bits de poids faible */
val16 = (int)val8;       /* extension de 8 à 16 bits, en signé ici */
pt8 = (char *)pt16;      /* pt8 prend la même valeur que pt16, mais pointera un char
                            et non plus un mot */
```

(permet de lire ou d'écrire un tableau ou une valeur numérique quelconque octet par octet)

5.1.5. Opérateurs de base

Arithmétiques	***Binaires***	
	&	ET bit à bit
+ - * / % (modulo= reste de la division entière)	|	OU bit à bit
	>> n	déc droit de n bits
>>b et << b décalages droite ou gauche de n bits	<< n	déc gauche de n bits

Relationnel						*Booléen*	
= =	>	<	>=	<=	!=	!	NOT
égal	sup	inf	sup ou egal	inf ou égal	différent	&&	AND
						||	OU
						^	OU EXCLUSIF

Attention aux priorités. Elles sont classiques pour les opérations arithmétiques (* et / avant + et -). Pour les autres, méfiance, mettre des parenthèses dans le doute !

Attention au format des variables obtenus avec les opérations arithmétiques :
Le **format du résultat** est **le même que celui des opérandes** : (ex si a et b sont des int, p = a*b est un int, d'ou dépassement possible, faire des cast en long si nécessaire pour travailler dès le début avec une taille d'opérande pus importante.

Attention ne pas confondre par exemple & et &&, ni le = avec le = = . Pas de message d'erreur bien sur, mais un comportement totalement différent !

Ecritures arithmétiques rapides et élégantes, ou mystifiantes ...

- Au lieu de faire s = s + machin, on peut faire plus élégamment s + = machin;
- A la place de s = s +1 on peut faire s++ !

(Remarque, pour un **pointeur**, **l'unité d'incrémentation** est la taille de l'objet pointé).

5.1.6. Instructions de boucle et de test

test après	**test avant**	***boucle sans fin***	**boucles d'attente**

do
{
instructions;
}
while(condition vraie);
	while(condition vraie)
	{
	instructions ;
	}
		do
		{
		
		.....;
		}
		while(1);
			while((*cra & 0x80) = = 0);
			
			Attendre tant que le bit 7 de la valeur cra est à 0

Boucle for	**if**

for(k=0 ; k < 128; k++)
{
...
}
	If (condition vraiee) { ……..
	}
	else { ……..
	}

Autres écritures de boucles sans fin :

```
While(1)          for( ; ; )
  {                 {
   …..               …..
   }                 }
```

Attention pas de virgule en fin de ligne, sinon les boucles sans fin ne font rien !

5.1.7. Fonctions sur chaînes de caractères

strcpy(char *chaine_destination, char *chaine_source ;/* Attention à la place disponible ! */
strcat(char *chaine_initiale_et finale, char *chaine_à_concaténer); /***attention à la place** */
int **strlen**(char *chaine); /* longueur de la chaine */
Etc ....

5.1.8. Instructions classiques d'entrée sortie printf et scanf ?

Ces fonctions classiques du C font appel évidemment aux fonctions « Système » d'un ordinateur, pour une application strictement microprocesseur, elles sont d'aucune utilité sauf pour la mise au point si le compilateur en permet l'usage, pour afficher des résultats sur l'écran du système de développement, ces instructions sont à retirer pour l'application définitive. Exemple : int val_entier = 567 ;
float val_flottante = 45.88 ;

printf(″%d %s %x %s \n″ , val_entier, ″nombre entier″,val_flottante, ″nombre flottant″) ;

On obtient l'affichage : 567 nombre entier 45.88 nombre flottant retour ligne

Il est possible et facile par contre de se créer en assembleur des **fonctions d'entrées sortie** utilisables en C, **spécifiques** à la carte développée, par exemple travaillant sur un clavier lu par balayage, une visu deux lignes à cristaux liquides.. fournissant des affichages analogues.

5.1.9. Tests et manipulation de bits

1) Tests de bits

On **isole un bit** par un **ET logique** avec un opérande ayant un bit à 1 à l'endroit utile.

- Attente du bit 4 du « **status »** d'un circuit (b7 b6 b5 **b4** b3 b2 b1 b0)

```
#define status  *(char*)status
        While( (status & 0x10) = = 0) ;
```

- Attente du bit 4 **ou** du bit 0 (b7 b6 b5 **b4** b3 b2 b1 **b0**)

```
        While( (status & 0x11) = = 0) ;
```

- Attente des deux bits 4 **et** 0 : (b7 b6 b5 **b4** b3 b2 b1 **b0**)

```
        While( (status & 0x11) ! = 0x11) ;
```

- Test du bit le plus à gauche, façon plus rapide : (**b7** b6 b5 b4 b3 b2 b1 b0)

```
        If( status  < 0 ) {  }          Si status est un char signé …. !
```

Donc par sécurité l'indiquer par un #define **status** ***(signed char*)**status car certains compilateurs par défaut travaillent sur des char non signés … Attention…. !

2) Manipulations de bits

Valable pour de **braves variables** en mémoires ou pour des registres standards en écriture et lecture. Pas pour des drapeaux situés dans certains registres spécifiques.

Attention, ceci n'est pas toujours valable pour certains drapeaux (d'interruption ou autres) certaines se remettent parfois à 0 en écrivant 1, ou en lisant ou écrivant à une autre adresse ! Voir la Doc technique et adapter ! !

- **Mise à 1** du bit 4 (valeur sur 8, 16 ou 32 bits)

Char, int ou long valeur ;

```
        Valeur = valeur | 0x10
```

- **Mise à 0** du bit 4 (valeur sur 8, 16 ou 32 bits)

```
        Valeur = valeur & 0x10 ;
```

- **Changement** d'état du bit 4

```
        Valeur = valeur ^ 0x10 ;   /* ou exclusif */
```

5.1.10. #define et Compilation conditionnelle et macros

5.1.10.1. #define

Permet un peu comme equ et le #define en assembleur de dire qu'un nom est défini ou qu'un nom peut être remplacé par d'autres caractères pour faciliter le compréhension du code source. Ce que l'on définit ainsi n'est **donc pas en mémoire** !

On peut avoir quelques constantes qu'il est inutile de mettre en mémoire, mais qui doivent être utilisées dans plusieurs fichiers du projet (aussi bien fichiers C que fichiers assembleurs).

Exemple :

La fréquence choisie F_BUS de travail du microcontrôleur HC12 (valeurs possibles 2,4,8,16,20,24 MHz) peut être définie par **#define F_BUS 2**

On désire que cette valeur soit connue dans tous les fichiers (elle conditionne en effet la programmation de nombreux registres). La directive extern est sans objet, puisqu'il ne s'agit de variables ou constantes en mémoire !

Solution :

- On écrit un fichier d'entête par exemple **ini_carte.h**

Remarque : les directives #ifnfef (si non défini) et #endif (fin de if) sont nécessaires car ce fichier d'en tête étant à priori inclus dans plusieurs autres fichiers, un message d'erreur de multiples définition interviendrait !

```
#ifndef  F_BUS
#define  F_BUS 2
#endif
```

- On inclut ce fichier dans tous les fichiers C (et assembleur comme nous l'avons vu) du projet, par **#include ″ini_carte.h″.**

Fonctionnement: Partout ou est écrit F_BUS, et dans tout le projet, l'assembleur ou le compilateur remplace cette écriture par 2.

5.1.10.2. Compilation conditionnelle

Une même source peut pour une application être compilée de plusieurs façons selon par exemple des options choisies ou non, selon qu'il s'agit d'une version réduite ou complète….

Si on définit un symbole dans un seul fichier, le **#ifndef** …. **#endif** n'est pas nécessaire.

Obligatoire par contre si on définit le symbole dans un fichier.h à inclure dans plusieurs autres, comme précédemment.

5.1.10.2.1. TECHNIQUE N°1, COMPARAISON A UNE VALEUR NUMERIQUE

On peut définir et donner une équivalence pour un symbole VERSION par :

```
#define  VERSION   1
```

Et dans le code on écrira :

```
#ifdef  VERSION==1
    lignes de C
#else
    autres
    lignes de C
#endif
```

5.1.10.2.2. TECHNIQUE N°2, SI UN SYMBOLE EST DEFINI OU NON

On peut définir un symbole : VERSION_1 ou VERSION_2 sans donner d'équivalence :

```
#define VERSION_1
```

Et dans le code on écrira :

```
#ifdef VERSION_1
    lignes de C
#endif
#if  VERSION_2
    autres lignes de C
#endif
```

On peut aussi utiliser des #else

5.1.10.3. Macro

Existent également en C. Non étudiées ici.

5.2. Fonctions en C

5.2.1. Sans paramètres, fausses fonctions

On peut écrire évidemment des fonctions sans paramètre **void Fonction(void) ;**

Mais celles-ci ci doivent alors travailler sur uniquement des paramètres variables globales. (Des variables locales, internes à la fonction peuvent évidemment exister). On gagne un peu en rapidité d'exécution car il n'y a plus de passage de paramètres, mais on perd en polyvalence.

Application :

- **fonctions courtes** et **spécifiques d'une application**, ou la rapidité d'exécution est une contrainte sévère.
- Pour placer des **instructions assembleurs spécifiques** telles que les instructions d'entrée-sortie de certains processeur ne travaillant pas sur des adresses de ports dans le plan mémoire (in et out), mais aussi les instructions très classiques CLI ou SEI validant ou masquant les interruptions générales.

Exemple fonction de validation :

Extern **void valid_il(void) ;**

Utilisation :
valid_it() ;

Dans un fichier assembleur :

```
        Public valid_it
valid_it  cli
          rts
```

(Pour des compilateurs acceptant des lignes d'assembleur directement dans le C avec une syntaxe appropriée, passer par une fonction est inutile, on ajoute JSR et RTS pour rien. On peut aussi parfois déclarer la fonction 'inline' qui permet aussi de supprimer cet aller retour).

Mais une « vraie fonction » est évidemment une fonction pouvant être réutilisées à tout moment dans l'application, ou pour des applications très diverses, elle doit pouvoir être mise (en code relogeable) dans d'autres fichiers ou dans une librairie. Elle doit donc travailler à chaque appel sur des « paramètres » transmis différents, elle doit être si possible réentrante .. Donc rien de tout cela n'est possible avec ces fausses fonctions !

5.2.2. Types de paramètres transmis, vraies fonctions

- **Transmission par valeur**: les paramètres d'entrée ne peuvent pas modifiés par la fonction, ce qui est évident puisque le sous programme ne connaît pas l'adresse de la valeur passée, on peut récupérer cependant un ou plusieurs résultats: (un seul en C).

  Exemple en C : **Z = Fonction(A, B, C);** A, B et C sont les valeurs des variables A,B et C. On récupère un résultat Z.

- **Transmission par adresse** : les paramètres peuvent travailler en entrée et en sortie. Exemple en C : **Fonction(A,B,C,&X,&Y);** A,B,C sont les valeurs de A,B et C, tandis que &X et &Y sont les adresses de X et de Y . Cette fonction peut modifier X et Y mais non A, B et C.

(En Pascal, on parle de procédure, en Fortran de Subroutine, les notations différent)

On peut ainsi transmettre et modifier facilement des tableaux de valeurs, des chaînes de caractères, en ne fournissant au sous programme que l'adresse du début du tableau ou de la chaîne.

On se sert en fait souvent des deux types de passages, par valeurs et par adresses.

♦ **Valeur retournée (valeur ou pointeur)**

Une fonction en langage évolué ne retourne qu'un seul paramètre (valeur ou adresse), ce n'est pas un handicap puisque que par le passage par adresse, la fonction peut, même en ne retournant rien modifier toutes valeurs. En assembleur seul, on peut évidemment retourner ce que l'on veut.

5.2.3. Exemple simple de fonction

calcul de la somme des valeurs d'un tableau.

On n'utilisera pas l'allocation dynamique.

On choisit ici simplement un tableau de taille = 10 contenant les nombres de 1 à 10. Mais pour que la fonction marche dans le cas général, pour ne jamais déborder, on choisit des opérandes sur 16 bits, et on effectue l'addition sur 32 bits.

Nous présentons les deux cas: le retour d'un opérande, ou le passage par adresse de l'opérande modifié.

Retour de la somme par valeur	Adresse de somme en opérande

#define taille_max_pratique 200
long somme(int taille_tableau, int *tableau);
 	
main()
{
int tab[taille_max_pratique];
int k,taille;
long **s**;
taille = 10; /* par exemple */
/* initialisation du tableau de taille valeur*/
for(k=1; k <=10;k++) tab[k] = k;
s = somme(taille, tab);
}
 	
long somme(int taille_tableau, int *tableau)
{
int k;
long x = 0;
for(k=0; k< taille_tableau; k++)
x+=(long)tableau[k];
return(x);
}
	#define taille_max_pratique 200
	somme(int taille_tableau, int *tableau,**long** *resultat);
	
	main()
	{
	int tab[taille_max_pratique];
	int taille;
	long int s;
	taille = 10; /* par exemple */
	/* initialisation du tableau de taille valeur*/
	for(k=1; k <=10;k++) tab[k] = k;
	somme(taille, tab, **&s**);
	}
	
	somme(int taille_tableau, int *tableau, **long** *resultat)
	{
	int k;
	long x;
	x=0;
	for(k=0; k< taille_tableau; k++)
	x+=(long)tableau[k];
	*resultat = x;
	}

Le prototype est parfois facultatif, (mais pratique pour contrôler à chaque fois les types).

Attention, si on omet le prototype , les « cast »implicites si il en existent ne peuvent s 'effectuer, et on obtient parfois des résultats ou comportements totalement erronés !

Conséquence : par précaution, toujours mettre le prototype !

Le cast (long)tableau[k] est nécessaire pour additionner sur 32 bits des nombres de 16 bits.

Variantes d'écriture :

ligne d'instruction	*équivaut à:*
long somme(int taille_tableau, **int *tableau**);	long somme(int taille_tableau, **int tableau[ ]);**
s = somme(taille, **tab**);	**s** = somme(taille, **&tab[0]**);

5.3. Retour sur le travail d'un compilateur

Il est intéressant et nécessaire de mieux connaître comment marche un compilateur pour :

- Implanter en mémoire un logiciel sur une carte à microprocesseur.
- Effectuer la mise en mémoire non volatile de l'application définitive.
- Savoir ce qui se passe **avant le main()** !
- Comprenne le passage de paramètre …

Nous avons entrevu son fonctionnement dans un exemple simple de démonstration (Application avec 4 leds.c)

5.3.1. Les grands principes

La phase de compilation proprement dite transforme le code C, en code assembleur, en respectant une structure bien définie. Des variantes existent suivant les compilateur, mais les grands principes sont les mêmes.

La pile système :

- **Sauvegardes automatiques** par le processeur lors des saut à des sous programmes (Compteur programme) ou lors des interruptions (Compteur Programme, Code condition et divers registres).
- **Variables locales.** Elles sont donc dynamique et sont crées lors de l'entrée dans la fonction (ou le main), disparaissent en sortie.
- P**aramètres** passées aux fonctions. (certains passent par registres).

Les **variables globales et statiques** sont placées dans des **sections spéciales.**

Les **valeurs initiales** des variables initialisées, des chaînes de caractères, les constantes sont écrites dans d'autres sections.

On spécifie les **adresses réelles** correspondant à la carte, uniquement à **l'édition de lien**.

Le compilateur s'occupe de l'**initialisation** correcte des **vecteurs d'interruptions** (et d'éventuelles sauvegardes nécessaires non automatiques si besoin est) (**Interrupt Handler**).

Le **passage** de la **mise au point à la mise en ROM** est **très simple**, au niveau du LINK, en modifiant seulement l'adresse de quelques sections et en choisissant éventuellement une option pour générer un fichier (type Motorola S1S9 par exemple), destinée à un programmateur externe.

➢ ***Point d'entrée du programme ?***

Le point d'entrée n'est jamais immédiatement le main(). Quelques instructions assembleur sont générées automatiquement par le compilateur, elles assurent l'initialisation :

- Initialisation de l'indispensable pointeur de pile,
- Initialisation des variables initialisées

- Initialisation de tout l'environnement C : registres, pointeurs diverses … Environnement spécifique du processeur et du compilateur.
- Saut au sous programme main() : JSR main
- Retour vers moniteur ou système d'exploitation (en Mise au point)

Ces lignes se trouvent dans une librairie du C (un code relogeable de nom par exemple boot.obj), et sont prises en compte au moment de l'édition de lien.

➢ ***Répartition en mémoire des différentes « SECTIONS »***

Le **compilateur** génère pour chaque sections des **codes assembleur relogeables**, que l'éditeur de lien placera correctement mémoire. Les noms de section dépendent du compilateur, mais on retrouve toujours les sections standards citées ci-dessous:

Sections destinée à être en ROM (application définitive) **Zone programme** **Sections de « type CODE »**	- Code - Valeurs initiales des variables initialisées (nombre et chaînes de caractères) - Constantes - Vecteurs d'interruption. Le vecteur Reset au minimum en mode mise en ROM

Sections destinées à être en RAM : **Zone données** **Sections de type « DATA »**	- Variables globales ou statiques initialisées - Variables globales ou statiques non initialisées - Pile pour variable locales, sauvegardes lors des sous programmes et interruptions, et pour passage de paramètres. - Vecteurs d'interruption déplacés ou table de JMP (en mise au point)

5.3.2. Exemple : le compilateur IAR Systems

5.3.2.1. Les sections utilisées par ce compilateur

Section de type CODE (destinée a être en ROM)

High **INTVEC**	Table des vecteurs d'interruptions (2 octets par interruption) (de $FFD6 à $FFFF)
CONST	Données déclarées const, et autres valeurs initiales.
CCSTR et CSTR	Valeurs initiales des Chaînes de caractères initialisées et chaînes constantes.
CDATA0 et **CDATA1**	Valeurs initiales des variables initialisées dans IDATA0 et **IDATA1**
CODE	Code exécutable
RCODE low	Code exécutable provenant des librairies du C : boot.obj du C, autres routines de l'outil

Section de type DATA (destinée à être en RAM)

High ECSTR et 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 low	Idem pour la page 0 seulement (adresses de 0 à 255, accessibles par adressage direct plus rapide)

5.3.2.2. Exemple de fichier de lien, pour un matériel donné

```
// 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 interrrupt 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)
cl6812
```

On retrouve les différentes sections décrites précédemment.

Ce fichier assurer l'implantation correspondant à une carte HC12 (plan mémoire vu précédemment pour nos programmes simple de démonstration). Elle permet le développement rapide de l'application car 1000 à 2FFF est en RAM. Pour l'application, définitive il ne restera qu'à changer cette zone d'adresses.

Mise en garde : La position du pointeur de pile évolue sans cesse en exécution , pour :

Empilement de paramètre lors des appels aux fonctions.
Empilement du compteur programme lors des sauts aux sous programmes
Empilement de tous les registres lors des interruptions.
Naissance des variables locales dans chaque fonction.

On a réservé 200 octets seulement pour la pile, si on déborde en exécution, il n'y a aucun message qui préviendra ! !. Il faut toujours **être sûr de la taille maximale nécessaire** pour la pile. Si on craint un débordement, il faut étudier le pire des cas (nombres de couches de fonctions, d'interruptions), et majorer par précaution !

Un **plantage** de ce type est très souvent **difficile à détecter** !

5.4. Optimisations du C pour accélérer la vitesse

On considère toujours l'HC12 et le même **outil IAR_System**

5.4.1. Programme simple testé

Soit un tableau (en variable locale) de taille 100 (définie par un # define). On doit l'envoyer à la **cadence la plus grande possible** sur un port d'entrée_sortie de label PTM défini par **#define PTM *(char *)0x250 ;**

Surtout pour les codes pas ou peu optimisés, ne pas chercher forcément l'utilité de chaque instruction ! ! (les variables sont en pile et l'HC12 ne sait pas les adresser directement à partir du pointeur de pile S, d'ou l'usage fréquent de l'index X). Observez surtout la boucle et le nombre d'instructions !

5.4.2. Niveaux d'optimisation différent à la compilation.

➢ ***Options et Niveaux d'optimisation :***

Selon le but recherché, on peut choisir une optimisation **en taille** de code ou **en vitesse** (l'une ou l'autre fournissent souvent des résultas similaires). Il peut exister aussi une optimisation en **taille de pile**.

Plusieurs **niveaux** d'optimisation sont possibles sur les compilateurs, en effet l'optimisation maximale fait parfois disparaître des instructions indispensables pour vous mais absolument superflues aux yeux du compilateur, il peut alors les supprimer sans hésiter, et votre programme ne marche pas. Des 'warning' signalent parfois que le niveau choisi est trop élevé et que le compilateur à peut-être supprimé des choses utiles.

➢ ***Résultats :*** La ligne principale du programme s'écrit :

For(k = 0 ; k < taille ; k++) PTM = tab[k] ;

L'indice k de ce genre de boucles est toujours en variable locale (sauf cas très particulier, la valeur de ce k n'est pas utilisée dans une autre boucle, et encore moins dans une autre fonction). On gagne souvent un peu en vitesse d'exécution avec ces variables locales donc en pile, ou même résidant juste dans un registre.

Cas N°1 ***pas d'optimisation***	***Code assembleur généré pour la boucle :***

#define taille 100
 	
For(k = 0 ; k < taille ; k++) PTM = tab[k] ;
 	
Tab[ ] et k en locale.
 	
Résultat :
 	
Boucle de 10 instructions !
	CLR 0,SP compteur à 0
	?0014:
	LDAA #99 taille-1
	CMPA 0,SP
	BLE ?0013 fin de boucle
	?0015:
	LEAX 5,SP
	LDAA 0,SP
	SEX A,D calcul début tableau
	LEAX D,X + indice
	MOVB 0,X,592 écriture PTM
	INC 0,SP incrémentation compteur
	BRA ?0014
	?0013

Cas N°2 ***Optimisation max en vitesse*** Même écriture. Résultat : Identique ! 10 instructions !	**Même code généré** pour ce cas simple ! Améliorations plus visibles pour des codes sources plus compliqué ! Et d'autres compilateurs peuvent faire beaucoup mieux !

5.4.3. Ecritures astucieuses du C

On aide le compilateur à générer un code plus court. Si l'on programmait nous même en assembleur on utiliserait un pointeur, faisons donc de même en C, avec un pointeur explicite, et nom par l'écriture tab[k]

Cas N°3 **Optimisation max** + Usage d'un **pointeur** déclaré par **char *pt** ; et initialisé sur tab **pt = tab ;** **For(k = 0 ; k < taille ; k++)** **PTM = *pt++** ; Résultat : 5 instructions Nettement mieux ici !	Et plus facile à comprendre : le compteur est en A. Pointeur dans X et auto incrémentation. LEAX 0,SP X sur tab LDD #1 ini compteur mais D ? **?0014** CMPA #99 **BGE ?0011 fin boucle** MOVB 1,X+,592 INCA BRA **?0014** **?0011 :**
Cas N°4 **Idem + Décrémentation** sur **k** **For(k = taille ; k >0 ; k--)** **PTM = *pt++ ;** Résultat : 5 instructions	Plus de comparaison à 99, mais à zéro ! LEAX 0,SP LDAA #4 **?0014** CMPA #0 **BLE ?0011** MOVB 1,X+,592 DECA **BRA ?0014** **?0011**

On ne peut pas faire mieux tout en C. Mais en assembleur, on devine que l'on peut encore gagner un peu en évitant la comparaison : CMP !

5.4.4. Programmation en assembleur dans le C

On passe en fait par l'écriture d'une fonction C en assembleur. Mais pas nécessairement une vrai fonction avec passage de paramètres, qui pour l'instant nous entraînerait trop loin, mais par une 'fausse fonction' travaillant sur des variables locales. On peut alors écrire soi même et facilement les lignes assembleur correspondant.

5.4.4.1. Fausses fonctions

Le prototype est rudimentaire : **void sortir(void) ;** La fonction travaille sur **tab[]** en **variable globale obligatoirement**.

Très souvent le nom du sous programme de la fonction en C à le même nom que le sous programme assembleur correspondant.

Prototype en C :	*Code assembleur généré à la compilation*
Extern void sortir(void) ;	Directive assembleur : extern sortir
Appel en C :	
sortir() ;	Code exécutable **Jsr sortir**

Il suffit donc d'écrire dans un fichier assembleur et en assembleur le sous programme, mais en **préservant le contexte du C**. Il faut donc bien lire la documentation du compilateur pour voir si vous devez sauver tous les registres ou si ce n'est pas utile.

Cas N°5	**IAR system** nécessite de préserver **seulement X et Y.** (donc attention, ne jamais utiliser ce sous programme tel quel dans un programme tout assembleur, il détruit A et B, il faudrait les sauver en pile ! !)
Fausse fonction C, écrite en assembleur, donc **boucle optimisée en assembleur** ! **Tab[] et taille** en variable **globale !** Et appel en C : **sortir() ;** Résultat : Boucle de **3** instructions ! Mais attention ici : une taille nulle ici entraîne une boucle de 256, ce qui est pratique mais dangereux si on l'oublie !	Donc à écrire dans un fichier assembleur : public Sortir Sortir LDX #tab pointeur LDAB #100 compteur **bou** MOVB 1,X+,$250 tableau→ PTM DECB compteur-- BNE **bou** RTS

5.4.4.2. Vraies fonctions ?

Une « fausse fonction » est évidemment très peu souple, ne peut être mise en bibliothèque travaillant toujours sur le tableau global de nom tab, n'est pas réentrante, … mais on peut s'en servir pour des systèmes simples.

Une fonction plus polyvalente aurait en fait comme prototype :

void sortir_portc (unsigned char taille, char *tab) ;

Pour l'écrire en assembleur, il faut connaître ce que fait le C pour passer les paramètres. Nous verrons ceci dans un autre chapitre, ou cette fonction sortir sera donnée comme exemple simple. Sa rapidité sera identique une fois la boucle amorcée, par contre on perdra évidemment des cycles avant et en retour de sous programme pour le passage des paramètres.

Remarque : La boucle est la même. Une fonction encore plus polyvalente, de nom **sortir**, pourrait se programmer, en passant en paramètre l'adresse du Port.

5.4.5. Comparaisons des performances

Voyons les **durées** d'exécution **d'une boucle** en nombre de période d'horloge Tck d'exécution, et en temps pour Tck = 500 ns (Fck = 2MHz). Et avec l'HC12, on pourrait monter à 24MHz, et gagner encore dans un rapport de 12 !

- *Tout en C :*

Cas N° 1 24 Tck **12µs** (on aurrait eu 56Tck et donc 28µs avec l'ancien HC11 !)
Cas N° 2 24 Tck 12µs
Cas N° 3 11 Tck 5.5µs
Cas N° 4 11 Tck 5.5µs **gain de 2.18 déjà !**

- *Avec un peu d'assembleur :*

Cas N° 5 (fausse fonction C écrite en assembleur) :
9 Tck **4.5µs** **gain de 2.7** par rapport au cas 1 et 2 !
Cas N°6 (vraie fonction C en assembleur, voir autre chapitre)
9Tck **4.5µs** (idem, mais quelques cycles de plus avant et après la boucle).

Les gains sont certes significatifs.

Mais pour une application quelconque tout dépendra évidemment des instructions situées dans la boucle, du nombre de boucles, du reste du programme…..

6. LES FONCTIONS, DE L'ASSEMBLEUR AU C EN TYPE HC12

Les méthodes décrites ici, à des variantes près, sont assez générales, et se retrouvent donc sur tous les compilateurs.
On se propose ici :
- **En assembleur**, de décrire les techniques de **passage de paramètres** à des **sous programmes**, en les appliquant à un processeur type HC11,HC12.
- **En assembleur** toujours, de se créer des **macro** instructions (qui sont en fait des sortes de fonctions en assembleur utiles lorsque l'on de dispose pas de langage évolué).
- **D'écrire des fonctions C** en **assembleur** (utiles pour optimiser la vitesse ou la taille de code généré, ou utiliser pleinement toutes les instructions d'un processeur, ce que ne fait pas toujours un compilateur)

➢ ***Sous programme et fonction :***

A partir de certaines **variables d'entrée** (ou **paramètres**), un sous programme fournit des actions déterminées, mais aussi un certain nombre de résultats nommés **paramètres de sortie**.

Un sous programme assembleur ne fait rien tout seul, et c'est son utilisation que l'on peut nommer fonction. Donc, ne pas confondre le sous programme et la façon de l'utiliser !

Structure générale d'une fonction:

Préparation : (envoi des paramètres au sous programme)
Appel au *sous programme correspondant*
Retour éventuel de résultats
remise en état éventuelle (de registres, de la pile ....).

L'ensemble de ces 4 étapes doit être géré lors d'un appel de fonction.
On peut pour cela réaliser une **"Macro"** si on reste en assembleur seul, **ou** utiliser le sous programme à partir d'un **langage évolué** si on respecte certaines règles propre au compilateur utilisé.

Nous chercherons à réaliser en assembleur des "Fonctions" travaillant comme ci-dessus.

On désire des programmes sans soucis d'emploi pour une totale polyvalence (du genre de ceux que l'on peut mettre tranquillement en librairie).
Le ***sous programme ainsi que la fonction complète*** **devra** alors **ne pas modifier les registres**, **être interruptible et si possible réentrant.**
Un **cahier des charges précis** devra être établi.

Comme on cherche à se rapprocher d'un code généré par un compilateur, on ne dépassera pas en assembleur plus de 1 paramètre de retour pour la fonction.

6.1. Passages de paramètres en assembleur

On prendra tout d'abord comme exemple le calcul du **maximum de n octets** signés
Soit : **n** (en non signé 8 bits) le nombre de valeurs, ou **N** l'adresse de ce nombre
TAB l'adresse du début du tableau de valeurs
MAX l'adresse ou on veut ranger le maximum.
On veut passer ces paramètres à un sous programme calculant ce maximum.
Rappel de **l'algorithme simple** de calcul d'un maximum :
- On part d'un maximum à zéro (non signé) ou à la plus grande valeur négative.
- On examine chaque valeur du tableau, et si max < valeur on fait max = valeur.

6.1.1. Par registres

Pour L'HC12 qui possède peu de registres, cette technique n'est possible que pour des cas simples. Sinon c'est la plus rapide en vitesse d'exécution.

On passe en paramètre : L'adresse du début du tableau : dans **X**
Le nombre d'octets : dans **A**
Renvoi du max : par **A**

L 'ensemble doit préserver tous les registres !

♦ ***L'ensemble :***

```
        PSHX            X et A sauvegardés car utilisés dans ces lignes
        PSHA
        LDX  #TAB       Adresse du tableau dans X
        LDAA N          nombre de valeur n dans A
        JSR  calcul_max
        STAA MAX        rangement du résultat
        PULA
        PULX            X et A récupérés
```

♦ ***Le sous programme :*** Soit **Y compteur**, **B** pour calculer le **max**

```
calcul_max  PSHB   on préserve B et Y qui servent en plus
            PSHY
            TAB
            LDY  #0          Y+B existe et non Y + A
            ABY              Astuce pour mettre le nombre de valeurs dans Y
            LDAB #0 ou #$80  ini max  (en signé à 0 ou en non signé à -128)
Bou         LDAA 0,X
            CBA              compare A à B  (A-B ?)
            BLE  Rien
            TAB              valeur = max
Rien        INX  (code compatible HC11 sinon écrire plus haut :  LDAA 1,X+)
            DEY
            BNE  Bou
            TBA              mise du résultat dans A pour le retour
            PULY
            PULB
            RTS
```

Le sous programme modifie X et A mais comme à priori on n'utilisera jamais ce sous programme sans utiliser les lignes d'envoi des paramètres (où X et A sont sauvés), ce n'est pas gênant.

On voit qu'on utilise vraiment tous les registres !

On se sert d'une variable de calcul (max) mise dans le registre B, qui est une sorte de variable locale.

Question : Si on mettait cette variable à une adresse fixe ? ca marcherait mais le programme serait moins polyvalent, et même dangereux si on ne se souvient pas des restrictions d'emploi : il ne serait par réentrant (possibilité d'être interrompu et réutilisé dans le programme d'interruption), il serait interdit dans un système à temps partagé.

6.1.2. Par la pile, variables locales en pile, retour par registre

Permet de passer un grand nombre de paramètres, même pour un processeur à peu de registres. Technique un peu plus lente d'exécution.

On décide **d'envoyer par la pile** les paramètres d'entrée, dans l'ordre **TAB**, puis **N**.

On choisit un **retour** par le registre **A.**

L 'ensemble doit bien évidemment préserver tous les registres !

♦ ***L'ensemble :***

```
PSHX        X et A préservés car utilisés dans ces lignes
PSHA

LDX   #TAB
PSHX        empilement de l'adresse début de tableau
LDAA  N
PSHA        empilement du nombre de valeurs
BSR   Calcul_max1 (pas le même sous programme que précédemment)
STAA  MAX         Retour par A vers MAX
INS
INS         réajustement de pile
INS

PULA
PULX
```

♦ ***Le sous programme***

L' HC12 ne sait pas adresser des valeurs dans la pile par rapport au pointeur de pile S.

On doit d'autre part (sauf cas très particulier), **toujours laisser le pointeur de pile en bas de la pile** ! (indispensable pour toute appel à un sous programme, et si une interruption survient). On se sert donc de **X** comme pointeur supplémentaire. X se nomme alors le « **pointeur de trame** » (**Frame pointeur**).

On utilise une **variable locale** (Max) pour calculer le maximum, on la place **en pile**.

On se sert de Y (pointeur) et de B (compteur), on les sauvegarde donc dans la pile.

On se sert de X, mais X est déjà sauvé dans les lignes envoyant les paramètres au sous programme, il semble donc inutile de le rempiler de nouveau ! Attention, cette façon de voir serait dangereuse, on peut en effet très bien se servir de ce sous programme avec des lignes de préparation un peu différentes (usage d'autres registres). Il faut donc par sécurité sauvegarder de nouveau X dans le sous programme.

A sert de registre de retour , il ne faut donc évidemment pas que le sous programme le préserve !

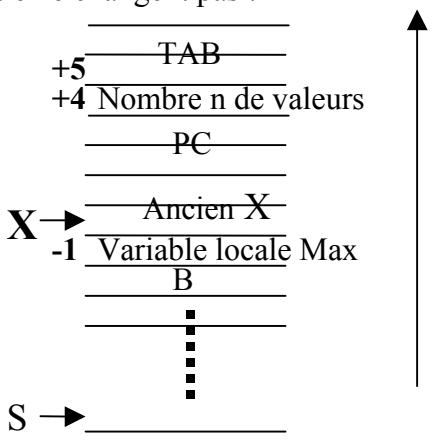
➢ Pour une programmation facile : Il faut toujours positionner X immédiatement !

```
Calcul1_max :
      PSHX En premier : X sauvé
      TSX  En second :  positionnement de X
      DES  En troisième :  place variable locale max

      PSHB   sauvegardes de B et Y
      PSHY

        LDY  5,X      adresse TAB dans Y
        LDAB 4,X      n dans compteur B
        LDAA #0  ou  -128  (non signé ou signé)
        STAA -1,X     ini Max
BOU     LDAA 0,Y      lecture octet
        CMPA -1,X     Comparaison à Max
        BLE  RIEN
        STAA -1,X     Mise à jour Max
RIEN    INY
        DECB          Décrementation compteur
        BNE BOU
        LDAA -1,X     Max dans A pour retour

        PULY
        PULB
        INS     Ordre !
        PULX

        RTS
```

On se sert de :
Y pour pointer le **tableau**
X pour pointer les **variables en pile**
B comme compteur
Retour par **A**

Variable locale Max en pile

Il faut dessiner la pile, les décalages indiqués sont par rapport à X.
Si on ajoute des variables locales ou des registres sauvegardés, les décalages pour accéder aux paramètres et à la première variable locale ne changent pas !



En positionnant X tout de suite, on voit que les décalages pour accéder aux opérandes ne changent pas si on ajoute des variables locales et si o empile d'autres registres. C'est donc un avantage pour ne pas faire d'erreur de programmation.

L'instructions spécifique importante:
TSX donne **SP → X** (pointeur de pile dans X)
Elle permet de placer le pointeur **X sur** le **dernier octet empilé**.
Remarque en HC11, cette instruction réalise S+1 dans X, mais les empilements s'effectuant alors par post_décrémentation, le pointeur S se trouve toujours une adresse en dessous.

6.1.3. Passage mixte

On peut très bien passer par exemple les premiers paramètres par les registres et les autres par la pile. C'est ce que font de nombreux compilateurs. Nous verrons cet exemple un peu plus loin, appliqué à un compilateur C.

6.2. Macros (en assembleur)

Nous allons écrire maintenant une Macro pour faciliter la programmation des lignes qui effectuent le passage de paramètres. On crée ainsi une sorte de fonction en assembleur.

- Si on a des contraintes de vitesse, et un sous programme court, on peut développer tout le sous programme dans la Macro. Mais le code généré est alors bien plus long !!. Cela ressemble aux fonctions "inline" (que l'on peut spécifier dans un langage évolué).
- Si on n'a pas de contraintes de vitesse, on regroupe seulement dans la Macro les 4 étapes principales de la fonction.

6.2.1. Passage par registres

♦ ***Description de la macro :***

```
maximum     macro
PSHX  X et A sauvegardés car utilisés dans ces lignes
PSHA
LDX   \1    premier paramètre,  tableau dans X
LDAA  \2    second paramètre, nombre de valeur n dans A
JSR   calcul_max
STAA  \3    troisième paramètre, ou ranger le résultat ?
PULA
PULX        X et A récupérés
endm
```

♦ ***Appel à la macro :***

Il faut évidemment respecter le type et l'ordre de paramètres transmis ! sinon erreur d'assemblage si impossible, ou d'exécution qui est pire finalement ! (il n'y a pas de vérification de prototype …).

Par exemple dans l'exemple précédent La ligne **maximum #TAB,N,MAX**
fournit exactement les 8 lignes de code désirées pour le passage des paramètres (\1 est remplacé par #TAB, \2 par N, et \3 par MAX)

Une ligne telle que **maximum #$1000**, **#100**,**maxi** donnerait les lignes :

```
PSHX
PSHA
LDX     #$1000
LDAA    #100
JSR     calcul_max
STAA    maxi
PULA
PULX
```

6.2.2. Passage par pile ou mixtes, et retour par registre

Même technique, une macro permet de la même façon d'automatiser l'ensemble (envoi des paramètres, appel au sous programme, renvoi) et de créer une sorte de fonction, un peu comme dans un langage évolué …

6.3. Fonction C en assembleur

Dans le but d'**optimiser le temps d'exécution d'une fonctions C**, et aussi la taille du code, il est intéressant de l'écrire en assembleur. Le gain de temps est surtout important sur les fonctions comportant des boucles de calculs.

Il y a **plusieurs façons de travailler:**

-fonction C entièrement écrite en assembleur

-A partir de sous programmes assembleur déjà existants, on peut les utiliser (avec ou sans modification) pour en faire une fonction C.

Remarque importante : on peut souvent s'inspirer du code généré déjà par le compilateur lors d'une programmation tout en C, on observera au préalable le résultat fourni, avec optimisation maximale, ou non, ensuite on essayera de faire mieux…

Attention*! :* On ne devra pas détruire **l'environnement C**, entre autres: le pointeur de trame si le compilateur l'utilise, ainsi que les registres dédiés aux différentes piles.

Il faut respecter le mode de travail du compilateur, et **connaître les registres à ne pas modifier** et aussi **ceux sur lesquels on peut travailler sans crainte**, sans même avoir besoin de les restaurer. En effet, les compilateurs autorisent souvent, dans une fonction écrite par soi même en assembleur, l'utilisation libre de certains registres sans nécessiter une restauration (se reporter à un document sur le compilateur), de tels sous programmes seraient naturellement d'usage interdit en assembleur seul pour raison évidente !

On rappelle les grands principes de travail des compilateurs :

- Toutes variables locales dans la pile système (ou autre)
- Toutes variables Globales et statiques autre part.
- Passage des paramètres (valeur ou adresse) par la pile (ou mixte comme par exemple par registre pour un retour, mais aussi parfois par registres pour les premiers et par la pile ensuite).

On suppose ici que le compilateur utilise **les règles suivantes** lors des appels aux fonctions :

Premiers paramètres (de gauche à droite) passés dans l'ordre par **B, A, ou D** si 16 bits

Puis empilement (pris de droite à gauche, donc attention à l'ordre) des paramètres restant.

Retour par **B** (**ou D** si 16 bits)

Le nom assembleur du sous programme de la fonction porte le même nom que celui de la fonction.

On doit **préserver seulement X et Y** (donc libre usage des autres registres A, B)

(Ce sont les règles du compilateur IAR Systems….)

6.3.1. Premier exemple simple

Reprenons notre calcul d'un maximum. On peut écrire alors la fonction suivante:

♦ ***prototype****:*

On déclare en même temps la fonction externe au fichier C*:*

extern char maximum(char taille, char *tableau) ;

♦ ***Appel à la fonction:***

MAX = maximum(n, TAB) ;

Cette ligne génère automatiquement des instructions effectuant tout seul le passage de paramètres.

Pour ce compilateur on sait que :

Le premier paramètre à gauche, soit la taille **n** est passé par **B**
Le second paramètre, **TAB** est sur 2 octets, il sera **empilé**.
Un **retour** se fait par **B**.

Des lignes ressemblant à celles ci dessous sont donc générées automatiquement :

```
LDAB  #n          nombre de valeurs dans B
LDX   #TAB
PSHX              empilement de l'adresse début de tableau
BSR   maximum     (pas le même sous programme que précédemment)
STAB  MAX         retour d'une valeur vers MAX
INS
INS               réajustement de pile
```

Des sauvegardes de B et X sont peut être présentes mais peu importe.

Il n'y a **rien à écrire à ce niveau**, c'est uniquement le sous programme **maximum** que l'on écrira en assembleur !

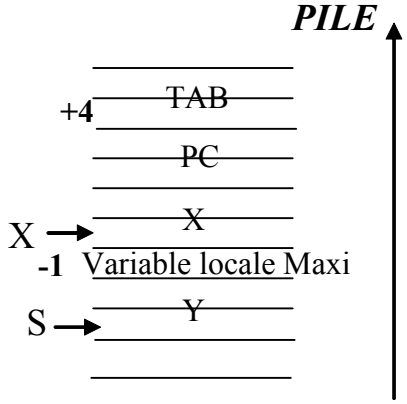
On reprend la technique de passage par la pile vue précédemment en assembleur, et l'usage pratique du pointeur de trame.

♦ ***Ecriture en assembleur de cette fonction :*** dans un fichier assembleur

Comme on doit préserver seulement Y, le code peut être plus court :

maximum :

```
        PSHX  En premier : sauvegarde X
        TSX   En second : pointeur de trame X sur le dernier octet empilé
        DES   En troisième, si besoin: variable locale Maxi
        PSHY  sauvegarde Y

        LDY 4,X     Y sur TAB
        LDAA #0     ou -128  (non signé ou signé)
        STAA -1,X   ini Maximum
BOU     LDAA 0,Y
        CMPA -1,X
        BLS  RIEN   ou BLE RIEN  (non signé ou signé)
        STAA -1,X
RIEN    INY         Compatible HC11
        DECB        Décrémentation compteur
        BNE BOU
        LDAB -1,X   Maxi dans B pour retour

        PULY    Récupération Y, attention à l'ordre !
        INS     Désallocation variable locale
        PULX    Récupération X
        RTS
```



Accu B

Nombre de valeurs **n**

DES (HC11) peut se remplacer par
LEAS 1,S (HC12 seul)

On peut aussi noter les deux différences selon que l'on travaille sur des nombres non signés ou des nombres signés.

♦ ***Et si on possède déjà un sous programme assembleur pour assembleur seul ?***

On suppose que l'on possède déjà le sous programme assembleur vu précédemment, travaillant par registre nommé **calcul_max :**

- On peut évidemment tout le modifier
- On peut en modifier juste quelques lignes au début et à la fin, si pas trop compliqué.

- On peut l'utiliser tel quel, c'est ce que nous allons voir maintenant. Ce sous programme pour assembleur seul devra avoir un nom différent du nom de la fonction.

Attention : Il faut connaître complètement le cahier des charges d'un tel sous programme pour assembleur, son mode de travail, si il utilise de vraies variables locales en pile ou de simples variables à des adresses fixes (programme alors non réentrant), il faut savoir si il modifie les quelques registres que l'on doit préserver.

Dans le doute, il faudra ajouter des sauvegardes en pile … (juste le nécessaire pour le C)

Si on n'est pas sûr de la réentrance, ne pas hésiter à écrire dans son cahier des charges que la fonction que l'on va réaliser est non réentrante !

Il peut parfois être astucieux de choisir un ordre d'opérandes plutôt qu'un autre.

➢ Premier choix de prototype : (Ordre des paramètres : taille n puis TAB)

Sous programme dèjà existant pour assembleur seul

Cahier des charges :

L'adresse début du tableau : dans **X**
Le nombre d'octets : dans **A**
Renvoi du max : par **A**
Aucun registre changé (à part A !)
Réentrant

On suppose qu'il fonctionne sans problème, on l'a vérifié, et qu'il ne perturbe pas l'environnement du C.

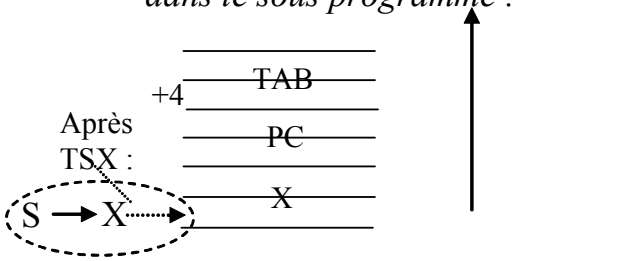
```
calcul_max   PSHB
             PSHY
             TAB
             LDY #0
             ABY
             CLRB
Bou          LDAA   0,X
             CBA
             BLE    Rien
             TAB
Rien         INX
             DEY
             BNE    Bou
             TBA
             PULY
             PULB
             RTS
```

Sous programme assembleur

Pour prototype :

Char maximum(char taille , char *tableau) ;

dans le sous programme :



Accu B Nombre de valeurs **n**

maximum :

```
PSHX
TSX
TBA      nombre d'octets dans A
LDX  4,X  TAB dans X
JSR  calcul_max  sous prog  ass seul
TAB    retour dans B
PULX
RTS
```

On voit qu'en ajoutant ces quelques lignes, on peut utiliser intégralement notre sous programme pour assembleur seul existant.

(on ajouterait éventuellement un PSHY et PULY si on n'était pas sûr que calcul_max préservait Y)

Le sous programme calcul_max préserve tous les registres, ce n'est pas forcément nécessaire pour le C (il faut juste préserver l'environnement C, ici X et Y) mais ce n'est pas nuisible (à part le temps d'exécution légèrement plus long …

On voit que le sous programme maximum détruit par contre A, ce n'est pas gênant, l'essentiel étant de préserver X et Y pour ce compilateur.

➢ Deuxième choix de prototype (ordre des paramètres : tableau TAB puis taille n)
L'adresse du tableau passe par D, il faut placer cette adresse en X.
Le nombre de valeurs n (en pile) devra être placé en A. Pour prendre n, le HC12 a besoin d'un pointeur de trame. On pourrait prendre X (mais en le sauvant et en le restituant car il contient déjà TAB), on prend ici Y (en le sauvant et en le restituant car on doit préserver ce registre).
Ainsi le sous programme calcul_max travaillera correctement, fournira max en A.
Le C renvoi par B, d'ou le TBA final.

Sous programme assembleur

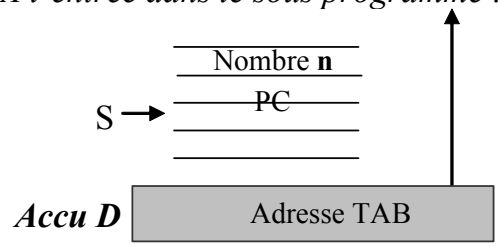
Pour prototype :
Char maximum(char *tableau, char taille) ;

maximum :

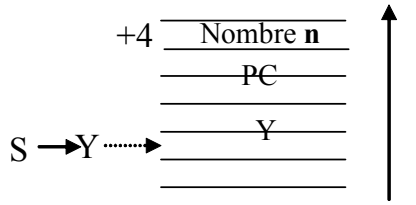
```
        PSHY      Y sauvé
        TSY       pointeur de trame Y
        XGDX      échange X et D, TAB dans X

        LDAA 4,Y  nombre d'octets dans A
        PULY      Y récupéré
        JSR calcul_max  sous prog ass seul
        TAB       pour le retour par B
        RTS
```

Ic, cette disposition est finalement un peu plus compliquée à programmer que l'autre

A l'entrée dans le sous programme :



Après PSHY et TSY :



6.3.2. Autre exemple simple

Et notre fonction « Sortir » un tableau sur le port C, que nous avions étudiée en C sous diverses écritures en vue d'optimiser le code et dont nous avions fait une « fausse fonction », c'est le moment d'en faire une vraie fonction !

Et pourquoi pas le plus polyvalente possible, en passant l'adresse du port en paramètre !

Prototype : **void sortir (unsigned char taille, char *tab, char *port) ;**

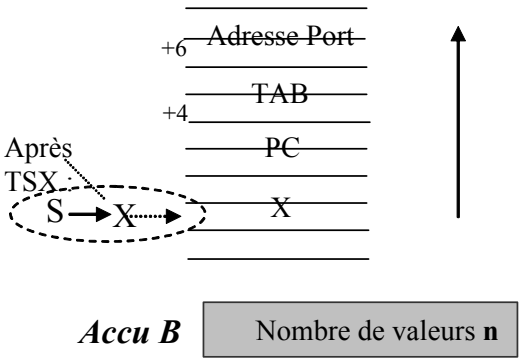
IAR system nécessite de préserver **X et Y.**

(donc attention, ne jamais utiliser ce sous programme tel quel dans un programme tout assembleur, il détruit X, A et B, il faudrait les sauver en pile ! !)

```
Sortir  PSHX
        TSX
        PSHY
                          Compteur déjà en B
        LDY  4,X          pointeur Y sur TAB
        LDX  6,X          pointeur X sur port
*                 Plus besoin de pointeur de trame
   bou  MOVB 1,Y+, 0,X    tableau → port
        DECB              compteur--
        BNE  bou
        PULY
        PULX
        RTS
```

Boucle toujours de 3 instructions !

dans le sous programme :



Cette fonction est **réentrante**, on peut en effet l'utiliser pour sortir sur un port et l'interrompre pour s'en servir pour sortir d'autres valeurs sur autre.

Attention, si on a déclaré #define PTM * (char*)0x250; il ne faudra évidemment pas écrire sortir(…, …., PTM) ; mais sortir(…, …., &PTM) ;

6.3.3. Autre exemple plus complexe, avec variable locale

Soit l'exemple du calcul de somme sur 16 bits de valeurs 16 bits signés, dans lequel on désire gérer un drapeau de dépassement (détectant même si un dépassement survient en cours de sommation) et essayons d'écrire en assembleur le fonction C correspondante.

Cet exemple serait bien plus facile en type 68xxx car on disposerait de plus de registres !

On veut écrire la fonction **extern char somme(int *tab, unsigned int n, int *s) ;**

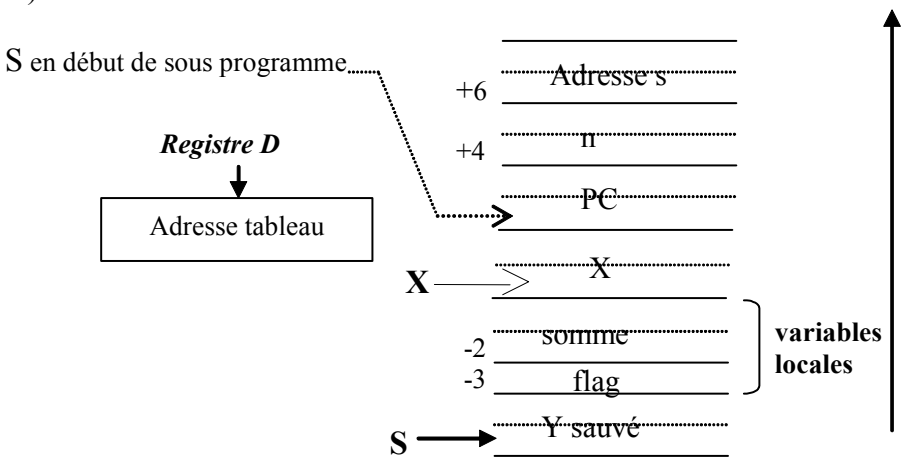
Paramètres d'entrée : Adresse du tableau de valeurs 16 bits
Nombre (sur 16 bits) de valeurs
Adresse ou ranger la somme sur 16 bits

Renvoi : 0 si OK
-1 si dépassement en cours de calcul.

On choisit :	X comme pointeur dans la pile (pointeur de trame)
	Y pour pointer le tableau
	Somme et flag deux variables locales dans pile.

→***Important:*** Comme en entrant dans la fonction on a placé immédiatement le pointeur de trame et réserver de la place pour des variables locales, les décalages pour l'accès aux paramètres et aux variables locales restent identiques même si on ajoute des variables locales ou des empilement dépilements divers.

- ***Registre D et pile*** après positionnement de X correctement (après le tsx du sous programme)



public **somme** (pour exporter ce label)

somme pshx
tsx ***X*** positionné correctement ***sur dernière variable empilée*** (S dans X)
leas –3,s place pour **variables locales somme et flag**
pshy à préserver obligatoirement pour le C, x déjà fait

```
xgdy            y pointe le tableau
ldd #0
staa -3,x       flag = 0
std -2,x        somme = 0
```

```
boucle   ldd 0,y          valeur pointée par Y dans D
         addd -2,x        D + somme dans D
         bvs  depass

pp       std  -2,x        rangement en somme
         ldab #2
         aby              incrementation du pointeur, + rapide que deux iny
         ldd  4,x
         subd #1          decrementation compteur
         std  4,x         (ne perturbe pas le bit Z)
         bne  boucle

         ldd  -2,x        reprise de somme finale
         ldy  6,x         adresse de somme dans y  (y sert pour pointer adresse résultat))
         std  -3,y
         ldab -3,x        flag dans B pour retourner

         ins              desallocation de flag
         pulx             desallocation de somme
         puly             récupération de Y
         rts
depass   psha
         ldaa #-1
         staa -3,x        flag à -1  (x pointeur de trame n'a pas changé avec le psh)
         pula
         bra  pp
```

- ***Performances sur l'outil de développement considéré*** en HC11 (1/3 moins en HC12)

Pour chaque boucle :

- **43 Tck** pour ce sous programme en assembleur.
- Et environ **70 à 78 Tck** pour une accumulation sur 16 bits **tout en C** selon les différents niveaux d'optimisation (le test de dépassement ne pouvant en outre pas se faire, car pour cela une addition sur 32 bits serait nécessaire et un nombre de cycles encore plus grand ..) Donc **gain non négligeable** !
-

6.4. Discussion sur le pointeur de trame

Rappel : sauf cas très particulier il est interdit de déplacer le pointeur de pile !	
Cas ou il est nécessaire :	Microprocesseurs ne permettant pas d'adresser des valeurs en pile par son pointeur de pile (Exemple HC11)
Cas ou il est pratique :	Lors de l'écriture d'un sous programme, les décalages par rapport au pointeur de pile, pour accéder aux valeurs en pile changent si on ajoute des variables locales et si on empile d'autres registres. D'ou l'usage pratique du pointeur de trame (si on le positionne le plus vite possible en début du sous programme).
Inconvénient :	Pour des processeurs à peu de registres, il faut parfois le sauver et le restituer plusieurs fois si on a besoin du registre concerné pour un calcul. (Nécessaire sur HC11, il ne serait pas forcément utile sur HC12 qui possède l'indexé par rapport à S, mais il est bien utile ….. !)
Inutile ?	Les compilateurs ne s'en servent pas toujours (si le mode d'adressage indexé par rapport à S existe).

7. SYNCHONISATION SUR EVENEMENTS EXTERIEURS, INTERRUPTIONS

Un programme doit très souvent se **synchroniser** sur des **événements extérieurs** :
On dispose pour cela : de **signaux** (lignes à 0 ou à 1)
de **simples bits** à 0 ou 1 (**drapeaux**)
de **valeurs particulières de variable** (en mémoire ou dans un registre d'un circuit quelconque)

Pour indiquer :

- En réception : qu'une donnée à été envoyée par un périphérique et que celle ci est donc prête à être lue par le processeur.
- En émission : qu'un circuit ou périphérique est prêt à recevoir une donné.
- La fin d'une conversion analogique numérique, la donnée pouvant alors être lue
- La frappe d'une touche de clavier.
- La fin d'un simple intervalle de temps (généré par circuits Timers)
- Une cadence (générée par Timers)
- Une procédure prioritaire ou d'urgence
- Etc…..

7.1. Sondage ou Interruption ?

Il y a **deux grandes techniques :**

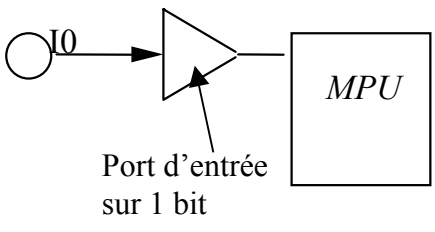
- **Sondage :**
  Boucle d'attente avec instructions de lecture de détection de ou des événements. Le programme ***s'attend donc à recevoir*** ces évènements.
- ***Interruption :***
  Simple boucle d'attente sans rien faire d'autre ou totalement autre chose. Une interruption déclenchée par un évènement quelconque provoque alors automatiquement le déroutement et le saut au programme d'interruption correspondant.
  Le processeur ***ne s'attend donc pas*** aux événement.

7.1.1. Événements signalés par une simple ligne logique

Le signal actif peut être un front (donc une transition) ou simplement le niveau.

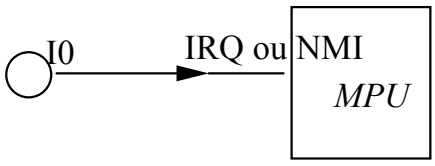
- Signal logique **lu** sur un **simple bit** d'un port d'entrée

Sondage obligatoire !



- Signal **câblé directement** sur **une ligne d'interruption** matérielle **du processeur**.

Interruption obligatoire !



7.1.2. Événements détectés par des circuits périphériques quelconques, drapeaux.

C'est le cas de circuits tels que interfaces d'entrée sortie parallèle ou série, convertisseurs analogique numérique, Timers, …. internes ou externes au processeur.

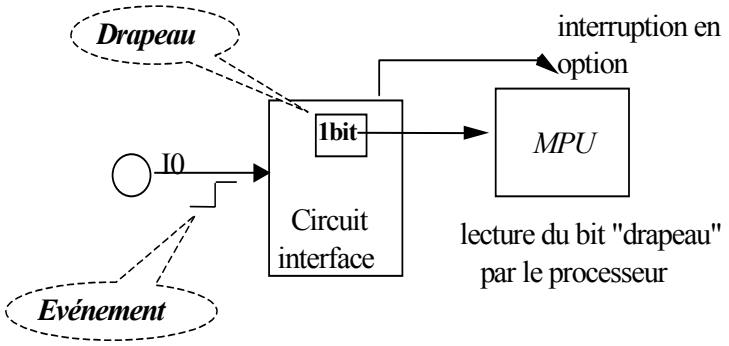
Ils sont pratiquement tous configurables et les **événements** (front actif sur une ligne de dialogue, fin de conversion ….) **font passer à 1** un **bit** nommé **drapeau** qui leur est associé. Ces bits sont situés dans les « **registres d'états »** (Status) des circuits.

On peut **valider ou non** les i**nterruptions** correspondantes (au moyen de **bits particuliers** de leur « **registre de contrôle »**.

Les **deux modes** sont ainsi **possibles**.

- **Exemple : Signal** câblé sur une **ligne de dialogue** d'un port d'entrée sortie.

Interruption ou sondage de drapeau.



7.1.3. Remise à zéro des drapeaux

Très important :

Que l'on travaille en sondage ou en interruption, Il faut toujours penser à **remettre un drapeau à zéro**. Ceci est essentiel, sinon le programme « plante » (détectant alors indéfiniment l'évènement …). Cette remise à zéro, selon le circuit, peut être :

- Automatique
  - S'effectuer par lecture ou écriture du port associé.
  - S'effectuer par écriture de 1 sur le drapeau lui même.
  - D'autres cas sont possibles

Il faut toujours bien lire la documentation sur le circuit utilisé ! ! !

- ***Attente du premier événement***

Il faut parfois différencier la détection d'un **front** ou d'un **niveau**, un événement étant signalé par une **transition** sur une ligne ou un bit.

La première fois, il faut alors remettre à zéro le drapeau correspondant juste avant de se placer en boucle d'attente, ou de valider une interruption.

Dans le cas contraire le test peut être immédiatement positif si un front antérieur à eu lieu. Il peut en en résulter des dysfonctionnements comme par exemple l'acquisition du premier caractère faux sur une ligne de transmission série.

7.2. Sondage

Le périphérique fournit un signal actif sur une ligne ou sur un drapeau.

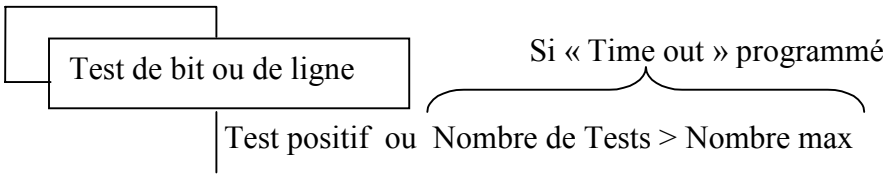
Le processeur doit régulièrement scruter la ligne ou le drapeau pour s'en rendre compte.

Si le test est positif:

- Il effectue la tâche correspondante.
- Il devra **remettre à zéro le drapeau** si ce n'est pas automatique (sinon prise en compte multiples)

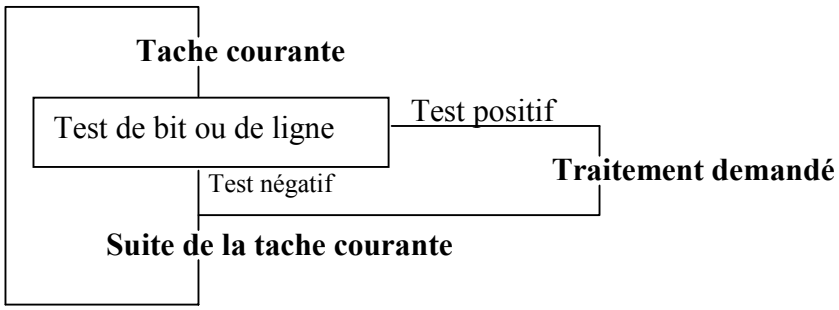
Deux possibilités :

- ***Simple boucle d'attente:***



Sans « Time out », le processeur attend, en ne faisant rien d'autre le bit testé. Si la demande n'arrive jamais, le processeur est bloqué. Une technique simple (nommée gestion de « **time out** ») est de programmer un compteur dans la boucle d'attente: au bout d'un certain nombre de tests, on peut ainsi sortir de la boucle.

- ***Lecture au vol :***



Le processeur peut ainsi faire une autre tache répétitive, tout en testant de temps en temps la demande. Exemple scrutation d'un clavier tout en exécutant une tache principale.

Conclusion :

La technique par « sondage » est simple, la prise en compte est synchrone de la tâche principale du processeur.

Inconvénients: Temps de réponse parfois longs dans le cas de sondages peu fréquents.
Il faut programmer un test qui s'attend donc à tel ou tel évènement.
Blocage si on ne programme pas de Time out.

7.3. Interruption

7.3.1. principe

Le processeur interrompt sa tâche principale et peut ainsi très rapidement réagir. L'initialisation au préalable d'un « **vecteur d'interruption** » permet au processeur de se savoir où se brancher. On distingue deux types de vecteurs :

- ***Vrais Vecteurs :***

Le processeur lors d'un départ en interruption **va chercher l'adresse de branchement** à une adresse bien précise, située dans la **table des vecteurs** d'interruption.

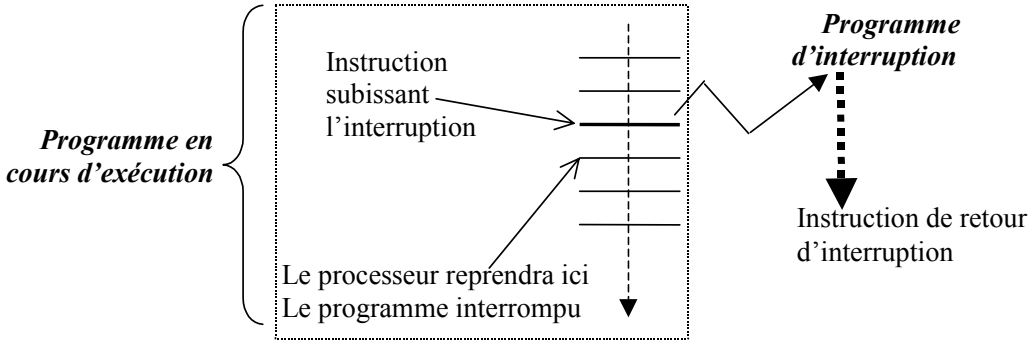
C'est le cas de L'HC11,HC12, du 68000 …

- ***Pseudo vecteurs :***

Le processeur **va exécuter l'instruction** située à une adresse bien précise située dans une table. Cette instruction doit être une instruction JMP de saut vers la routine correspondante d'interruption. Bien que l'on parle souvent aussi de vecteurs La table est en fait une table de Jump.

C'est le cas par exemple des processeurs de signaux Texas …

Le programme interrompu est automatiquement repris juste à l'endroit où il a été interrompu, dès l'instruction de retour d'interruption (RTI en assembleur HC11,HC12).



La plupart des interruptions sont de type MASQUABLES. Ce qui permet de les valider seulement si on en a besoin.

Dans le cas d'interruption à priorité maximale (procédures d'urgences par exemple), on peut se servir de quelques interruptions NON MASQUABLES.

Les **drapeaux éventuels** se positionnent comme précédemment, et peuvent servir à reconnaître d'où provient l'interruption dans le cas de plusieurs lignes d'interruption reliées sur une ligne unique, et donc vecteur identique.

Veiller à leurs remise à zéro si ce n'est pas automatique !

Avantages :

Le processeur **ne s'attend pas à cette demande**, aucune instruction d'attente de dialogue n'est à programmer. Il suffira une fois pour toute de programmer des bits de certains registres (coté microcontrôleur et coté interface) pour travailler ainsi.

La réponse est la plus **rapide** possible (on parle aussi de **préemption**).

7.3.2. Déroulement d'une interruption niveau processeur

Dès qu'une interruption est détectée par le processeur, celui ci part en interruption :

- Il termine l'instruction en cours d'exécution. (niveau code machine).
- Il positionne à **1** le bit **I masque général des interruptions.**
- Il sauve en pile son compteur programme et son code condition (au minimum) et souvent tous ses registres (sinon le programmeur devra sauver les registres qu'il utilise, ce qui est automatique si on programme en C)
- Il va chercher où se brancher dans sa **table de vecteurs** d'interruptions (voir prochain chapitre).
- Le masque général **I** se remet toujours **automatiquement** à **0** lors du retour d'interruption, permettant éventuellement un nouveau départ.

7.3.3. Préparation pour un travail en interruption

Pour que le processeur puisse partir en interruption, il faut préparer la ligne ou le circuit à travailler ainsi, dans une phase d'initialisation.

Important : Il faut toujours initialiser les 'vecteurs d'interruption' avant les validations sinon risque de plantage du programme !

➢ ***lignes d'interruptions non masquables du processeur***

Exemple **NMI** en HC11, **RESET**.

1) Initialiser le vecteur d'interruption (souvent presque automatique en C).
2) Et c'est tout, par définition l'interruption est toujours validée.

Si on ne fait rien, être sûr de ne jamais avoir de signaux parasites sur cette entrée, sinon plantage !

➢ ***lignes d'interruptions masquables du processeur***

Exemple sur HC11,HC12 :

IRQ, **XIRQ**, masquables par respectivement le bit **I** (qui est aussi le masque général) et le bit **X,** situés dans le registre Code Condition, où se trouvent aussi les bits résultats d'opérations arithmétiques classiques N, V, Z, C . Au Reset, les interruptions sur ces lignes sont masquées.

1) Initialiser le vecteur d'interruption (souvent presque automatique en C).
2) M**ettre à zéro** le **masque** d'interruption.
Instruction assembleur : CLI met I à 0, donc valide l'interruption
Remarque : XIRQ devient non masquable dès une première validation.

➢ ***Interruptions masquables des circuits périphériques***

1) Initialiser le vecteur d'interruption (souvent presque automatique en C).
2)**Valider l'interruption** correspondante à l'événement, **au niveau du circuit périphérique.**
Si plusieurs bits de validation se trouvent dans le même registre, pour ne modifier que les bits concernés il est souvent pratique de se servir des opérateurs logiques ET (&) et OU (|).
3) M**ettre à zéro** le **masque I général** d'interruption (si ce n'est pas déjà fait par ailleurs).

7.3.4. Interruptions prioritaires, contraintes de temps réel

Le masque général **I** se **repositionne à 1 automatiquement** lors du **départ** en interruption, interdisant ainsi des interruptions sans cesse, mais aussi toutes autres interruptions de même priorité. Il ne repasse à zéro que lors de l'instruction de retour d'interruption.

En conséquence : quand un processeur part en interruption, il ne peut plus être interrompu, ou si il possède plusieurs niveaux d'interruption, ne peut être interrompu que par une ligne à priorité supérieure.

Pour les **applications** nécessitant des **réponses très rapides** à des interruptions (**contraintes** de **temps réel**) :

- Si on dispose de processeur à plusieurs niveaux possibles, il faut les utiliser (ou les programmer en conséquence)
- Sinon, il faut ruser. On peut en effet rendre une tâche d'interruption absolument prioritaire sur d'autres interruptions :

Il suffit pour cela de remettre à zéro le masque **I** principal d'interruption à l'entrée de toutes les autres tâches non prioritaires (après avoir remis leur drapeau à 0, sinon elles s'interrompraient sur elles mêmes sans cesse …. !).

On rend ainsi leur routine d'interruption de nouveau interromptible par la tâche prioritaire !

Interruption ***Non prioritaire***

Remise à zéro du drapeau correspondant
Masque général **I** remis à **zéro** (CLI)
…..
…..
…..
rts

Attention cependant que la première tache interrompue ne soit pas de nouveau interrompue par elle même, il y aurait alors plantage du système !

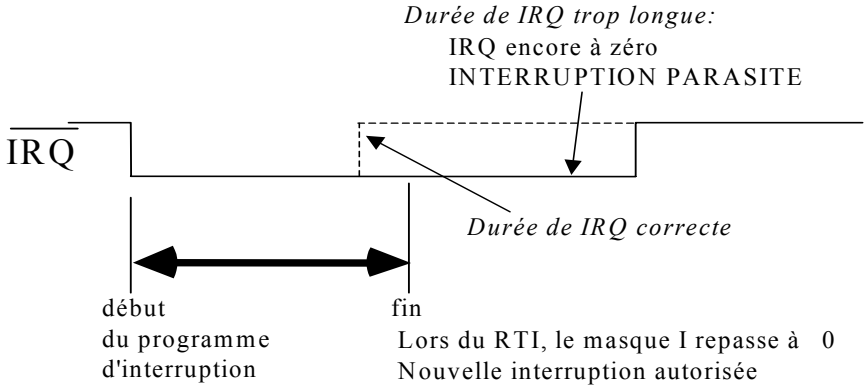
Une technique ou les interruptions sont utilisées systématiquement est le « multi-tache en temps partagé, avec contraintes temps réel ». Un chapitre est consacré à ce sujet.

7.3.5. Précautions d'emploi des lignes d'interruption du processeur lui même

7.3.5.1. Sur niveau (exemple IRQ de l'HC11)

- L'usage direct est assez délicat :

Il faut en effet éviter des interruptions multiples parasites pouvant survenir si l'impulsion $\overline{IRQ}$ est trop large, comme le montre la figure ci dessous.



- L'usage de cette ligne par l'intermédiaire de circuits extérieurs avec drapeau élimine ce problème.

La remise à zéro du drapeau, indispensable avant le retour d'interruption, provoque justement le passage de $\overline{IRQ}$ à 1.

7.3.5.2. Interruptions sur front (NMI, XIRQ de l'HC11,HC12)

Le problème précédent ne se pose pas, puisque c'est seulement le front qui déclenche l'interruption. Attention toutefois aux rebonds possibles, le front doit être un bon front logique.

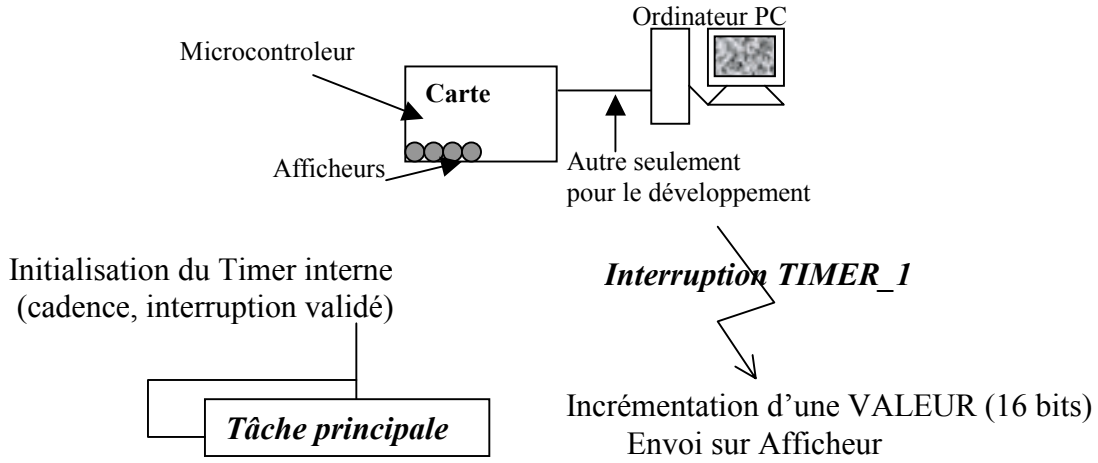
8. LES VECTEURS D'INTERRUPTION MISE AU POINT ET APPLICATION FINALE

Nous avons déjà parlé un peu de l'interruption Reset. Nous voyons ici le cas général et les problèmes dues au développement d'une application.

Il nous faudra distinguer :
La **phase** de **mise au point** de la carte (Debug), avec ordinateur et logiciel de développement associé.
La **phase finale** (le produit fini) ou l'application doit démarrer à la mise sous tension et est autonome.

Nous prendrons l 'exemple tout simple ci-dessous, avec :

- Une tâche principale
- Une interruption interne (par Timer, programmée à cadence régulière).



Il existe d'autre part deux types de microcontrôleurs et de cartes, nous allons les étudier à tour de rôle.

8.1. Développement sur Microcontrôleurs nécessitant un « Moniteur » : exemple l'HC11

C'est le cas de nombreux microcontrôleurs et cartes de développement associées.

8.1.1. Rôle du Moniteur en phase mise au point

Un programme situé en ROM ou EPROM, nommé « moniteur » fourni par le constructeur de la carte ou du compilateur, est le plus souvent indispensable pour communiquer avec l'ordinateur de développement, sinon on ne peut rien faire !

- **Le vecteur Reset** est donc en **Mémoire non volatile et lance ce moniteur** à la mise sous tension.
- On peut alors charger un programme complet en RAM, démarrer l'exécution à une adresse, mettre au point (Debug) par points d'arrêt, pas à pas, examen des registres des contenus mémoires …
- **Les autres vecteurs** d'interruptions doivent **pouvoir être modifiés**.

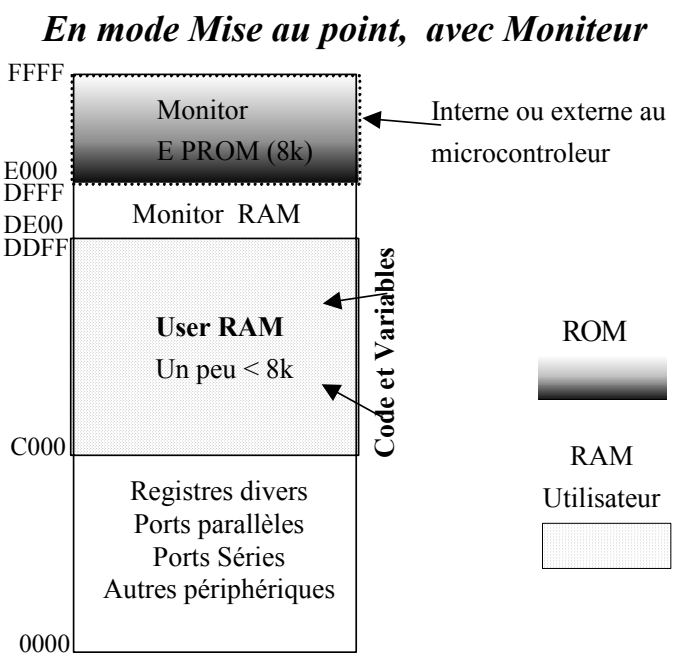
8.1.2. Exemple de plan mémoire (ici l'HC11)

8.1.2.1. En mise au point

Exemple de plan mémoire avec les vecteurs en haut du domaine adressable.

En mise au point :

→ Le moniteur est lancé à la mise sous tension, et permet de communiquer avec l'outil de développement.
→ On charge **code et variable** en **RAM**, tout étant modifiable.



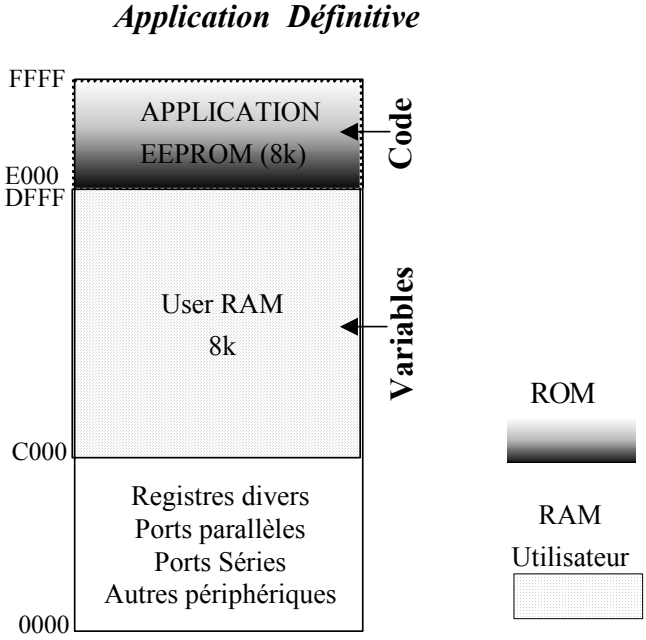
8.1.2.2. Application finale

Pour l'application finale :

→ Le **moniteur est à priori inutile**
(Dans certains cas on peut toutefois conserver quelques routines de celui ci)

→ **L'application finale** est lancée à la **mise sous tension**

→ On charge **code et constantes en mémoire non volatile (ici EEPROM).**
→ **Variables en RAM**



8.1.3. Nécessité d'une table de JUMP en RAM en mise au point

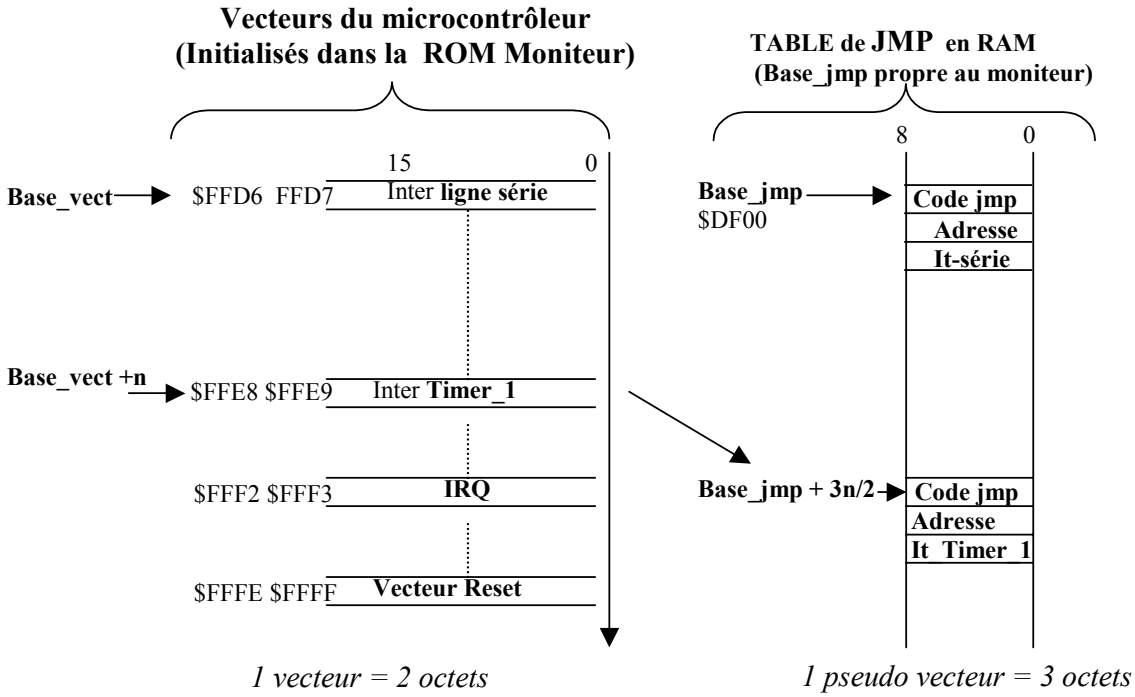
Le vecteur Reset est en mémoire non volatile ROM ou EPROM (moniteur) pour lancer celui-ci à la mise sous tension. Or, le vecteur Reset fait partie de la table des vecteurs, donc toute celle ci est en Mémoire non volatile.

On veut pouvoir tout de même modifier les vecteurs en mise au point !

Remède : En mode mise au point, les vecteurs du processeur sont tous initialisés dans la ROM moniteur et renvoient vers une **table de JMP** en RAM que l'utilisateur peut changer.

Cette table de jump dépend du processeur évidemment, mais aussi du moniteur.

Exemple :



Exemple **si** la **table de JMP** commence en **$DF97**

Pour l'interruption Timer_1 de vecteur $FFE8 $FFE9, donc de décalage n = $FFE8-$FFD6 = $12 ou 18d, le **jmp** correspondant se trouve en :

$DF97 + 3.18/2 = $DF97 + 27 = $DF97 + $1B = **$DFB2**

8.1.4. Programmation tout assembleur (code HC11)

8.1.4.1. Programmes assembleur

8.1.4.1.1. PROGRAMME PRINCIPAL

```
                RSEG            STACK
sommet_pile     equ     *       label sommet pile placé

                RSEG            VARIABLES
                Public valeur   VALEUR à incrémenter (globale)
                valeur          rmb     2

                RSEG            CODE
debut           lds # sommet_pile   initialisation obligatoire de S
                ……              Initialisation du Timer_1 (avec son interruption validée)
                cli             Masque général des IT à 0
boucle          ……..            Tache principale sans fin
                bra             boucle
```

8.1.4.1.2. ROUTINE D'INTERRUPTION

Sur l'HC11,HC12, les registre A et B (comme tous les registres) sont sauvés automatiquement en pile et restitués lors de l'interruption, donc pas de sauvegarde à programmer soi même.

```
it_ timer_1     Raz drapeau  (selon Timer utilisé)
                Remise de la bonne tempo timer   (selon Timer utilisé)
                LDD     valeur
                ADDD    #1
                STD     valeur
                Jsr ….          'routine d'affichage de D'
                RTI             Retour d'interruption
```

8.1.4.2. Initialisation des interruptions

8.1.4.2.1. EN MISE AU POINT

Celle ci peut s'effectuer apparemment soit dès le départ (directives assembleur) à la création du code final téléchargeable, ou à l'exécution (instructions assembleur).

- En exécution on écrirait (Code HC11,HC12)

Idée N01
A Eviter !

```
LDAA  #7E              Code du jmp
STAA  $DFB2
LDD  #it_timer_1       routine d'interruption du timer1
STD  $DFB3
```

Avantage : on peut même en exécution modifier plusieurs fois si besoin est les vecteurs.
Inconvénient: La table de jump devra être conservée même pour l'application définitive. En effet, pour l'application finale, à aurait envie d'écrire directement l'adresse de it_timer_oc1 dans le vecteur initial, cela est impossible car des instructions d'écriture simple ne peuvent pas programmer une mémoire EEPROM ou FLASH et encore moins une ROM ou PROM !

- **Au téléchargement** par des directives ORG (ou nom de section) et FCB et FDB.

On ajouterait donc dans notre programme assembleur :

```
ASEG          (Section absolue)
ORG $DF97
FCB  $7E      Code du Jmp
FDB it_timer_1
```

Et souvent pour que le Debugger démarre correctement en début de programme, on initialise le vecteur Reset vers celui ci, comme on le ferait pour l'application finale. On ajoute donc :

```
ORG $FFFE
FDB debut
```

8.1.4.2.2. MODIFICATION POUR APPLICATION FINALE

Au téléchargement obligatoirement !

```
ASEG          (Section absolue)
ORG  $FFF8    Suppression de la table de Jump inutile !
FDB  it_timer_1    Directement
ORG $FFFE
FDB  debut
```

On peut aussi regrouper tous les vecteurs dans une section Absolue à placer par l'édition de lien, avec une seule indication ORG , et des FDB 0 (ou RMB …) servant juste à laisser de la place pour les interruptions non utilisées.

8.1.4.3. Fichier de Link (IAR Systems) tout assembleur

8.1.4.3.1. PHASE DE MISE AU POINT

```
-c68hc11                                        // Se reporter aux plans mémoire !
-!          SECTIONS de type  CODE  de C000 à CFFF      -!
-Z(CODE)CODE,CONSTANTES   =   C000-CFFF
-!     Vecteurs d'interruption du moniteur, garder le nom !     -!
-Z(CODE)INTVEC=FFD6
-!          SECTIONS de type  DATA   de D000 à DDFF      -!
-Z(DATA)VARIABLES   =      D000-DDFF
-Z(DATA) STACK=FF
```

8.1.4.3.2. APPLICATION FINALE

```
-c68hc11                                        // Se reporter aux plans mémoire !
-!          SECTIONS de type  CODE       A la place de la ROM Moniteur    -!
-Z(CODE)CODE,CONSTANTES   =   E000-FFFF
-!     Vecteurs d'interruption du moniteur, garder le nom !     -!
-Z(CODE)INTVEC=FFD6
-!          SECTIONS de type  DATA     Toute la RAM  possible      -!
-Z(DATA)VARIABLES =        C000-DFFF
-Z(DATA) STACK=FF
```

8.1.5. Programmation en C: Interrupt Handler

Les compilateurs C pour microcontrôleurs permettent une programmation très aisée des interruptions et le passage de la mise au point à l'application finale.

8.1.5.1. Principe et utilisation de fonctions d'interruptions

On peut définir ce type de fonction pour tout type d'interruption.

- Cette fonction :
  - Ne doit pas être mise en prototype
  - Travaille uniquement sur **des paramètres Globals**. Pas de passage de paramètres ni de valeur retournée. Peuvent avoir des variables locales comme toutes autres fonctions.
- Le compilateur **se charge tout seul** de presque tout :
  - De la gestion automatique d'une table de Jump si nécessaire en mode mise au point
  - De l'initialisation des vecteurs initiaux ou de cette table de jump, donc aussi bien pour la mise au point que pour l'application finale (***).
  - Des instructions supplémentaires de sauvegardes éventuelles de registres qui ne seraient pas sauvés automatiquement par l'interruption.
- Pour passer de la mise au point à l'application finale :
  - Il suffit alors juste de changer les implantations des sections de type code et de type données au niveau de l'édition de lien.
  - De modifier si besoin est l'option du format du fichier final (cas d'envoi vers un programmateur universel par exemple).
  - Les modifications au niveau des vecteurs d'interruption (Reset et autres) s'effectuent toutes seules !

- **Très pratique** !
- *** ***Attention*** *:* Certains Interrupt Handler (de plus en plus rare) n'initialisent pas les vecteurs ni la table de jump. Le faire alors soi même en assembleur !

La notation dépend beaucoup du compilateur. On utilise :

- Soit des **noms réservés** et spécifiés dans le compilateur (Exemple c_INT0, C_INT1, C_INT2 ….)
- Soit **une syntaxe à respecter** avec un **nom optionnel**, et une **valeur** situant l'interruption concernée dans la table des vecteurs.

Pour IAR, une fonction d'interruption nommée par exemple **inter_1** s'écrirait ainsi :

```
Interrupt[décalage par rapport à Adresse_base_ vecteurs]   void inter_1(void)
{
…….
}
```

Cet outil ne dispense pas de savoir comment ça marche réellement si il y un problème … !

8.1.5.2. Exemple précédent (Notation compilateur IAR)

On rappelle l'**adresse_base_ vecteurs : $FFD6** (Table initiale du microcontrôleur).

8.1.5.2.1. PROGRAMME PRINCIPAL

```
unsigned int     valeur                          /* en variable globale */
main()
{
……..    // Initialisation du Timer avec interruption
While(1) {
         ……. Tache principale sans fin
         }
}
```

8.1.5.2.2. FONCTION D'INTERRUPTION (HC11)

```
Interrupt[18]   void  timer_1(void)          // ou 0x12  ou  0xFFE8-0xFFD6
   {
remise de la bonne tempo dans le Timer    (Voir Timer)
Raz drapeau   (voit Timer)
Valeur+=1 ;
Afficher(valeur) ;                          // Fonction d'affichage
   }
```

8.1.5.2.3. FICHIER DE LINK EN MISE AU POINT (IAR SYSTEM)

```
-c68hc11
   - !                          Sections de type CODE                    - !
-Z(CODE)RCODE,CODE,CDATA0,CDATA1,CONST,CSTR,CCSTR=C000-CFFF
-Z(CODE)INTVEC=FFD6
   - !                          Sections de type DATA                    - !
-Z(DATA)IDATA1,UDATA1,ECSTR,WCSTR,TEMP,CSTACK+200=D000-DDFF
-Z(DATA)IDATA0,UDATA0=00-FF
cl6811
```

8.1.5.2.4. FICHIER DE LINK POUR L'APPLICATION FINALE

```
-c68hc11
   - !                          Sections de type CODE    a la place du Moniteur          - !
-Z(CODE)RCODE,CODE,CDATA0,CDATA1,CONST,CSTR,CCSTR=E000-FFFF
-Z(CODE)INTVEC=FFD6
   - !                          Sections de type DATA    possible toute la RAM           - !
-Z(DATA)IDATA1,UDATA1,ECSTR,WCSTR,TEMP,CSTACK+200=C000-DFFF
-Z(DATA)IDATA0,UDATA0=00-FF
cl6811
```

On ne s'occupe donc à priori pas de l'initialisation des vecteurs !

En mise au point, une table de Jump était gérée.

L'option **Génération de fichier Motorola S1S9** destiné à un programmateur de PROM, ou l'option code sans Debug (selon les compilateurs) supprime automatiquement la table de JMP devenue inutile, les vecteurs sont écrits pour être chargés en mémoire non volatile.

8.2. Développement sur Microcontrôleurs sans « Moniteur »

8.2.1. Introduction

Quelques processeurs (comme l'HC12) permettent la mise au point complète par un port spécial nommé **BDM** (**B**ack Ground **D**ebug **M**ode). Le moniteur n'est donc plus utile.

L'ordinateur de développement est alors relié à ce port par l'intermédiaire d'une **sonde BDM spéciale** (Hélas plus onéreuse que la carte de développment … !).

Ces microcontrôleurs possédant d'autre part de la mémoire **FLASH** interne.

8.2.1.1. Phase de mise au point. Table de Jump inutile

→ PAS DE TABLE DE JUMP

En **mise au point** comme pour **l'application finale**, les **vecteurs** du processeur seront donc toujours initialisés au téléchargement dans la mémoire **FLASH**.

→ Option N°1 : Code et Variables en RAM

C'est le cas usuel, à part les vecteurs, on charge tout en RAM.

Intérêt : Le chargement est plus rapide.
La mise au point (pas à pas, points d'arrêt …) toujours possible.

→ Option N°2 : Code en FLASH et **variables en RAM**. Comme pour l'application final !

Intérêt : Nécessaire pour de gros programmes si la taille de la RAM n'est pas suffisante. Les microcontrôleurs actuels (comme l'HC12) possèdent une mémoire FLASH de taille importante, même supérieure à 64k, et des possibilités de travailler par pages.

Inconvénient : Certains Outils ne permettent pas forcément la mise au pont avec le code en FLASH….

8.2.1.2. Phase application finale

Option N°1 : Il suffit de changer l'emplacement mémoire du CODE.

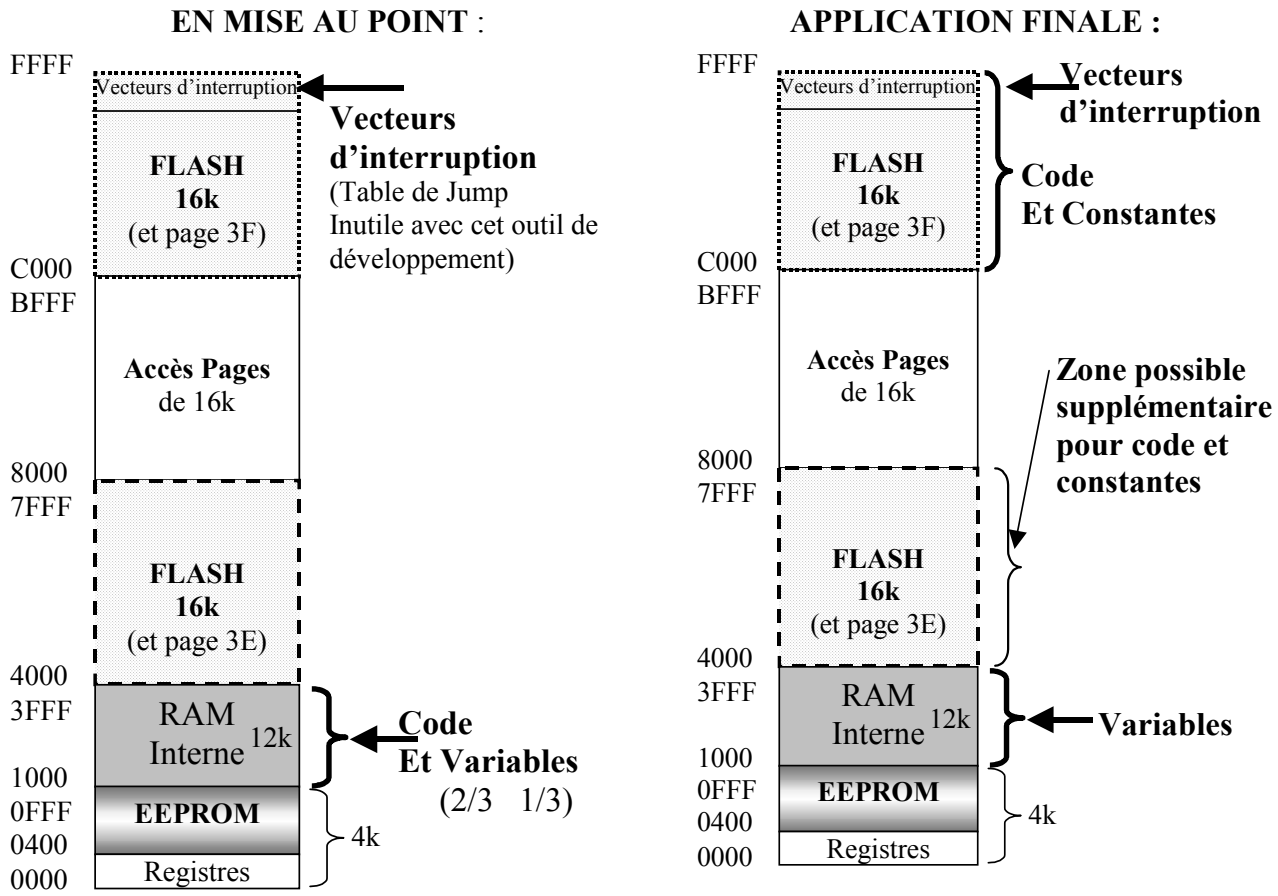
Option N°2 : Aucune modification à faire, on travaille finalement directement sur l'application finale en FLASH !

8.3. Exemple sur microprocesseur type HC12, en C, sans moniteur, avec sonde BDM

8.3.1. Vecteurs d'interruption de l'HC12

Se reporter à l'annexe.

8.3.2. Plan mémoire



8.3.3. Le programme précédent, adapté pour HC12

```
unsigned int        valeur                          /* en variable globale */
main()
{
……..        // Initialisation du Timer avec interruption
While(1)    { ….  Tache principale sans fin  }
}
Interrupt[0xFFEE-0xFF80]   void  timer_1(void)        // pour TC0, ici base_vect = FF80
{
remise de la bonne tempo dans le Timer    (Voir Timer)
Raz drapeau   (voit Timer)
Valeur+=1 ;
Afficher(valeur) ;                // Fonction d'affichage
}}
```

8.3.4. fichier de commande de l'éditeur de lien

On le nomme par exemple : LNK**_CML12S_C.XCL**

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

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

8.3.4.1. Mise au point, tout en RAM (Sauf vecteurs)

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

-Z(CODE)INTVEC=FF80-FFFF        // Vecteurs  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'HC12 !
-Z(DATA)DATA0,IDATA0,UDATA0=FFFFFFFF-FFFFFFFF

// La librairie C fournie par le constructeur (contient au minimum ici  le Start up du C)
cl6812
```

8.3.4.2. Application finale tout en FLASH

```
-c6812                  // Type de micro
//                      Sections de type CODE
-Z(CODE)CDATA0,CDATA1,CCSTR= C000-FF7F     // (ou 4000_7FFF)
-P(CODE)RCODE,CODE,CONST,CSTR,CHECKSUM= C000-FF7F // (ou 4000_7FFF)

-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
                                                        (ou toute la RAM: 1000-
3FFF)
// Non utilisé en page 0 sont les registres de l'HC12 !
-Z(DATA)DATA0,IDATA0,UDATA0=FFFFFFFF-FFFFFFFF

// La librairie C fournie par le constructeur (contient au minimum ici  le Start up du C)
cl6812
```

Remarque : nous l'avons déjà vu, ce fichier peut même servir en mise au point si le Debugger le permet …., mais le téléchargement constamment en FLASH est plus long, et fatigue peut être la FLASH à la longue ?

9. PORTS PARALLELES

Ils peuvent être internes ou externes au microcontrôleur. Nous prendrons comme exemple les ports internes de l'HC12.

Les ports parallèles comprennent toujours quelques lignes pouvant être utilisées comme entrées matérielles d'interruption, avec drapeau associés. Certaines mêmes peuvent gérer automatiquement certains dialogues (génération automatique d'impulsion à chaque écriture ou lecture, Hand Shake….).

Les **mêmes lignes** servent aussi souvent à **plusieurs fonction**s. Par défaut au Reset, toutes les lignes sont des lignes générales d'entrée-sortie standards initialisées en entrée. Si on valide d'autres fonctions (Ports Série, sortie Timer, Bus IIC ….), les lignes correspondantes perdent alors leur rôle de lignes IO.

9.1. Utilisation en simple port IO

Un port IO se programme au minimum avec deux registres :

Le registre de **donnée** proprement dit, par exemple : **PORT**

Le registre de **direction** associé, par exemple : **DDR** (**D**ata **D**irection **R**egister)

On peut programmer individuellement chaque ligne en entrée, ou en sortie au moyen de ce registre : **0** **Entrée**

1 **Sortie**

Au départ, un port est en entrée. Si l'on veut démarrer correctement avec par exemple les 4 lignes de faible poids en sortie et à zéro :

```
PORT  = 0x00 ;
DDR   = 0x0F ;
```

Cet ordre est préférable sinon, si par hasard des lignes (internes du PORT sont à 1), écrire en premier DDR = 0x0F ; peut générer une impulsion parasite gênante parfois.

Sur certains processeurs comme l'HC12, on peut aussi au moyen d'autres registres programmer des résistances de Pull up ou Pull down, forcer une puissance réduite en sortie pour diminuer la consommation …..

9.2. Utilisation possibles de certaines lignes en entrées d'interruption

Certaines lignes d'un port, ou des lignes spéciales, peuvent se programmer en entrée d'interruption.

On peut choisir le front actif, parfois aussi des Pull up ou Pull down.

On valide l'interruption de la ligne désirée

Puis les interruptions générales comme d'habitude.

Chaque ligne possède un drapeau associé.

Le vecteur d'interruption peut être commun pour plusieurs lignes, on test alors la provenance au moyen du drapeau.

9.3. Le Port Integration Module de l'HC12, résumé

Les ports parallèles sont situés dans un module interne à l'HC12, nommé le « Port Integration Module ».Ce module assure la liaison entre les différents composants internes et les broches du circuit.

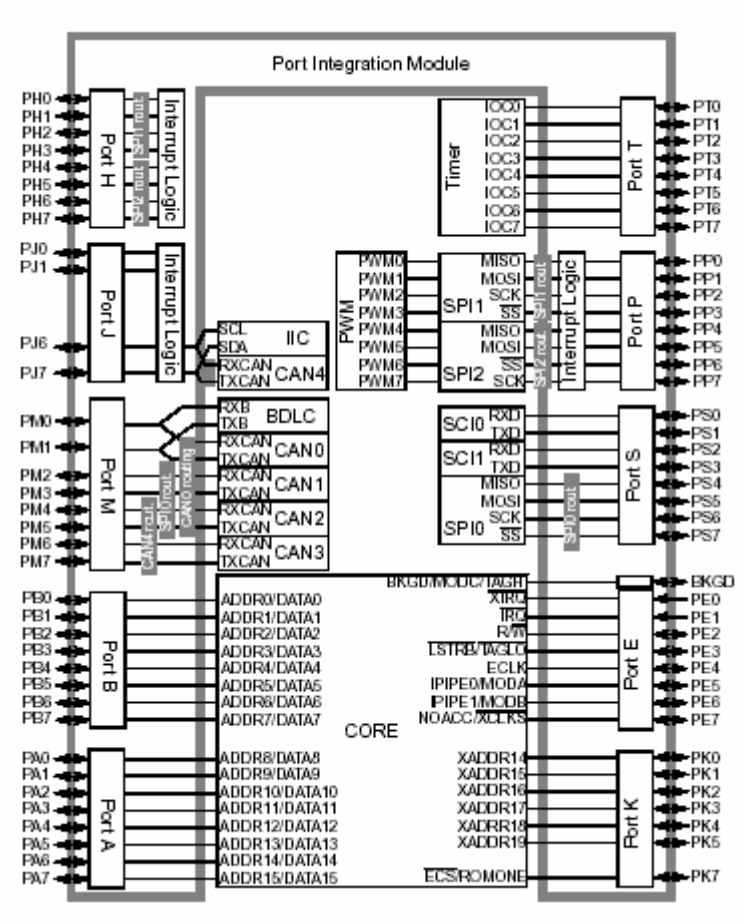
Schéma général :

Adresse de base : $240

On voit donc que chaque port servent à plusieurs fonctions différentes.

En Mode étendu, PORTA PORTB PORTE et PORTK sont réservés à la gestion du Bus externe (Adresses, Données sur 8 ou 16 bits, sélection des pages, signaux de contrôle …). Ces ports seront donc inutilisables en entrées sorties classiques.

Remarque :
La partie convertisseur Analogique Numérique fait partie d'un autre module interne.



Dés que l'on valide sur certaines lignes des fonctionnalités autres que le port parallèle de base, on inhibe alors automatiquement les entrées-sorties à usage générale correspondantes, et leurs registre de base deviennent sans objet.

9.4. Ports parallèles d'entrée-sortie (I/0) en usage général que l'on utilisera sur notre maquette

Remarque : Lors de la première initialisation des différents bits, on peut directement écrire 0 dans les bits non utilisés (soit c'est la valeur par défaut, soit écrire 0 est sans effet).

9.4.1. Port T : 8 bits I/O et Timer

Chaque Bit se programme individuellement :

PTT
$240

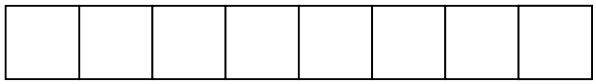
Port I/0 8 bits

DDRT
$242

Registre de Direction
0 Bit en entrée
1 Bit en sortie

Data Direction Register

PERT
$244



Pull Enable Register

Validation de Résistances de Pull up ou Pull down (Pratique ! !)

PPST
$245

Pull Polarity Select

Type de Pull
Si validé dans PERT:
0 Up
1 Down

Remarque : il existe aussi un registre **RDRT** (**R**educe **D**rive **R**egister), pour diminuer la consommation du port programmé sortie en diminuant ses possibilités de sortance (courant max disponible).

→ Sur la maquette :

Diverses lignes sont utilisables en entrée ou en sortie, en gestion du Timer et du Pulse accu. Elle possède un connecteur supplémentaire interne. Voir le chapitre concerné.

9.4.2. Port H : 8 bits I/0 et Interruptions

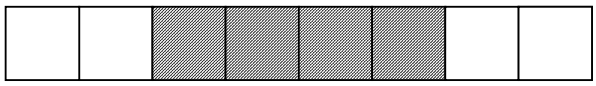
PTH $260 **Port I/0 8 bits**
DDRH $262 Registre de Direction
PERH $264 Validation de Résistances de Pull up ou Pull down

→ Réservé sur la maquette au panneau cristaux liquides.

Possibilités de lignes d'interruption non utilisables sur la maquette.

9.4.3. Port J : 2 bits I/0 et Interruptions + 2 bits IIC, interruptions,CAN4

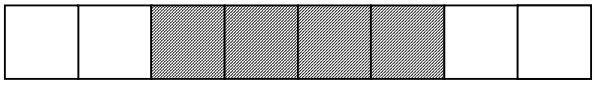
PTJ
$268



CAN4 et IIC

Port I/0 2 ou 4 bits

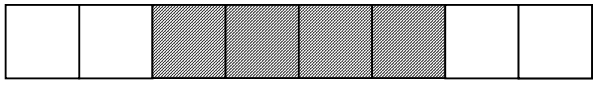
DDRJ
$26A



Data Direction Register

Registre de Direction
0 Bit en entrée
1 Bit en sortie

PERJ
$26C

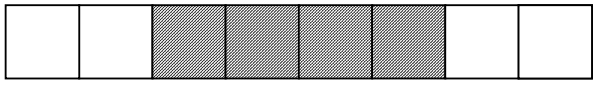


Pull Enable Register

Validation de Résistances de Pull up ou Pull down

(PRATIQUE ! !)

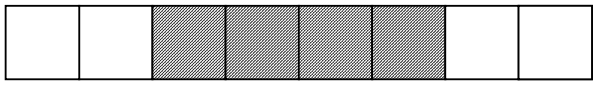
PPSJ
$26D



Port Polarity Select

Type de Pull et de Front d'interruption
(Pull si validé)
0 pull Up Front ↓ **Actif**
1 pull Down front ↑ **actif**

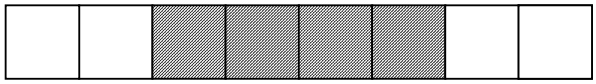
PIEJ
$26E



Port Interrupt Enable

Port Interrupt Enable
0 inhibé
1 Validé
Classique : logique positive

PIFJ
$26F



Port Interrupt Flag
→ ***Ecrire 1*** pour Remettre à zéro le drapeau

Port Interrupt Flag
0 inhibé
1 Validé
(Classique, logique positive)

→ On se servira de : PJ0, PJ1 : Entrées binaires ou interruption
PJ1 : interruption lignes L3 L4 clavier
PJ7,PJ6 : Bus IIC vers double CNA

9.4.4. Port M : 8 bits I/0, bus CAN, bus BDLC

Registres classiques :

PTM	$250	**Port I/0 8 bits**
DDRM	$252	Registre de Direction
PERM	$254	Validation de Résistances de Pull up 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**.

9.4.5. Port S : 8 bits IO Port Série SCI0, Port série rapide SPI0

Registres identiques :

PTS	$248	**Port I/0 8 bits**
DDRS	$24A	Registre de Direction
PERS	$24C	Validation de Résistances de Pull up ou Pull down
PPSS	$24D	Si validé dans PERS, Type de pull (0 up, 1 down)

→ On ne se servira que du bit PS2 pour les capteurs de température.

9.4.6. Port P : 8 bits I/0, PWM,SPI

Registres identiques :

PTP	$258	**Port I/0 8 bits**
DDRP	$25A	Registre de Direction
PERP	$25C	Validation de Résistances de Pull up ou Pull down
PPSP	$24D	Si validé dans PERP, Type de pull (0 up, 1 down)

→ Disponible sur connecteur supplémentaire interne
→ Sinon on ne se servira que du bit PP0 en sortie binaire ou PWM0.

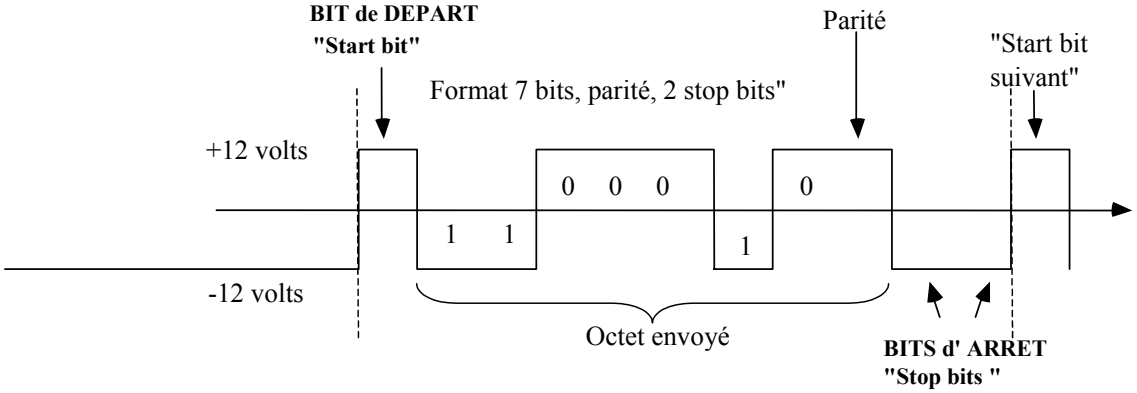
10. PORT SERIE ASYNCHRONE TYPE UART

10.1. Petit rappel sur la liaison série asynchrone

Avantage de la transmission série: peu de fils de liaison
Mais: débit relativement lent.
Protocole nécessaire pour ralentir l'émetteur si le récepteur est trop lent.

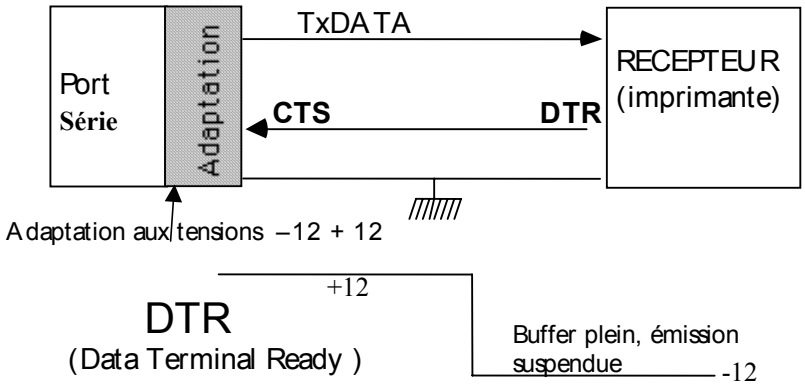
10.1.1. format du signal série .

Les données sont transmises suivant la norme RS232C, V24, sous forme de caractères de"7" ou 8 bits (7 bits pour du code ASCII, 8 bits pour du binaire), exemple :



La logique en ligne est négative (-12 volt pour le niveau logique 1).
La ligne au repos est à - 12 volt .
La rapidité de modulation peut varier de 300bauds à 192000bauds ou plus, ce qui correspond ici à un débit binaire en bits/s identique (un seul bit est transmis par temps élémentaire).
Le récepteur re-synchronise son horloge sur le front montant de chaque "start bit", une différence de période entre les horloges d'émission et de réception de 5% à 10% est admissible.

10.1.2. Protocole DTR .



CTS = signal « Clear to send » à gérer
La liaison comprend, une ligne de donnée série, un signal de retour de handshake, **DTR** ou DATA TERMINAL READY (Terminal pret). On notera la présence des adaptateurs permettant de passer des signaux TTL ou HCMOS, aux signaux de ligne RS232.
DTR à 12 volt autorise l'émission (branché sur CTS) .
Si le buffer de l'imprimante est presque plein, DTR passe à -12 volt .
Il s'agit d' un *demi handshake*.

10.1.3. Protocole XON-XOFF

Le handshake de retour s'effectue par des caractères de controle transitant sur la ligne de données RxDATA. Ce protocole permet, sans liaison supplémentaire, un échange d'informations dans les deux sens .

Le récepteur peut suspendre l'émission en envoyant le caractère **XOFF** (caractère ASCII $13). Au moyen du caractère **XON** (caractère ASCII $11), il lui demande de reprendre.

L'émetteur doit après chaque octet envoyé vérifier si il ne reçoit pas de caractère XOFF. Si oui, il doit attendre un XON pour continuer à émettre. Une durée limite dite « Time out » peut être gérée pour éviter un blocage !.

Ce protocole est désormais le plus courant.

10.2. Le port SCI de l'HC12: Sérial Communication Interface (RS232)

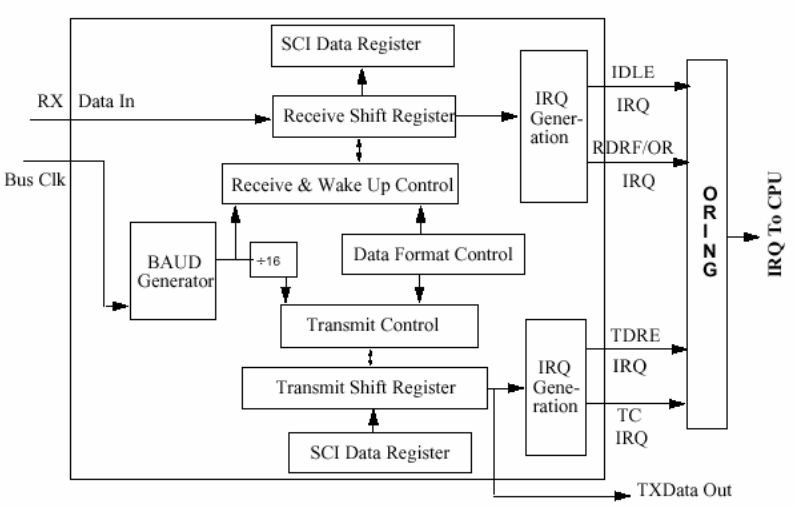
Deux ports SCI0 et SCI1.

Full Duplex
(il peut émettre et recevoir en même temps)

Systèmes de réveil du récepteur.

Détection d'erreurs de trame

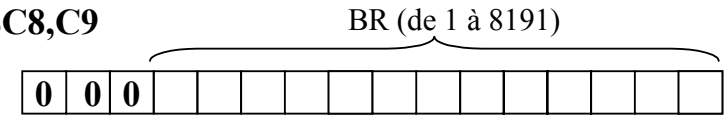
Block Diagram :



Nous n'étudions ici que le les bits assurant le fonctionnent de base, et le port SCI0. Le port SCI1 est identique à part les adresses de ses registres.

Remarque : Lors de la première initialisation des différents bits, on peut directement écrire 0 dans les bits non utilisés (soit c'est la valeur par défaut, soit écrire 0 est sans effet).

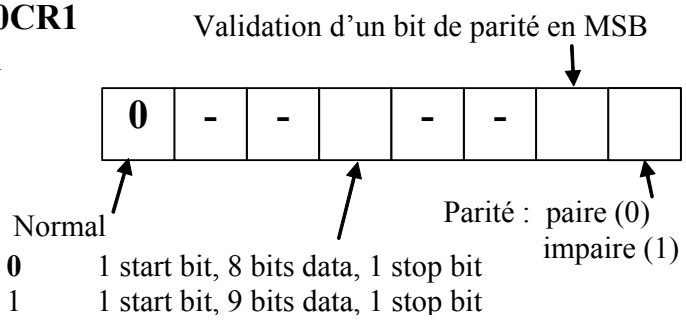
SCI0BD
$C8,C9



SCI Baud Rate byte H

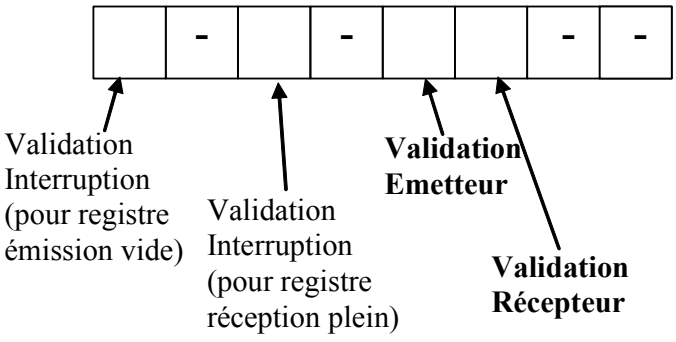
$$Baud\_rate = \frac{F\_BUS}{16.BR}$$

SCI0CR1
$CA



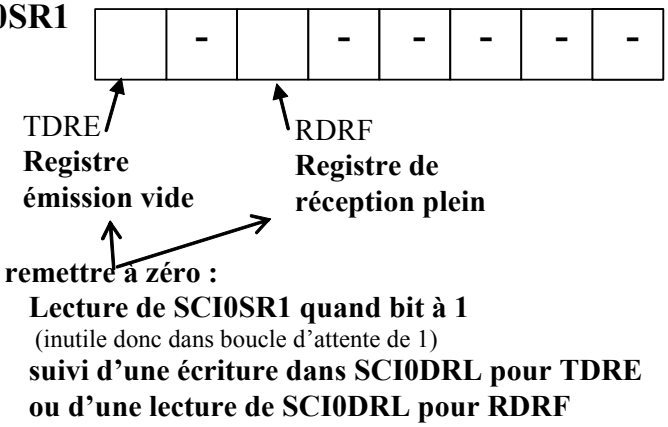
SCI Control register 1

SCI0CR2
$CB



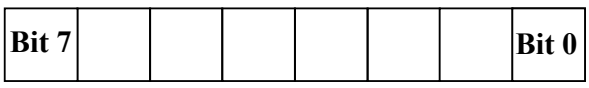
SCI Control register 2

SCI0SR1
$CC



SCI Status register 1

SCI0DRL
$CF



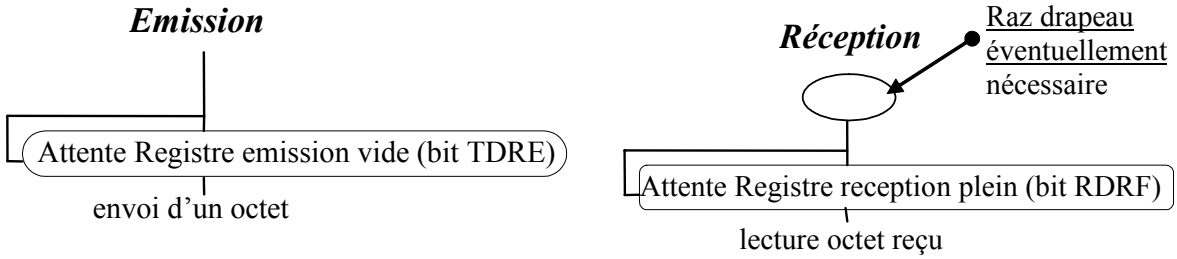
SCI Data Register Low
La donnée classique 8 bits

→ ***Remarque,*** pour **un Quartz de 4MHz**, avec **BR = 13**,
Avec F_BUS = 2MHz, on obtient **9615,38 Bauds**, correct pour une liaison à **9600 Bauds**.
On peut en effet calculer aisément la dérive : ici df = 9615,38 – 9600 = 15,38Hz, donc comme $\frac{dT}{T} = -\frac{df}{f}$ il vient $\frac{\Delta T}{T} = 1{,}6.10^{-3}$ donc $\Delta T = 1{,}6.10^{-3}$ T. Au bout de 10 bits (le code ASCII du caractère, un start bit et un stop bit), la dérive est donc d'environ $\Delta T = 1{,}6.10^{-2}$ T. Or on sait que sur une transmission asynchrone l'interface récepteur recale son horloge à chaque caractère reçu donc tous les 10 bits, <u>la dérive ne se cumule donc pas</u> lors de la

transmission d'une série de caractères. L'étude sur un caractère fournit ici une dérive de 1,6% de la durée bit T, donc tout à fait négligeable.

Avec F_BUS = 24MHz, on passerait à **115384,6** correct pour le débit classique de **115200 Bauds.** (On pourrait effectuer un calcul de dérive identique).

10.3. Organigrammes



➢ ***Très Important :***

En réception, toujours faire une **première remise à zéro du drapeau** « **Registre de Réception Plein »** avant toute première réception ou boucle de réceptions successives.

Il se peut en effet que ce bit soit déjà à 1 (réception précédente mal terminée, ou octets entrants non intéressant) pour que le premier octet d'une nouvelle réception soit alors faux et tous les autres décalés !

➢ ***Time Out ?***

Dans certain cas une attente sans fin peut être programmée, mais on peut aussi parfois sortie automatiquement de la boucle d'attente au bout d'une durée fixée à l'avance dans le cas ou aucune réponse par exemple n'arrive. On programme ce que l'on nomme un Time Out.

Idée N°1 : au moyen d'un **simple compteur auto-incrémenté dans la boucle d'attente** (variable **unsigned long Time_out**).

La valeur choisie ici (1000000) est à déterminer à l'essai et un peu au feeling… et dépend évidemment de F_BUS du processeur. Mais cela fonctionne car on ne cherche pas une durée précise ! La boucle devient alors :

Attente Registre Réception plein **OU** Time_out < 1000000

Ce qui peut s'écrire en C :

```
While(  ( (SCIOSR1 & 0x20) = = 0 )  && Time_out++ <1000000 ) ;
```

Si on veut sortir d'une boucle for (…..) { } d'acquisition de plusieurs caractères par exemple, on peut de nouveau tester la variable Time_out en même temps que la fin de chaîne, ou bien utiliser un if… et l'instruction C : **break** qui force la sortie d'une boucle.

Idée N°2 : On programme un **Timer sans interruption** (voir le chapitre sur la génération d'intervalle de temps, Timer) pour une durée de base. On le démarre à chaque attente de caractère et on teste le drapeau fin de comptage. Un compteur supplémentaire peut être ajouté pour des Time out plus longs. On maîtrise ainsi bien mieux la durée générée et elle peut être rendue indépendante de F_BUS du processeur.

11. PORT SERIE SYNCHRONE

Nommé **SPI** : **S**érial **P**eriphéral **I**nterface chez Motorola

11.1. utilisation

Interface série synchrone permettant le câblage avec seulement 4 ou 5 fils, de plusieurs périphériques (CAN, CNA, Afficheurs, simples registres à décalages pour conversion en parallèle, circuits plus complets IO de conversion SPI_Parallèles….) compatibles sur le même port, et même de plusieurs microcontrôleurs entre eux. Contrairement à un périphérique série Asynchrone type UART, l'horloge est transmise sur un fil séparé.

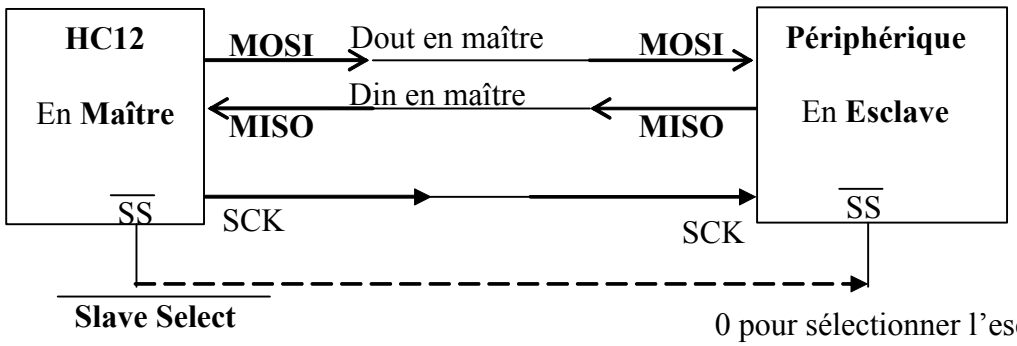
♦ ***Caractéristiques :***

- Simple liaison à 4 ou 5 fils (masse, data IN et OUT, et Sérial Clock) + Slave Select Eventuel.
- Débits assez importants (plusieurs Mégabits/s selon les processeurs)
- Fonctionnement avec un circuit maître, l'autre esclave
- Liaison Full Duplex possible.

- Présent sur de nombreux processeurs: Microcontrôleurs Motorola HC11,HC12 (Bus SPI et IIC), Philips(bus IIC), Processeurs de Signaux Texas ……

11.2. Principe du SPI

11.2.1. Liaison Maître ←→ Esclave



Din et Dout peuvent circuler en même temps (liaison full duplex possible).

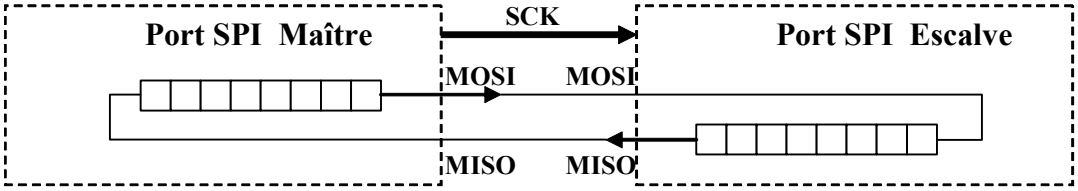
Les circuits se programment soit Maître soit Esclave, le sens des lignes MISO et MOSI en dépendent..

MISO signifie **M**aster **I**nput **S**lave **I**nput
MOSI signifie **M**aster **O**utput, **S**lave Input

La ligne $\overline{Slave\_Select}$ peut être activée :

A chaque transfert,
Pout un transfert de 2 ou de plusieurs octets,
En permanence (laison alors inutile), tout dépend du périphérique.

11.2.2. « L'echange » entre deux SPI



Seul le Maître peut générer l'horloge **SCK** par envoi d'octet vers l'esclave. Il a l'initiative de tous les échanges.

Ecriture Maître→Esclave :

Le maître écrit un octet dans son registre de donnée
La donné est sérialisée, SCK générée.
L'esclave la récupère bit par bit en synchronisme avec SCK.
En même temps la donnée de l'esclave sort vers le maître, mais inutile.

Lecture Maître← Esclave :

Le maître écrit une donnée quelconque dans son registre de donnée (0) par exemple.
SCK est générée, et envoyée à l'esclave pour que celui-ci émette sa donnée.
Le maître récupère cette donnée bit par bit en synchronisme avec SCK.

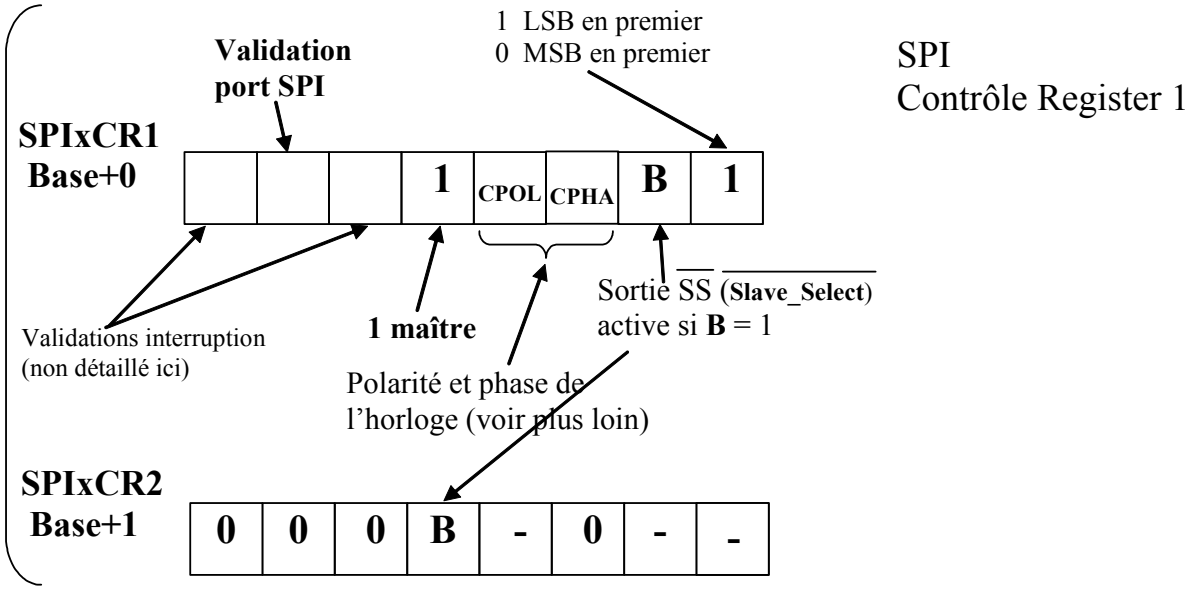
On voit que l'on pourra en fait écrire **une seule fonction d'échange**, de prototype par exemple : **char io_spi(char c) *;***

Utilisation :

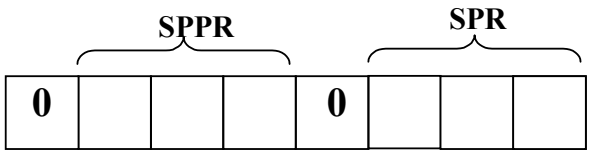
Ecriture Maître→ Esclave : io_spi(octet) ;
Lecture Mâitre← Esclave : c = io_spi(0) ;

11.3. Description brève des ports SPI de l'HC12

L'HC12 possède 3 ports de ce type : SPI0, SPI1, SPI2, d'adresse de **base $D8**, **$F0**, **$F8** respectivement.



SPIxBR
Base+2

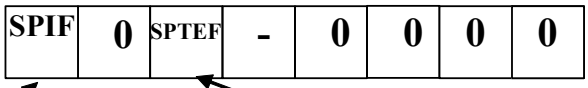


SPI Baud Rate:

$$Baud = \frac{F\_bus}{(SPPR+1)2^{(SPR+1)}}$$

Attention : **max F_bus/4**

SPIxSR
Base+3



SPI Status Register:

SPIF interrupt flag
Et **Fin de sérialisation**
Raz :
Lecture SPIxSR avec bit à 1
ET Lecture SPIxDR

SPI Transmit Empty
Interrupt flag
Raz :
Lecture de SPIxSR
ET écriture dans SPIxDR

Attention: Remise à zéro de drapeau un peu particulière pour SPTEF:
Une lecture de SPIxSR est nécessaire AVANT d'envoyer un octet !

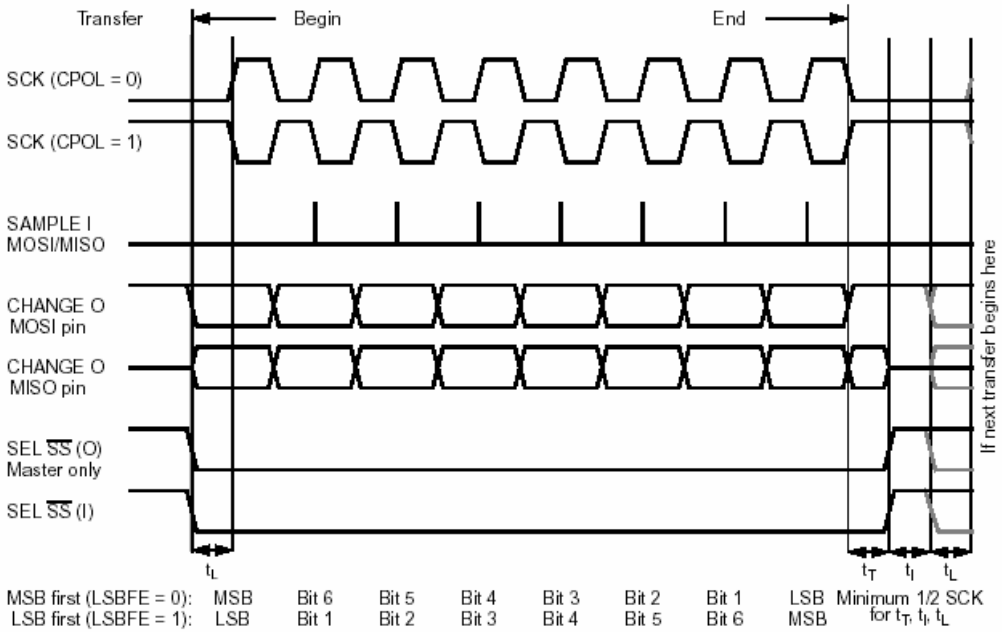
SPIxDR
Base+5

SPI Data Register

Remarque : Lors de la première initialisation des différents bits, on peut directement écrire 0 dans les bits non utilisés (soit c'est la valeur par défaut, soit écrire 0 est sans effet).

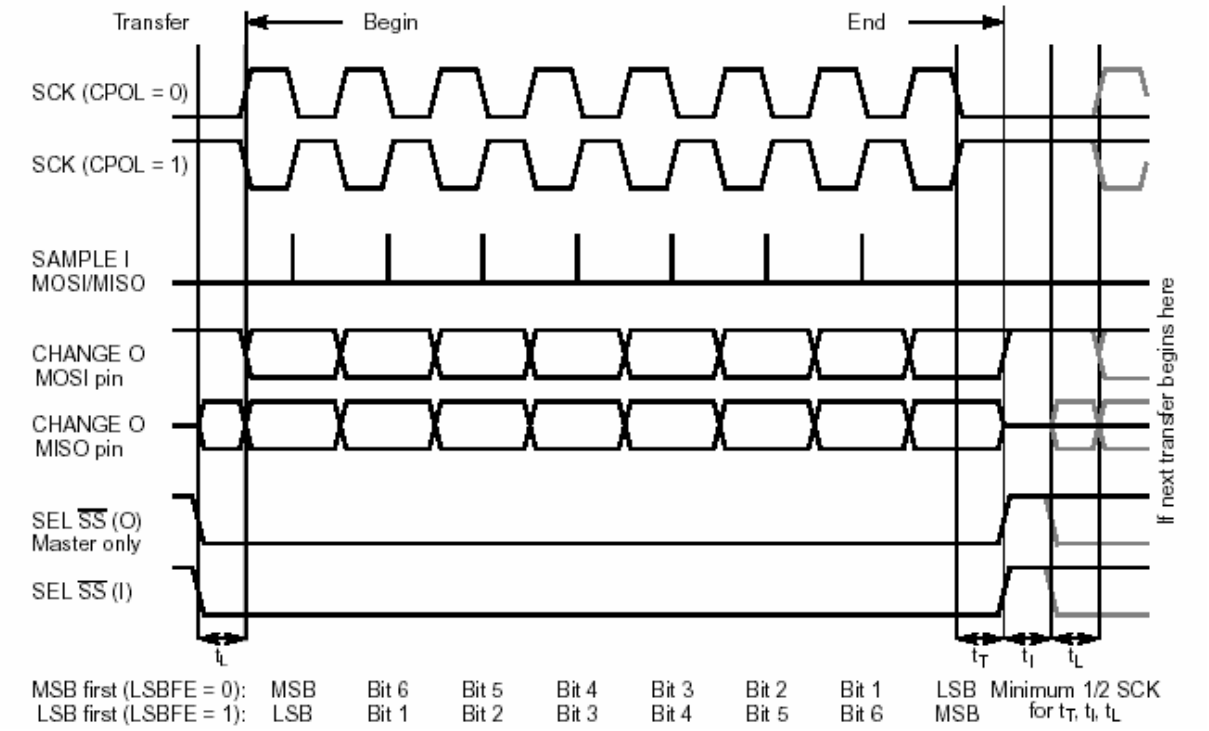
On peut programmer 4 possibilités de signaux échangés, pour s'adapter à tel ou tel périphérique :

→ ***Format pour CPHA = 0 :***



→ Format pour CPHA = 1 :

Pour certains périphériques, un premier Ck doit être présent avant la première donnée.



11.4. Exemples d'application

11.4.1. Sortie parallèle câblée sur un bus SPI

1) Montage

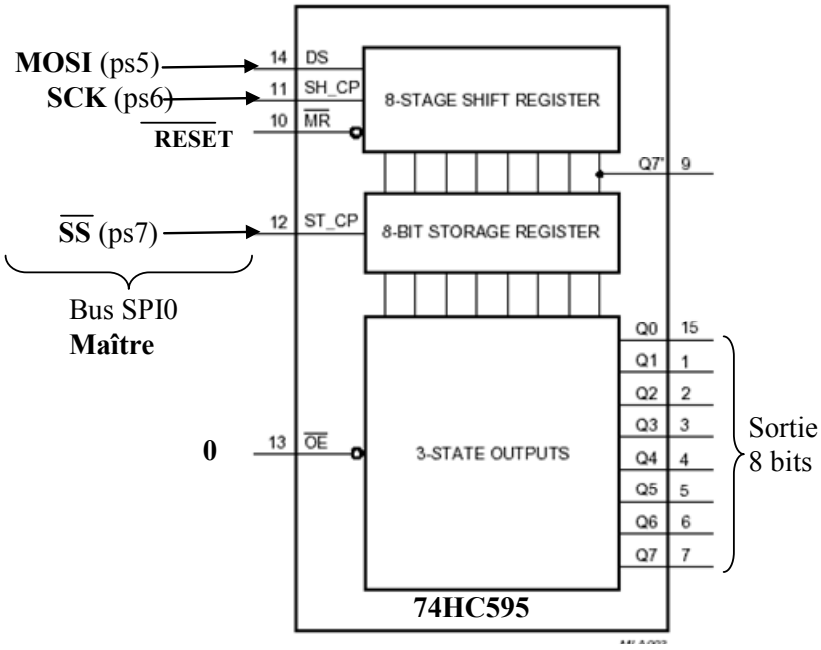
Sont nécessaires : Un registre à décalage avec entrée série, ainsi qu'un registre « latch » de 8 bascules D assurant le changement de la donnée en sortie sans états intermédiaires.

Exemple de câblage d'un circuit 74HC595 :

DS Entrée série
SH_CP entrée clock décalage
Font ↑ actif

ST_CP entrée clock latch
Front ↑ actif

OE* mise à basse impédance des buffer de sortie.



SCK câblé sur l'entrée SH_CP active sur **front montant,** effectue le **décalage** de la donnée arrivant bit par bit par la ligne MOSI.

Après 8 fronts d'horloge, la **donnée est mémorisée** vers les Qi sur la remontée du signal **SS** ($\overline{Slave\_Select}$), câblé sur l'entrée **ST_CP.**

On doit donc envoyer un signal $\overline{Slave\_Select}$ actif à chaque transfert, le mode automatique est très bien.

2) ***Exemple de programmation très simple, SPI0 en mode maître:***

// à 500kHz pour F_BUS 2MHz
// à 1MHz pour F_BUS 4MHz
// à 2MHz pour F_BUS >=8MHZ

On suppose F_BUS dèjà défini par un #define.

➔ ***Fonction initialisant le Port SPI0, compatible 74AC595***

```
// ------------ Initialisation de SPI0 différente, si on veut utiliser l'interface Output série parallèle de la carte
//            NE PAS UTILISER SI ON SE SERT DU CLAVIER   !!!!!!!!!
//  à 500kHz F_BUS 2MHz
//  à 1MHz   F_BUS 4MHz
//  à 2MHz   F_BUS >=8MHZ
void init_spi0(void)  //  En maître, baud = voir ci-dessus
{
char c=0;
if(F_BUS <= 8)c=1;   // div par 4
if(F_BUS == 12)c=0x20;       // div par 6  code   0 010 0 000
if(F_BUS == 16)c=2; // div 8
if(F_BUS == 24)c=0x21;       // div par 12 code   0 010 0 001
SPI0CR1 = 0x52; // 0101 0010 spi1 validé, maitre, validation Sélection esclave
SPI0CR2 = 0x10; // 0001 0000  normal
SPI0BR = c;     // Baud rate = Fbus/1  = 4/4 = 1MHz  (pas plus car Baud <=Fbus/4)
}
```

➔ ***Fonction de sortie sur ce port***

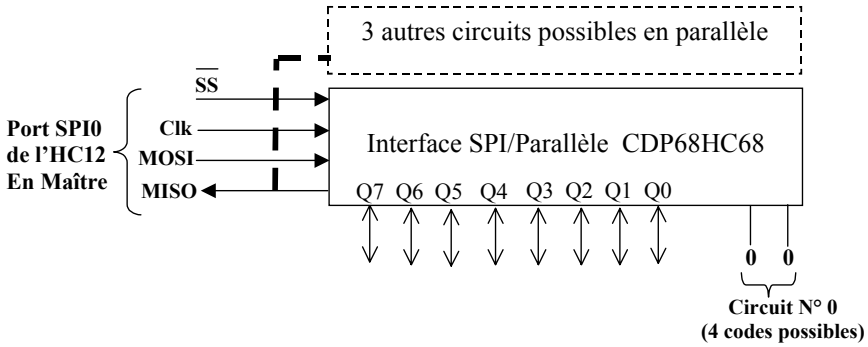
Ce port étant par son cablage même unidirectionel, on peut écrire seulement une simple fonction de sortie :

```
void envoi_spi0(char octet)  // en Maitre
{
static char c;  // sinon l'optimisateur peut le supprimer !
c = SPI1SR;   // obligatoire juste avant envoi, sinon envoi ignoré
SPI1DR = octet;              // envoi data
while((SPI1SR & 0x80) ==0);  // attente sérialisation
c = SPI1DR ;  // lecture inutile ici, mais obligatoire pour raz drapeau de fin de sérialisation.
 }
```

11.4.2. Circuit bidirectionnel interface SPI_parallèlle : CDP68HC68P1

Sur la maquette que nous utilisons, le clavier est cablé de cette facon sur le port SPI0. Nous allons donc étudier un peu plus en détail ce composant.

On pourrait ainsi obtenir **4 ports parallèles** par un simple câblage **sur un port SPI**.



11.4.2.1. Description brève du CDP68HC68P1

Ce circuit câblé sur un bus SPI le transforme un peu en un port parallèle classique, avec son registre de donnée (Data Register) et son registre de direction (Data Direction Register, ou DDR, pour programmer le sens de chaque ligne). On peut aussi manipuler individuellement chaque bit, par des commandes spécifiques.

Des Drivers sont évidemment nécessaires, et ce port sera bien moins rapide qu'un vrai port parallèle !

Tous les **accès** à ce circuit se font **en deux temps** : un **octet de commande** (avec indication du circuit concerné, 4 possibles sur le même bus, du registre, et du sens de transfert), suivi d'un **octet de data**.

La sélection du circuit par $\overline{Slave\_Select}$ doit être maintenue durant les 2 deux accès. Le mode automatique n'est donc pas possible. Celle ligne qui est en fait PS7 du PORTS, sera donc gérée comme une ligne de port parallèle.

Pour plus de détail, voir la documentation PDF sur le NET….

11.4.2.2. Programmation, drivers de ce port parallèle

1) Fonction initialisant le port SPI0

```
#define F_BUS        2                  // dans ini_carte.h
void init_spi0_hc68_clavier(void)
{
// Initialisation de SPI0 pour compatible avec circuit IO série parallèle: CDP68HC68P1
 SPI0CR1=0x54; // 0 1 0 1  0 1 0 0   SPI Enable, Maître, Pol 0, phase 1 , Slave Select non géré automatiquement, MSB en tête
 SPI0CR2=0x00; // Slave Select non géré automatiquement, car doit être actif durant deux octets
 if(F_BUS == 2) SPI0BR= 0x01;  //  Spi Baud Rate = 2/4 = 500 KHz
 if(F_BUS == 4) SPI0BR= 0x01;  //  Spi Baud Rate = 4/4 = 1 MHz
 if(F_BUS == 8) SPI0BR= 0x01;  //  Spi Baud Rate = 8/4 = 2 MHz
 if(F_BUS ==16) SPI0BR= 0x02;  //  Spi Baud Rate = 16/8 = 2 MHz
 if(F_BUS ==20) SPI0BR= 0x40;  //  Spi Baud Rate = 20/10 = 2 MHz   - 1 0 0 - 0 0 0
 if(F_BUS ==24) SPI0BR= 0x21;  //  Spi Baud Rate = 24/12 = 2 MHz   - 0 1 0 - 0 0 1
 DDRS=0x80;PTS=0x80;  // Slave Select manipulé par le bit PTS7  (pour transmission par bloc de 16 bits)
```

2) Fonction de lecture écriture du port SPI0

```
char io_spi0(char octet)
{
static char raz;                    // sinon l'optimisateur peut le supprimer !
raz = SPI0SR;                       // obligatoire juste avant envoi, sinon envoi ignoré: raz drapeau lecture
SPI0DR = octet;             // envoi data
while((SPI0SR & 0x80) ==0);   // attente sérialisation: drapeau
return(SPI0DR);                             // et raz drapeau lecture
      // obligatoire même si on ne fait qu'une écriture, car écriture et lecture sont indissociables
 }
```

3) *Fonctions initialisant le DDR du CD68HC68P1 (Câblé N°0)*

```
Void ecrire_ddr(char ddr)
{
 PTS=PTS&0x7f;  // SSn à 0: Sélection du CDP68HC68P1, cablé numéro 0
 io_spi0(0x30);             // Code commande pour l'accès à DDR en écriture
 io_spi0(ddr);
 PTS=PTS|0x80;  // SSn à 1: Déselection de CDP68HC68P1
}
```

4) Fonction d'écriture et de lecture de ce « nouveau port parallèle » (câblé N°0)

Fonctions d'écriture ou de lecture du port parallèle

```
 void ecrire_port(char c)
 {
  PTS=PTS&0x7f;
  io_spi0(0x10);                    /*code commande pour écrire une donnée dans Data Register  */
  io_spi0(c);
  PTS=PTS|0x80;
 }

 char lire_port(void)
 {
 char k;
 PTS=PTS&0x7f;
 io_spi0(0x00);                     /* code commande pour lire une donnée dans Data Register */
 k=io_spi0(0x00);
 PTS=PTS|0x80;
 return(k);
 }
```

1. INTRODUCTION ET MATERIEL 1

1.1. Microprocesseurs, Microcontrôleurs et les autres 1
1.2. La « carte » à Microcontrôleur 1
1.3. Types de structure interne 3
1.4. Le mode Single Chip d'un microcontrôleur 4
1.5. Le mode étendu d'un microcontrôleur 4
1.6. Problèmes de puissance 6

2. INTRODUCTION A LA PROGRAMMATION 7

2.1. Choix du ou des langages de programmation 7
2.2. Programmation structurée 8
2.3. Chaîne de compilation 8

3. PROGRAMMATION EN ASSEMBLEUR 15

3.1. Résumé du langage assembleur 16
3.2. L'Assembleur type HC12, résumé 25
3.3. Programme de démonstration en assembleur HC12 : 4_leds 37
3.4. Astuces de gestion de variables indicées en assembleur 43

4. PROGRAMMATION EN C, INTRODUCTION 45

4.1. Avantage : 45
4.2. Programme de démonstration en C : 4_leds_c 45

5. C POUR MICROCONTROLEURS 55

5.1. Le langage C « minimum » 55
5.2. Fonctions en C 63
5.3. Retour sur le travail d'un compilateur 65
5.4. Optimisations du C pour accélérer la vitesse 68

6. LES FONCTIONS, DE L'ASSEMBLEUR AU C EN TYPE HC12 71

6.1. Passages de paramètres en assembleur 72
6.2. Macros (en assembleur) 75
6.3. Fonction C en assembleur 76
6.4. Discussion sur le pointeur de trame 81

7. SYNCHONISATION SUR EVENEMENTS EXTERIEURS, INTERRUPTIONS 82

7.1. Sondage ou Interruption ? 82
7.2. Sondage 83
7.3. Interruption 84

8. LES VECTEURS D'INTERRUPTION MISE AU POINT ET APPLICATION FINALE 88

8.1. Développement sur Microcontrôleurs nécessitant un « Moniteur » : exemple l'HC11 88

8.2. Développement sur Microcontrôleurs sans « Moniteur » : exemple l'HC12 94
8.3. Exemple sur microprocesseur type HC12, en C, avec sonde BDM 95

9. PORTS PARALLELES 97

9.1. Utilisation en simple port IO 97
9.2. Utilisation possibles de certaines lignes en entrées d'interruption 97
9.3. Le Port Integration Module de l'HC12, résumé 97
9.4. Ports parallèles d'entrée-sortie (I/0) en usage général que l'on utilisera sur notre maquette 98

10. PORT SERIE ASYNCHRONE TYPE UART 101

10.1. Petit rappel sur la liaison série asynchrone 101
10.2. Le port SCI de l'HC12: Sérial Communication Interface (RS232) 102
10.3. Organigrammes 104

11. PORT SERIE SYNCHRONE 105

11.1. utilisation 105
11.2. Principe du SPI 105
11.3. Description brève des ports SPI de l'HC12 106
11.4. Exemples d'application 108