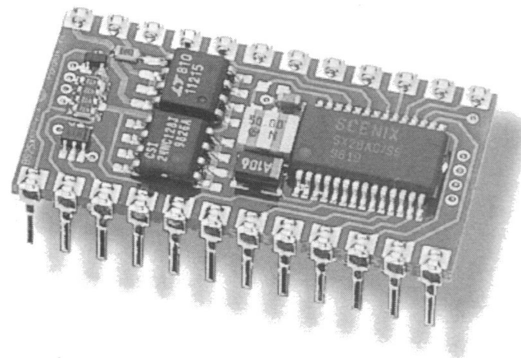
École polytechnique universitaire de Marseille

Les microcontrôleurs



Dpt ME - Otion SIIC 2ème année
2005-2006 – ver 4.0

Jérôme VICENTE

Sommaire

AVANT PROPOS **4**

I. MICROPROCESSEUR **7**

1.1. Système informatique minimal 7

1.2. Exécution de programme 11

1.3. Les Interruptions 18

II. MICROCONTROLEURS **21**

2.1. Architecture d'un Microcontrôleur 21

2.2. Etude du microcontrôleur 8 bits ST7 24

a) Les registres internes 24
b) L'organisation mémoire (memory map) 26
c) Le jeux d'instruction 26

2.3. Les Périphériques du ST7 27

2.3.1. Programmation et configuration des périphériques 27
2.3.2 Les Timers 29
2.3.3. Le Convertisseur analogique/numérique (CAN) 33
2.3.4. Les ports d'entrées/sorties parallèles 34

2.4. Les interruptions 39

III. UTILISATION DU LANGAGE C **40**

3.1. Organisation mémoire du ST7 et options de compilation 41

3.2. Allocation des variables 42

3.2.1. La zone DEFAULT_RAM 42
3.2.2. Les variables en page zéro 43
3.2.3. Les constantes 43
3.2.4. Stockage en mémoire EEPROM 44
3.2.5. Allocation de bouts de code 44
3.2.6. Accés à la mémoire via des pointeurs 44

3.3. Les registres des Périphériques 45

3.3.1. Déclaration des registres 45
3.3.2. Lecture, écriture et test d'un bit dans un registre 46
3.3.3. Configuration des registres lors d'initialisation de périphériques 47
3.3.4. Utilisation de macros pour les opérations sur les bit 47

3.4. Programmation des Interruptions 49

3.5. Langage C optimisé pour microcontrôleurs 50

IV. LA MISE EN ŒUVRE 54

4.1. Mise en œuvre matérielle 54

4.2. Mise en œuvre logicielle 56

4.3. La chaîne de développement ST7 59

4.4. Exemple de projet........ 61

4.4.1. Programme principal “main.c” 61

4.4.2. fichier de link “enviro.prm” 63

4.4.3. fichier d’environnement “defaut.env” 64

4.4.4. fichier make “enviro.mak” 64

Avant propos

Microcontrôleur :

Circuit programmable capable d'exécuter un programme et qui possède des circuits d'interface intégrés avec le monde extérieur.

Les microcontrôleurs sont apparus quand :

⇒ Quand on a sut les fabriquer, cad quand les technologies d'intégrations ont suffisamment progressées

⇒ Quand dans les applications domestiques ou industrielles ont avait besoin de systèmes « intelligents » ou tout au moins programmables.

Exemple
La machine à laver qui doit commander différents organes avec des cycles bien définis mais variables en fonction du programme choisi.

Quand utiliser un microcontrôleur ?

Toutes les solutions à base de composants programmables ont pour but de réduire le nombre de composants sur le circuit électronique et donc fiabiliser le circuit.

Le microcontrôleur est en concurrence avec d'autres technologies
Suivants les applications : 3 types de technologies

Logique câblée
☺ très rapide, fonctions réalisées par une voie matérielle
☹ non programmable, peu économique quand l'application est complexe
peu de souplesse : durée d'étude prohibitif et circuit difficilement modifiable

Réseaux de logique programmables (PAL, LCA,..)
☺ rapide, adapté au traitement de signaux complexes
☹ prix élevé et langage de programmation non standard

Les µprocesseurs
☺ grande souplesse : fonctions sont réalisées par voie logicielle
puissance de calcul, langage évolué
☹ nombre important de composant à réunir, solution onéreuse

A retenir
si la fonction à réaliser est simple ⇒ une logique câblée
si le nombre d'unités à réaliser est très important ⇒ circuits intégrés dédié en logique câblée pour les fonctions simples

Une réalisation logicielle est toujours plus lente qu'une réalisation en logique câblée : le microprocesseur exécute une instruction à la fois

Les µcontrôleurs = avantage des µprocesseurs mais limités aux applications ne nécessitant pas trop de puissance de calcul, nombre de composant très réduit, mais souvent surdimensionnement devant les besoins de l'application)

Les avantages des microcontrôleurs

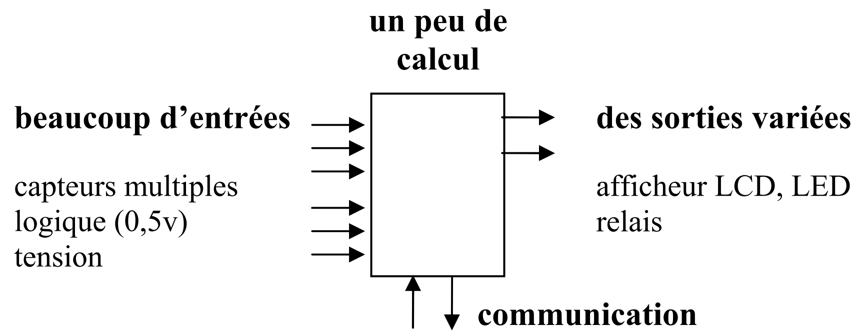
- Diminution de l'encombrement du matériel et du circuit imprimé
- Simplification du tracé du circuit imprimé p(lus besoin de tracer de bus !)
- Augmentation de la fiabilité du système
  nombre de composants ↘
  connexions composants/supports et composant circuit imprimé ↘
- Intégration en technologie MOS, CMOS, ou HCMOS
  diminution de la consommation
- Le microcontrôleur contribue à réduire les coûts à plusieurs niveaux:
  moins cher que les composants qu'il remplace
  Diminution des coûts de main d'œuvre (conception et montage)
- Environnement de programmation et de simulation évolués

Les défauts des microcontrôleurs

- le microcontrôleur est souvent surdimensionné devant les besoins de l'application
- Investissement dans les outils de développement
- Écrire les programmes, les tester et tester leur mise en place sur le matériel qui entoure le microcontrôleur
- Incompatibilité possible des outils de développement pour des microcontrôleurs de même marque.
- Les microcontrôleurs les plus intégrés et les moins coûteux sont ceux disposant de ROM programmables par masque
  Fabrication uniquement en grande série >1000

  Défaut relatif car il existe maintenant systématique des version OTPROM un peu plus chère.

En conclusion :



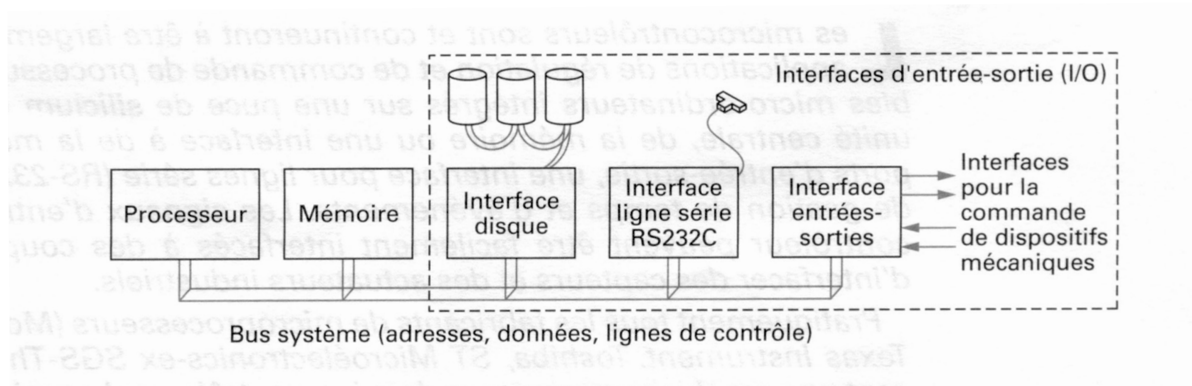
le microcontrôleur présente l'avantage des µprocesseurs mais limités aux applications ne nécessitant pas trop de puissance de calcul (architecture courante 8bits)

Il existe plusieurs architecture de microcontrôleurs de 4 à 128 bits pour les applications demandant une certaine puissance de calcul (injecteurs automobile)

I. Microprocesseur

1.1. Système informatique minimal

D'un point de vue matériel, un système info minimal est constitué d'un processeur, d'une mémoire, et d'entrées sorties (figure 1)



a) Le processeur (CPU)

Il a pour mission de rechercher les instructions qui sont en mémoire, de les décoder et de les exécuter. il composée de plusieurs éléments internes :

CPU = UAL + UC + registres CPU

1 **unité arithmétique et logique (UAL)** : chargée des calculs +,-,*,/, AND, OR, NOT

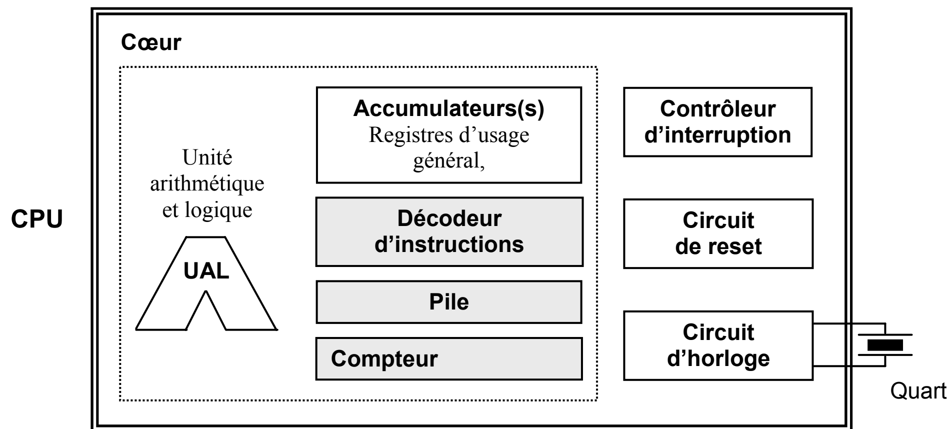
1 **unité de commande** chargée de traduire puis d'exécuter les commandes

Son rôle est d'aller chercher une information en mémoire centrale, d'analyser cette instruction **(décodage)**, d'exécuter cette instruction, de localiser l'instruction suivante.

- Un décodeur d'instruction
- Un séquenceur et des circuits de commande

Un ensemble de circuits électronique commandés par l'unité de contrôle et permettant :

- D'échanger des informations avec la mémoire centrale et avec le monde extérieur (avec les périphériques)
- D'effectuer des opérations sur les données (**Unité Arithmétique et Logique : UAL ou ALU en anglais)**
- De mémoriser l'adresse de la prochaine instruction dans un registre particulier PC (program counter))
- De mémoriser le résultat d'opérations dans des mémoires spéciales : les Registres de travail



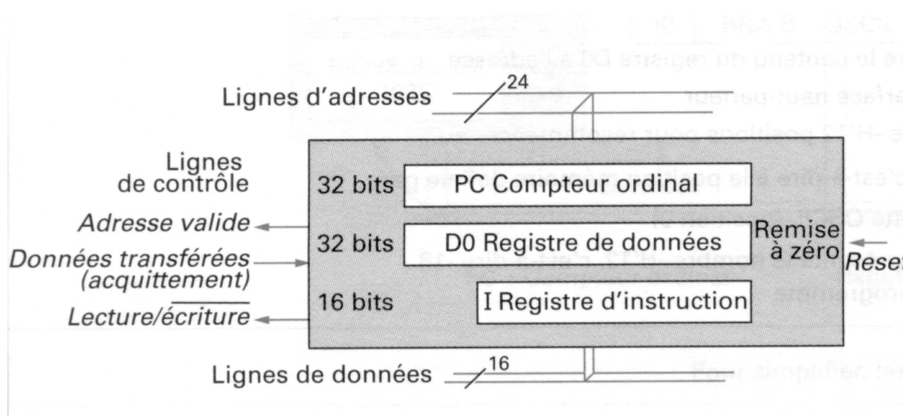
les **registres CPU :** cellules mémoire interne au processeur (rapide)

1 **compteur ordinal (PC)** pointant à l'adresse mémoire où se trouve la prochaine instruction à rechercher et exécuter. Après chaque recherche d'instruction, le compteur ordinal est incrémenté afin de pointer à la prochaine instruction (en fait après le chargement de l'octet ou du mot mémoire faisant partie d'une instruction, de manière à pointer sur un code d'instruction.

1 **registre d'instruction (I)** qui contient le code de l'instruction (code opératoire) recherchée en mémoire.

Les **registres de données (X, Y, D0, Dx,..)** permettent de stocker les opérandes nécessaires aux instructions de calcul ainsi que les résultats lors d'opérations logiques et arithmétiques.

Les **registres d'adresses (A, A0, Ax,..)** permettent de stocker les adresses d'opérandes qui se trouvent en mémoire.



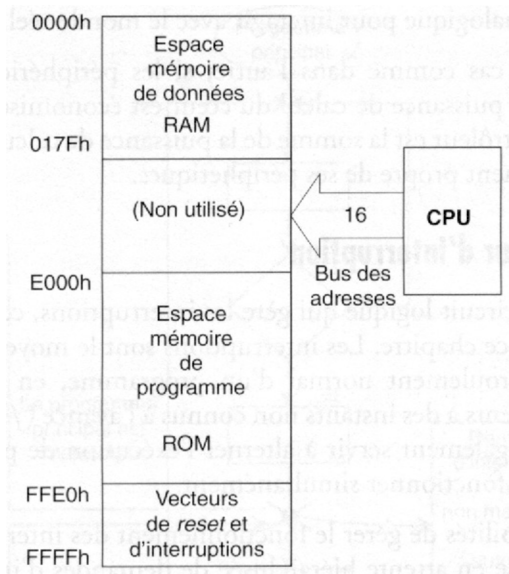
b) la mémoire

la mémoire est utilisée pour le stockage d'instructions, de données et de la pile (la pile est une portion mémoire réservée pour sauvegarder le contexte d'exécution d'une procédure cf. plus

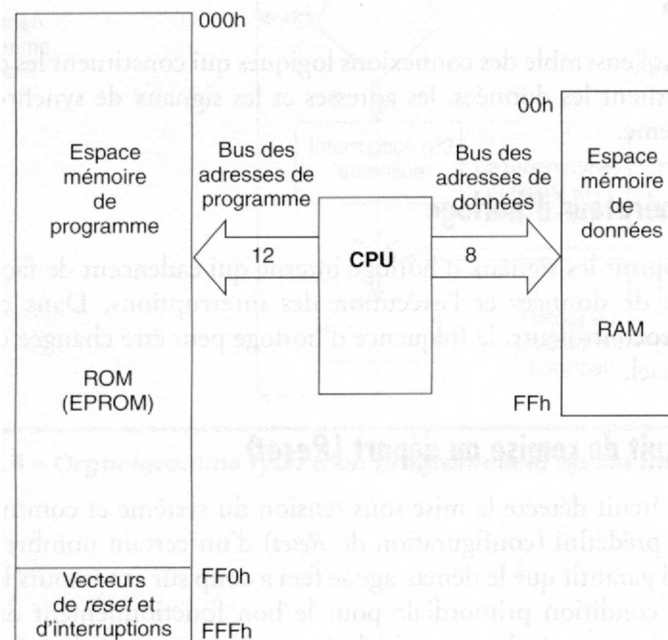
loin). La mémoire est constituée de cellule de mémorisation binaires, groupées en mots de 8, 16, 32 ou 64 bits. La mémoire est adressée par mots de 8 bits, c.a.d par **octets (bytes)**. Une position mémoire est spécifiée par l'adresse d'un octet.

On distingue 2 architectures différentes.

- La première dite « de **Von Neumann** » à un seul espace adressable. Le programme et les données sont simplement rangés à des adresses différentes. (c'est le cas du ST7)

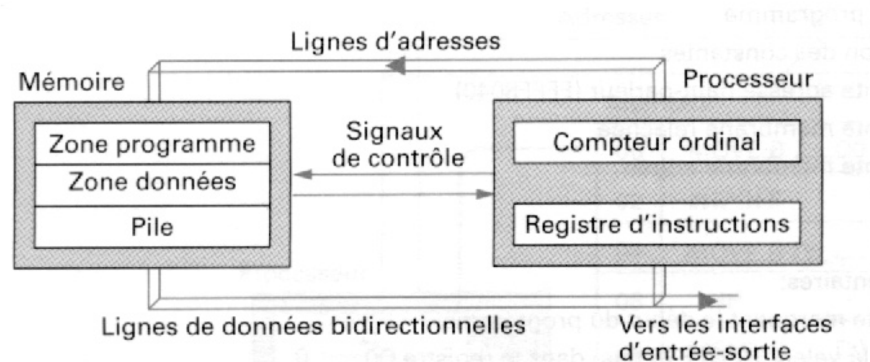


- La deuxième dite « de **Harvard** » offre deux espaces physiquement séparés pour le programme et les données. Il est donc impossible d'écrire dans l'espace de programme, ce qui le protège contre les destructions accidentelles et augmente l'espace adressable. (cas du ST6, ST9, etc..)



c) Le bus système

Il comporte les lignes permettant de relier entre eux le processeur, la mémoire, et les entrées sorties. Il comprend les **lignes d'adresses** provenant du microprocesseur, les **lignes de données** bidirectionnelles et les **lignes de contrôle**. Le cœur d'un système informatique est formé par l'interaction entre le processeur et sa mémoire.



d) Une horloge

(un circuit oscillateur délivrant des impulsions à une certaine fréquence) sera nécessaire pour exécuter les opérations séquentiellement). Plus la fréquence sera grande, plus l'unité centrale travaillera vite. (8MHZ veut dire 8 millions d'impulsions par secondes)

e) Les entrées-sorties

Elles permettent au processeur d'accéder au monde extérieur par l'intermédiaire de cycle de lecture et d'écriture sur des interfaces d'entrée-sortie. (Interface clavier, écran, disque, ligne série, etc…). Une interface d'entrée-sortie est généralement formée d'un registres se trouvant à une certaine position d'adressage du microprocesseur.

Les Circuits d'interfaces et les Périphériques

On distingue 3 types de périphériques pour un ordinateur classique :

- d'entrées (clavier, souris, etc…)
- de sorties (écran, imprimante, etc..)
- d'entrée/ sorties (ou mémoire de masse, lecteur de disquette, etc..)

Pour relier les périphériques à l'ensemble unité central + Mémoires, il faut passer par une **Interface (carte interface, circuit interface, coupleur de périphériques).**

L'interface se charge de l'adaptation des signaux électroniques et de gérer le dialogue entre le système et le périphérique (protocole d'échange).

1.2. Exécution de programme

a) Règles d'un système programmable

Le système doit respecter les 4 règles suivantes :

1) Les instructions et les données sont dans une mémoire unique, banalisée, accessible en lecture/écriture

2) Les contenus de la mémoire sont accessibles par leurs adresses

3) La commande de l'ensemble, l'exécution des opérations se fait de manière séquentielle (sauf indication expresse). L'exécution d'une opération doit être terminée avant le lancement de la suivante.

4) L'unité de traitement contient un jeu complet des opérations de l'algèbre de Boole (au minimum).

Le programme est une suite de nombres qui code les instructions du langage de programmation. Les données sont d'une part ce qui est traité sous le contrôle des instructions et d'autre part les résultats du calcul.

Exécution d'une commande (1 à plusieurs cycles horloge) :
Fetch = aller chercher le code de la prochaine instruction
Decode = analyse du code pour savoir quelle opération exécuter
Execute = exécuter l'instruction => UAL
Write back = écriture du résultat en mémoire (ou registre)

Exemple :

L'unité centrale **lit** l'instruction LD A,100 en mémoire centrale.
Elle **décode l'instruction** : il faut charger le registre A avec la valeur contenue dans la case mémoire n°100
L'unité centrale **lance l'exécution** : demande de lecture de la mémoire à l'adresse 100.
Elle **récupère la valeur lue** et enfin **la range** dans A.
Elle **met à jour le compteur PC**, et l'on continu : lecture de l'instruction suivante….

b) exemple du programme BruitHP

Périphérique de sortie : le haut parleur
Le programme consiste à envoyer de manière répétitive sur le registre de sortie haut-parleur un signal binaire alternatif (1,0,1,0…) afin de faire vibrer celui ci

Tableau 1 – Signification d'instructions de transfert et de saut

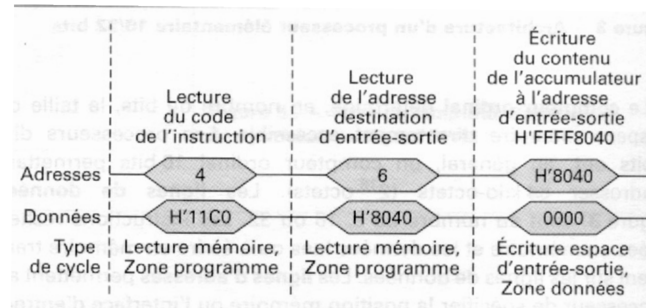
Instruction en assembleur	Signification
MOVE.B #QUANTITE, D0	Chargement de **QUANTITE** dans le registre **D0**, **QUANTITE** est une valeur immédiate 8 bits qui se trouve dans l'instruction
MOVE.B D0, PERIPH	Transfert du contenu du registre D0 (8 bits) vers l'adresse d'entrée-sortie **PERIPH**
BRA ETIQUETTE	Saut relatif à l'instruction se trouvant à l'adresse spécifiée par **ETIQUETTE** (adresse symbolique représentant une adresse absolue)

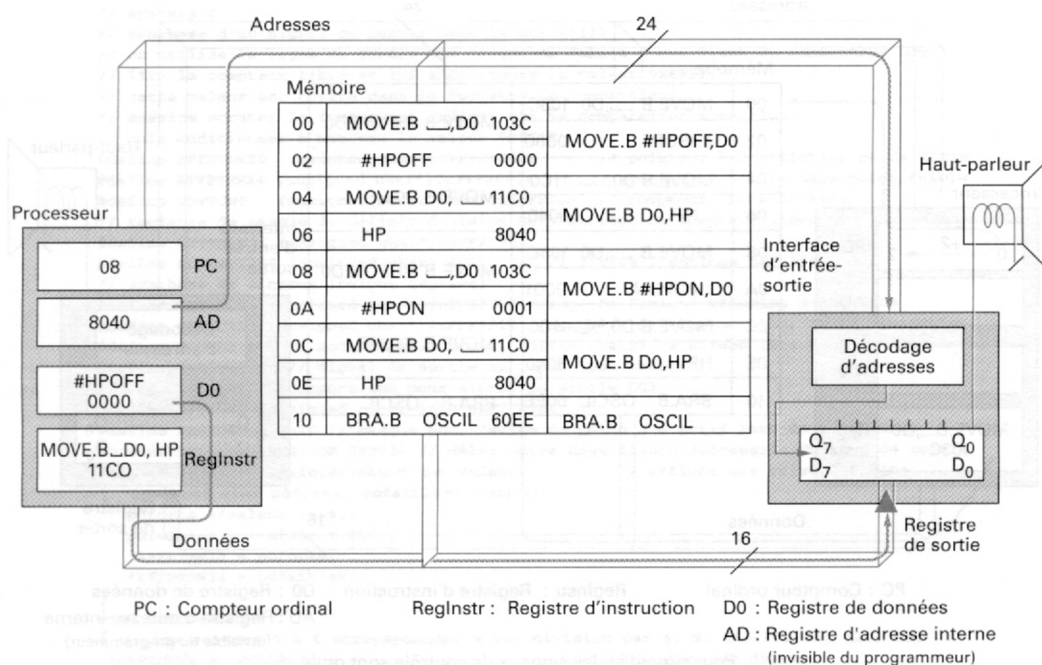
```
                TITLE BruitHP               ; titre du programme
                                            ; définition des constantes
                      HP      EQU 0x8040    ; constante adresse haut-parleur (FFFF8040)
                      HPOFF   EQU 0         ; constante membrane relachée
                      HPON    EQU 1         ; constante membrane attirée
;        code
;        hexa-        programme
;adresse: décimal:    assembleur:           ; commentaires:
                      OSCIL:                ; étiquette marquant le début du programme
 0000     103C        MOVE.B  #HPOFF, D0    ; charge la valeur HPOFF (8 bits) dans le registre D0
          0000
 0004     11C0        MOVE.B  D0, HP        ; transfère le contenu du registre D0 (8 bits) à l'adresse
          8040                              ; de l'interface haut-parleur
 0008     103C        MOVE.B  #HPON, D0     ; charge la valeur HPON dans le registre D0
          0001
 000C     11C0        MOVE.B  D0, HP        ; transfère le contenu du registre D0 à l'adresse
          8040                              ; de l'interface haut-parleur
 00010    60EE        BRA.B   OSCIL         ; saute de -H'12 positions pour recommencer au
                                            ; début, c'est-à-dire à la position mémoire définie par
                                            ; l'étiquette OSCIL (position 0)
                                            ; H'EE représente le nombre -H'12, c'est-à-dire -18
                      END                   ; fin du programme
```

Dans le cas de notre processeur simple, après activation de la ligne de remise à zéro (*reset*) le compteur ordinal (PC) est remis à zéro. La première action entreprise par le processeur consiste à effectuer un cycle d'accès mémoire afin de lire le code de la première instruction à l'adresse 0 en mémoire. Le processeur mémorise ce code (H'103C) dans son registre d'instruction, le décode et l'interprète. Le compteur ordinal (PC) est ensuite mis à jour afin de pointer à la prochaine instruction.

Le décodage de l'instruction H'103C indique au processeur qu'il s'agit d'une instruction de transfert de la valeur immédiate dans le registre de données D0.

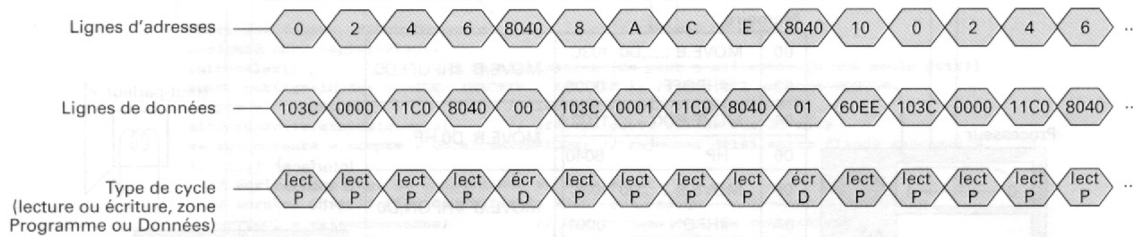
L'exécution proprement dite d'une instruction peut s'effectuer implicitement à l'intérieur du processeur, sans exiger de cycles mémoire supplémentaire (par exemple lorsqu'une valeur immédiate est chargée dans un registre de données). Elle peut également exiger un ou plusieurs cycles d'accès mémoire, lorsqu'il s'agit d'écrire une valeur en mémoire ou à une position d'entrée-sortie. Par exemple lors de l'exécution de la 3ème instruction du programme *MOVE.B D0,HP*, il y a d'abord un cycle mémoire de recherche de code d'instruction, puis un cycle de recherche de l'adresse *HP* (cette adresse est placée dans le registre interne AD) et enfin l'exécution proprement dite de l'instruction par un cycle d'écriture de la valeur qui se trouve dans le registre *D0* sur l'adresse d'entrée-sortie





Pour simplifier, les signaux de contrôle sont omis

le diagramme temporel suivant donne la succession des cycles d'accès mémoire et entrées-sorties lors de l'exécution du programme BruitHP.



c) Architectures de microprocesseurs

L'architecture CISC

CISC = Complex Instruction Set Computer

è instructions complexes qui nécessitent plusieurs cycles horloge pour s'exécuter.

Exemple

REP MOVSB = déplacement de plusieurs octets d 'une adresse A vers une adresse B

var t1, t2 : array[1..10] of integer;
for i := 1 to 10 do t2[i] := t1[i];

L'architecture CISC (*Complex Instruction Set Computer*, ce qui signifie "ordinateur avec jeu d'instructions complexes") est utilisée par tous les processeurs de type x86, c'est-à-dire les processeurs fabriqués par Intel, AMD, Cyrix, ...

Les processeurs basés sur l'architecture CISC peuvent traiter des instructions complexes, qui sont directement câblées sur leurs circuits électroniques, c'est-à-dire que certaines instructions difficiles à créer à partir des instructions de base sont directement imprimées sur le silicium de la puce afin de gagner en rapidité d'exécution sur ces commandes.

L'inconvénient de ce type d'architecture provient justement du fait que des fonctions supplémentaires sont imprimées sur le silicium, d'où un coût élevé.

Aujourd 'hui : Instructions CISC + noyau RISC

L'architecture RISC

RISC = Reduced Instruction Set Computer
(Ordinateur à jeu d'instructions réduites)

è 1975 David Patterson = 80 % des instructions d'un programme n'utilisent que 20 % des instructions du CPU

è En réduisant le jeu d'instructions on simplifie le décodage et donc l'architecture => temps de développement (time to market) plus court

è Principe de l'exécution d'une instruction par cycle horloge

è Problèmes liés au report de la complexité sur le logiciel => compilateur

La creation du RISC vient d'une critique des compilateurs, et non pas des processeurs CISC. Les compilateurs n'etaient pas capables de bien d'utiliser le jeu d'instructions d'un CISC. Faire un bon compilateur RISC est plus simple qu'on bon compilateur CISC.

Contrairement à l'architecture CISC, un processeur utilisant la technologie RISC (*Reduced Instruction Set Computer*, dont la traduction est "ordinateur à jeu d'instructions réduit") n'a pas de fonctions supplémentaires câblées. Cela impose donc des programmes ayant des instructions simples interprétables par le processeur. Cela se traduit par une programmation plus difficile et un compilateur plus puissant.

Pipelining

objectif : faire en sorte que l'UAL n'ait pas à attendre de données, temps d'exécution / k, problèmes liés aux dépendances.

A1 = **Fetch** (aller chercher le code de la prochaine instruction)
A2 = **Decode** (analyse du code pour savoir quelle opération exécuter)
A3 = **Execute** (exécuter l'instruction => UAL)
A4 = **Write back (**écriture du résultat en mémoire ou registre)

Exécution séquentielle classique

A1	A2	A3	A4	A1	A2	A3	A4	

Exécution séquentielle (Pipeline 1 étage)

A1	A1	A1	A1	A2	A2	A2	A2	

Exécution séquentielle (Pipeline 4 étages)

Fetch	A1	A2	A3					
Decode		A1	A2	A3				
Execute			A1	A2	A3			
Write				A1	A2	A3		

è exploiter le parallélisme au niveau des instructions
è traiter plusieurs instructions par cycle
è dupliquer le nombre d'unités de traitement (2 UAL, 2 UVF)
è limite atteinte pour 5 à 6 instructions à exécuter en parallèle

Processeurs Vectoriels

Les registres du processeur sont des vecteurs
Les instructions sont de type SIMD (Single Instruction Multiple Data)

Exemple : somme de deux vecteurs
var a, b, c : array[1..10] of integer;
for i := 1 to 10 do c[i] := a[i] + b[i];
Vectoriel :
c := a + b;

VLIW = Very Long Instruction Word

è Il s'agit de faire tenir plusieurs instructions en une seule
è Par exemple on code sur 128 bits plusieurs instructions qui seront exécutées en parallèle

MMX = Multimedia

Les unités MMX sont intégrées au CPU et sont de type SIMD (Single Instruction Multiple Data).

è Les données sont stockées sur 64 bits (2x32,4x16,8x8) et peuvent être traitées en même temps
è L'unité MMX permet de traiter les données vidéo et audio beaucoup plus rapidement que l'UAL
è SSE = Internet Streaming SIMD (opération d 'imagerie 3D, accélération du téléchargement de fichiers audio et vidéo)

d) Le concept de Pile

Une pile permet de stocker en mémoire de nombreuese valeurs successives. Ces valeurs sont à relir dans l'ordre inverse de leur sauvegarde : structure LIFO :Last In First Out.

L'instruction *PUSH.X D1* équivalente à l'instruction *MOVE.X D1,-(A7)* sauve le registre *D1* dans la pile. A cette fin, le pointeur de pile (A7) ets décrémenté de 1,2 ou 4 pour respectivement sauver sur la pile un octet de 8 bits (.B), un mot de 16 bits (.W) ou un long mot de 32 bits (.L) du registre considéré.

PUSH.L D1 ⟷ MOVE.L D1,-(A7)

Le pointeur de pile pointe à une nouvelle position en mémoire.

L'instruction *POP.X D1* équivalente à l'instruction *MOVE.X (A7),D1* permet de récupérer une valeur précédement sauvée sur la pile. Le pointeur de pile A7 est incrémenté afin de pointer à la prochaine valeur sur la pile.

POP.W D1 ⟷ MOVE.W(A7)+, D1

En résumé, le pointeur de pile est initialisé à la valeur max : quand on empile on décrémente le compteur de pile et quand on dépile on incrémente le compteur de pile.
La pile croit en direction de positions mémoire d'adresses décroissantes.

1.3. Les Interruptions

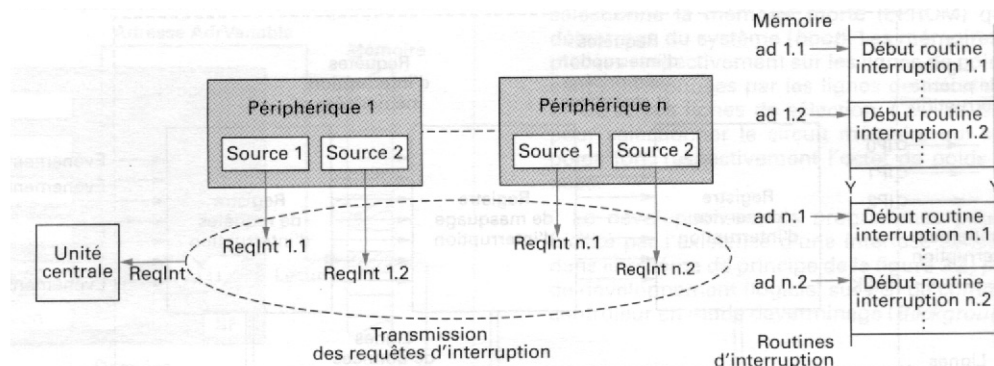
a) Principes

Afin d'agir en réponse à un événement extérieur sans nécessiter de scrutation continue, les processeur possèdent un **mécanisme d'interruption**. Celui-ci offre la possibilité de *modifier* le déroulement du programme lors de l'occurrence de l'événement extérieur. L'adjonction au processeur d'une **ligne d'interruption permet**, lorsque cette ligne devient active, d'engendrer une requête d'interruption. Le processeur appelle alors automatiquement une procédure appelée **routine d'interruption** qui se trouve à une position prédéfinie indiquée via le vecteur d'IT..

On distingue deux type d'inerruptions :

- les interruptions provenant de périphériques ouvert sur le monde exterieur (reset, bouton poussoir, clavier, changement d'état d'un signal exterieur, communication serie,…

- les interruptions provenant de périphériques internes (timer,passage par zero d'un compteur interne,..)



Tout se passe comme ci une instruction CALL était insérée juste après l'instruction en cours au moment où la requête d'interruption devient active. La routine d'interruption doit s'effectuer de manière transparente: après retour au programme interrompu l'état du processeur doit être exactement le même qu'avant l'interruption. Ceci implique que la routine d'interruption doit sauver au début de son exécution tous les registres internes y compris le registre d'état contenant les fanions (flags). Cette opération est appelée **sauvegarde du contexte**.

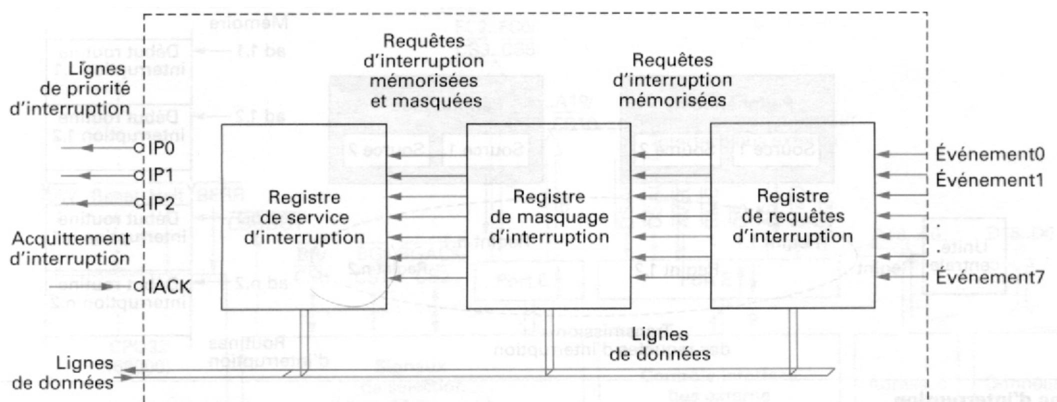
A la fin de la routine d'interruption, une instruction spéciale de retour de routine d'interruption restaure le registre de fanions et d'état ainsi que le compteur ordinal avec les valeurs sauvegardées sur la pile lors de l'appel de la routine d'interruption.

Le processeur, lors d'une interruption, demande à l'interface d'entrée-sortie ayant déclenché l'interruption de s'identifier lors d'un cycle **d'acquittement d'interruption**. L'identificateur

de l'interface ayant déclenché l'interruption est transmis sous forme d'un octet sur les lignes de données. Cet octet qui identifie la source de l'interruption et donc la routine d'interruption est intitulé le **vecteur d'interruption**. Une correspondance est établie entre la valeur du vecteur d'interruption et la case mémoire (fixée par le constructeur) qui contiendra l'adresse du début de la routine à exécuter.

b) gestion des IT

Le mécanisme de prise en compte des interruptions qui se trouve dans le processeur empêche une nouvelle interruption de niveau identique à celui de l'interruption précédente de venir interrompre l'exécution de la routine d'interruption.



Au cas où 2 interfaces d'entrée-sortie interrompent simultanément le microprocesseur, un arbitrage a lieu entre les deux interfaces concernées afin q'une seule d'entre elle place sont vecteur d'interruption sur les lignes de données pendant le cycle d'acquittement de l'interruption. L'autre vecteur est alors empilé dans le système de gestion des interruption pour être exécuter a la fin de la routine d'interruption en cours.

Schématiquement, les principaux registres internes associés au contrôle des interruptions sont
le **registre de mémorisation de requete d'interruption**
le **registre de masquage d'interruption** (Interrupt Mask Register) qui permet d'autoriser ou d'inhiber de manière sélective une ligne de requête d'interruption.
le **registre de service d'interruption** qui mémorise les vecteurs d'interruption associées aux différentes sources d'interruption et qui indique l'interruption actuellement en service.

c) conclusion

Un système d'interruption doit être capable de :

- Transmettre les requêtes d'interruption en fonction de leur priorités respectives, tout en mémorisant les requêtes pendantes.

- D'assigner à chaque requête d'interruption l'adresse du début de la routine d'interruption correspondante.

Les parametres important d'un système d'interruption sont :

- nombre de sources de requêtes d'interruption

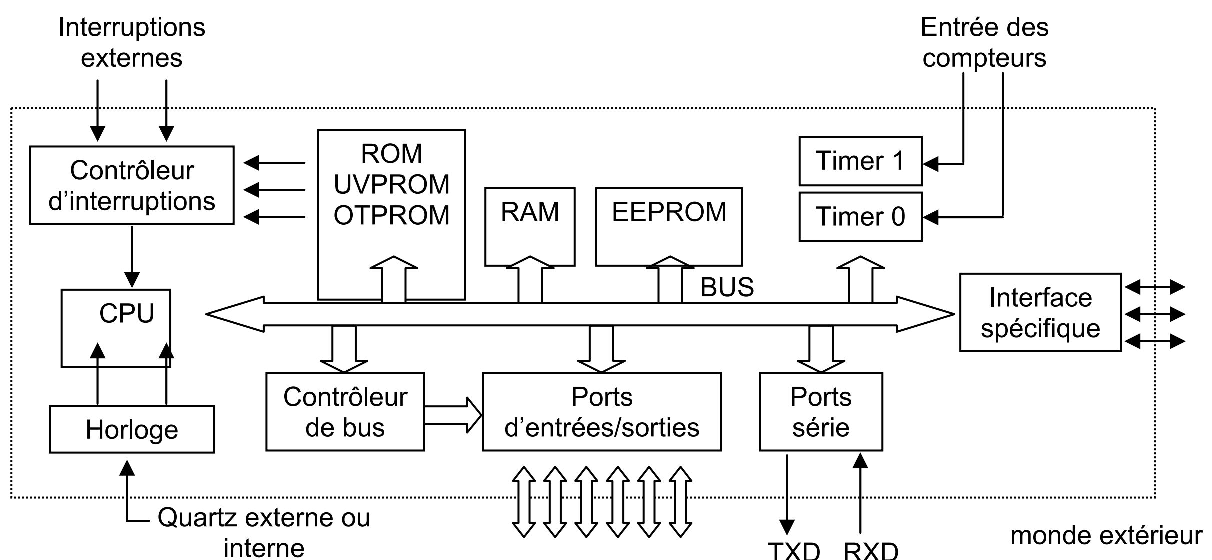
- fréquence des requêtes d'interruption
- temps pris par l'exécution d'une interruption
- délai entre la requête et l'exécution de la routine d'interruption

II. Microcontrôleurs

2.1. Architecture d'un Microcontrôleur

Le microcontrôleur est un dérivé du microprocesseur. Sa structure est celle des systèmes à base de microprocesseurs. Il est donc composé d'une unité centrale de traitement et de commande (équivalente au microprocesseur), de mémoires et de ports d'entrées/sorties.

En plus de cette configuration minimale, les microcontrôleurs sont dotés d'un ou plusieurs systèmes de comptage (TIMER). Quelques uns sont dotés d'un convertisseur analogique/numérique (CAN) intégré. Ces atouts supplémentaires permettent de nombreuses applications telles que :
- acquisition et traitement de données analogiques (CAN)
- comptage d'événements (TIMER)
- mesure de fréquence ou de période (TIMER)
- génération d'impulsions (TIMER)
    - les programmes peuvent être différents (gestion d'un thermostat intelligent, d'une photocopieuse..)
    - les programmes ont en commun peut de calculs complexes contrairement à un système informatique)



Un système minimal programmable pour fonctionner à besoin :

- d'une unité centrale
- de mémoires *morte* pour le programme (PROM, EPROM,….)
- de mémoires *vive*, pour les calculs, pour stocker les données,..
- de circuits interfaces, pour connecter les périphériques qui vont permettre la communication avec l'extérieur

d'où l'apparition des microcontrôleurs (ou Monochip)
Dans un seul circuit on va trouver :

- Une Horloge (oscillateur)
- Un processeur (Unité centrale)
- De la mémoire vive (RAM)
- De la mémoire morte (PROM, EPROM, EEPROM)
- Des interfaces qui vont dépendre du type de microcontrôleurs choisi
    - Compteurs/Timer
    - Convertisseurs Analogiques/numériques (C.A.N.)
    - Chien de garde (« Watch Dog »)
    - Gestion d'un port parallèle (d'entrée/sortie)
    - Gestion d'une liaison série RS232
    - Gestion des interruptions
    - Gestion des moteurs en PWM (pulse w modulation)
    - Gestion d'écran LCD
    - Gestion de bus I2C
    - Etc…

Il suffit de choisir le microcontrôleur le mieux adapté à l'application que l'on doit réaliser. !

la ROM contient le programme à exécuter

contrairement à un système informatique classique, il n'est pas question de charger un programme en mémoire vive à partir d'un disque ou d'une disquette car l'application doit démarrer dés la mise sous tension et ne possède pas d'organe de ce type.

la RAM ou mémoire vive

(Random access memory :mémoire à accés aléatoire)

On les appel comme ça de façon impropre LES ROM sont aussi a accès aléatoire
Ces mémoires perdent l'information lorsqu'elles ne sont plus alimentées.

Pour pouvoir travailler normalement le µcontrôleur doit pouvoir a souvent besoin de stocker des données temporaire qque part et c'est la qu'intervient la RAM.

Contrairement a un système informatique classique la RAM d'un µcontrôleur est de petite taille.

Les entrées sorties

les circuits d'interfaces peuvent piloter des matériels très différents, (moteur pas à as, afficheur LCD, thermistance, communication avec des pc ou d'autres µcontrôleurs, etc…)

Le bus système

L'unité centrale doit pouvoir communiquer avec les mémoires et les périphériques.

Exemple : pour écrire une donnée en mémoire, l'UC doit d'abord spécifier l'adresse de la mémoire, puis envoyer la donnée, et en dernier lieu, envoyer un signal qui validera la mémorisation de la donnée. Tous ces signaux seront véhiculés par les « bus », ensembles de « conducteurs », sur lesquels viennent se brancher les mémoires, les interfaces des périphériques.

On distingue 3 types de bus

- le bus d'adresses
- le bus de données
- le bus de contrôle (pour les signaux de service)

Les différents types de mémoires dans les microcontrôleurs

Le prix du microcontrôleur dépend fortement du type de mémoire qu'il contient. Outre les différents périphériques possibles, les différents types de mémoire constituent les différentes gammes de microcontrôleurs de même architecture.

La ROM qui contient le programme à exécuter (plusieurs kilo octets)
ne s'efface pas hors tension

1. **PROM**. (Programmable Read Only memory)

On les appelle aussi mémoire fusibles ou **OTP** (One Time Programmable)
La programmation de ce type de mémoires consiste à faire « claquer » des fusibles (qui sont en fait des jonctions semi-conductrices)

2. **EPROM** (Erasable Programmable Read Only Memory) **ou UVPROM**

Ce sont des mémoires programmables électriquement et effaçable par UV donc réutilisables. Il faut noter que l'effacement par UV (environ 15 min) et l'écriture (qques minutes) sont des opérations relativement longues.
Nécessite une petite fenêtre en quartz pour laisser passer les UV.
(principe utilisé :utilisation de transistors FAMOS : Floating Gate Avalanche Injection Metal Oxide Silicium)
⇒ apparition d'**OTPROM** qui sont des UVPROM sans fenêtre et donc non réutilisable mais pas chère.

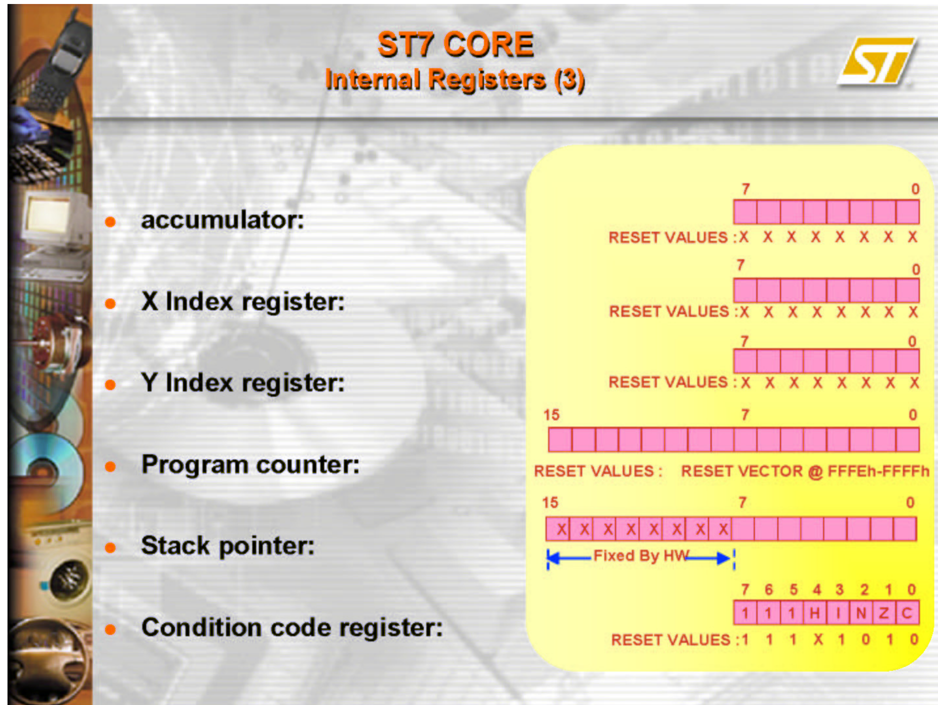
3. **Autres**

EEPROM (Erasable Programmable Read Only Memory)
Ce sont des mémoires programmables et effaçables électriquement, ou aussi mémoires **FLASH**. Procédé beaucoup plus rapide que lors d'une exposition UV.

la RAM (mémoire des données, variables, piles,etc..) **(quelques octets)**
s'efface hors tension.

2.2. Etude du microcontrôleur 8 bits ST7

a) Les registres internes



L'accumulateur (A) (Registre de donnée)

Registre où sont effectués les calculs arithmétique et logiques.
Pour effectuer une opération en mémoire il faut d'abord copier la première dans l'accumulateur, puis effectuer l'opération entre l'accumulateur et la deuxième valeur. Le résultat peut alors être recopié de l'accumulateur vers la mémoire.

Les registres d'index (X et Y) (Registre d'adresse)

Ces registres contiennent des adresses. Avoir 2 registres d'adresse permet de gérer les déménagement de mémoire à mémoire. Sinon on peut les utiliser pour stocker des données temporaires

Exemple
D[k]=C[j]+B[i] ;

```
LD X,i
LD A,([B],X)
LD X,j
ADD A,([C],X)
LD X,k
LD ([D],X),A
```

Le compteur de programme (PC)

Ce registre commande le séquencement des instructions. PC contient l'adresse de la prochaine instruction à exécuter.
PC s'incrémente automatiquement avec une constante (x octets = code instruction + code opérande) sauf en cas de saut ou branchement (PC se charge alors avec l'adresse de début de la routine appelée suite à un CALL par exemple).

Le pointeur de pile (SP)

La pile est une zone de mémoire vive utilisée par les instructions CALL, RET et PUSH , POP
Quand une valeur est empilée, SP est décrémenté et la valeur est écrite à l'adresse située immédiatement en dessous de la dernière valeur empilée.

L'empilement décrémente et le dépilement incrémente permet l'imbrication d'appel de sous programme.
Exemple : 1 ss. prog. peut appeler un ss. prog. qui en appelle 1 troisième. 3 adresses de retour sont empilées (3 call) : le premier RET fera retourner la troisième à la deuxième, etc…jusqu'au prog principal

Cas des interruptions
Empilent l'adresse de retour, mais aussi le contenu de tous les registre (A,X,CC) sauf Y. Le retour d'interruption se fait par une instruction spéciale IRET, qui dépile toutes les valeurs et remplace les registres d'origine.

Initialisation du pointeur de pile
Au dévbut du prog le pointeur de pile doit être initialisé. RSP qui met SP à une valeur qui dépend du modele de ST7.

Le registre code condition (CC)

1 1 1 H I N Z C

les bits de ce registre sont mis à jour après chaque opération.

Le bit C
C'est le bit de retenu (addistion ou soustraction)
Exemple : addition 16 bit
1200 + 6230
4B0 + 1856 = 1D06
B0 + 56 = 106 (hexa) 1 est la retenu
4 + 18 = 1C, 1C+1=1D

pour cela la première instruction utilse ADD, et toutes les autres ADC
IDEM pour la soustraction avec SUB et SUBC

Le bit Z
Il permet de savoir si le dernier résultat était zero ou non

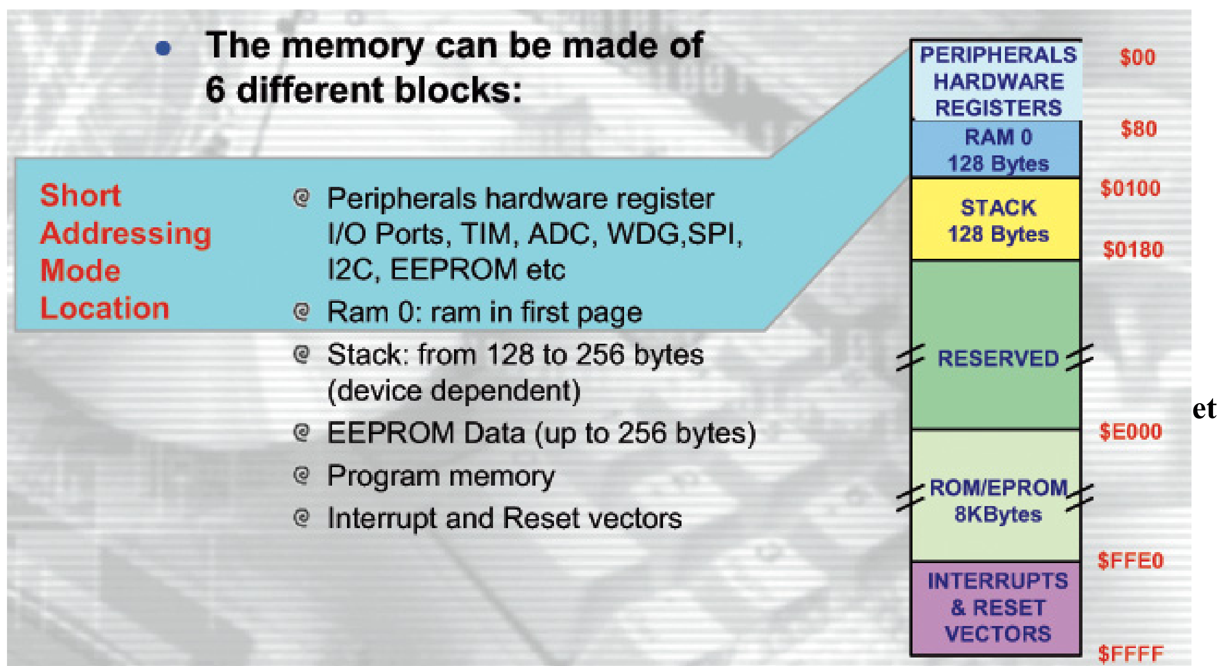
Le bit N
Il permet de savoir si le dernier résultat était négatif ou non

Le bit I
C'est le masque global d'interruptions. Quand il est à 1 toutes les demandes d'interruption sont ignorées. Si une demande est en attente son passage à 0 déclenche l'interruption.

Le bit H
Il reflète la retenu entre le 4ème et le 5ème bit d'un octet. Utilisé en arithmétique décimale codée binaire (BCD) chaque groupe de 4 bits contient un digit décimal, mais le groupe suivant à une valeur seulement 10 fois plus grande au lieu de 16. chaque groupe peut prendre les valeurs 0 à 9 , les valeur 10 à 15 étant interdites.
Utilité de l'arithmétique BCD : évite les conversions décimale en binaire et *vice versa*.

b) L'organisation mémoire (memory map)

ST7 : Architecture de type **von Neuman**, donc un seule espace adressable continu.



et

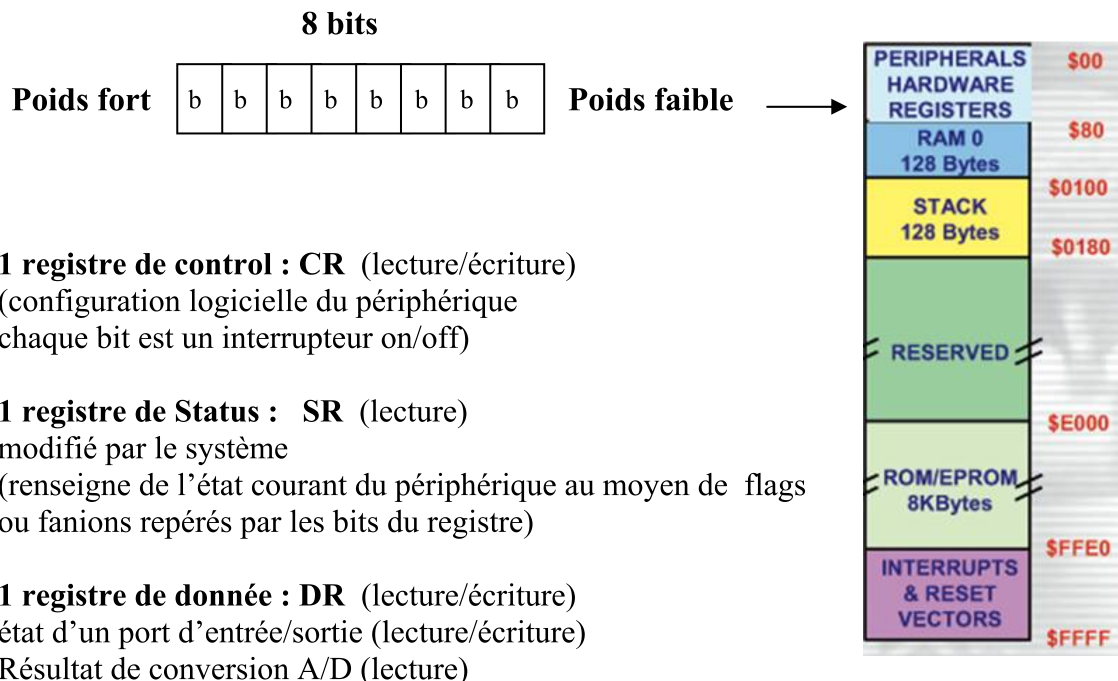
c) Le jeux d'instruction

Architecture RISC, le jeu d'instruction est cependant assez riche. Il ne manque que la division. Les modes d'adressage sont nombreux et disponibles sur presque toutes les instructions élémentaires.

2.3. Les Périphériques du ST7

2.3.1. Programmation et configuration des périphériques

Les registres des périphériques matériels



1 registre de control : CR (lecture/écriture)
(configuration logicielle du périphérique
chaque bit est un interrupteur on/off)

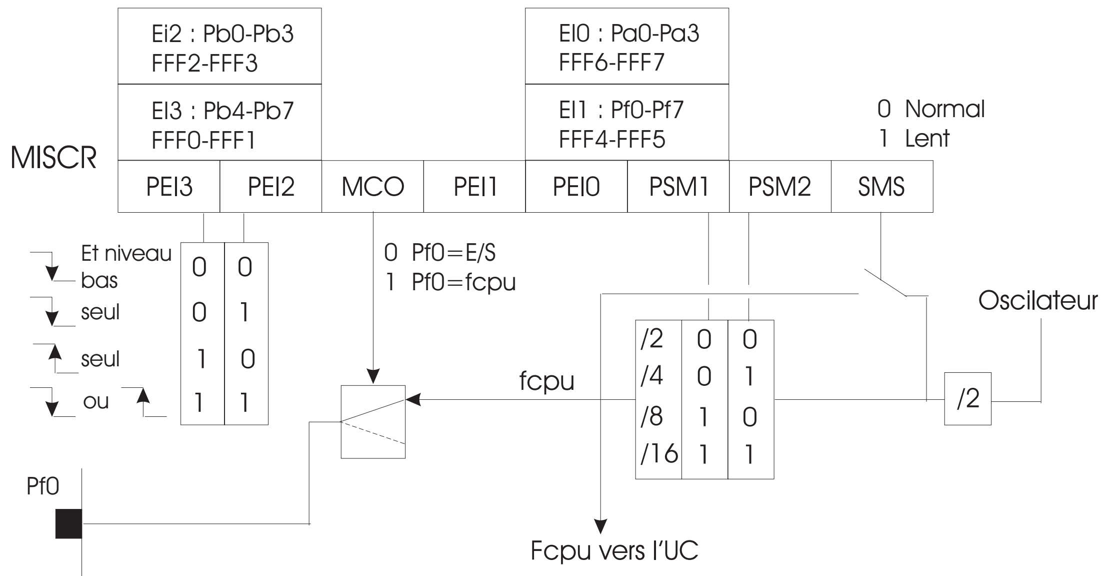
1 registre de Status : SR (lecture)
modifié par le système
(renseigne de l'état courant du périphérique au moyen de flags ou fanions repérés par les bits du registre)

1 registre de donnée : DR (lecture/écriture)
état d'un port d'entrée/sortie (lecture/écriture)
Résultat de conversion A/D (lecture)

a) Le registre Miscellanaeous du ST72311

Le cœur du ST7 est cadencé par une horloge interne qui provient de la division de la fréquence de l'oscillateur. Le rapport de division est programmable, ce qui permet de sélectionner le meilleur compromis entre vitesse et consommation d'énergie.

Ce choix est fait à l'aide de plusieurs bits du registre miscellaneous. Ce registre contient aussi des bits servant à d'autres usages ; la répartition des bits de ce registre diffère selon la variante de ST7 considérée. Nous donnons ici les fonctionnalités du ST72311 :



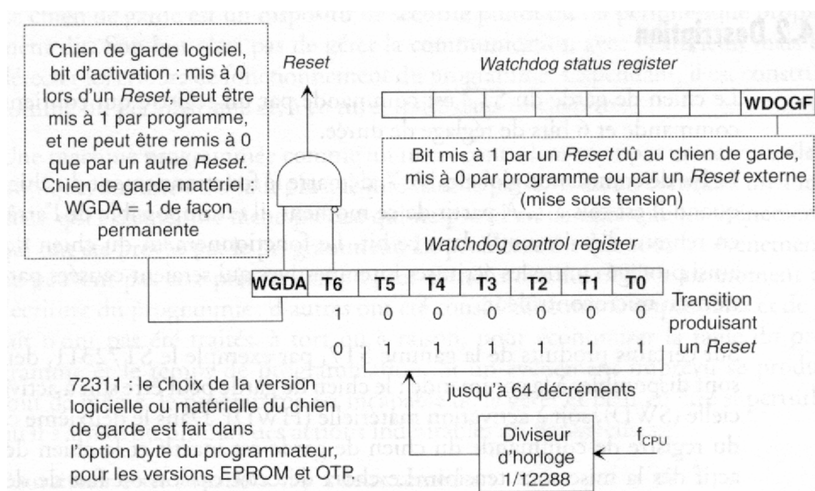
b) Le chien de garde du ST72311

Le chien de garde est plutôt 1 dispositif de sécurité qu'un périphérique.
Il est mis en œuvre pour, non pas éviter un plantage, mais relancer le programme du microcontrôleur (reboot)

Le plantage peu provenir d'une agression électromagnétique ou d'un bug du programme provenant de la non prise en compte d'événement matériels extérieurs ou souvent de leurs enchaînements.

A l'achat du microcontrôleur on peut spécifier 2 options :

- chien de garde à activation logicielle
- chien de garde à activation matérielle



Une fois le programme au point, on peut fixer la valeur du compteur à la moitiée du max et ensuite approcher par la valeur optimale par **dichotomie**.
Si le chien de garde déclenche trop tôt on double la valeur, si le reboot n'est pas gênant pour l'appli, on prend la moyenne etc,..de façon à trouver la plus petite valeur qui permet à l'appli de fonctionner. Une fois cette valeur trouvée, il est de bon ton de la multiplier par un coefficient de sécurité (gestion des imprévus) !

2.3.2 Les Timers

Le st7 dispose de 2 circuits Timer 16 bits (Timer A et Timer B) indépendants et proposant les mêmes fonctionnalités.

Chacun permet par exemple :
Le comptage d'impulsions
La mesure de fréquence
La mesure d'intervalles de temps
LA génération d'impulsions, isolées ou périodiques

L'intérêt d'un tel périphérique est qu'il traite ces opérations sans l'intervention du cœur !!

Horloge du temporisateur

2 sources possibles :
- horloge interne
- horloge externe (sur une broche spécifique d'un port parallèle)

En cas d'utilisation de l'horloge interne, on peut configurer différentes fréquences de fonctionnement qui vont prédiviser la fréquence du cpu. Pour cela on utilise un prédiviseur paramétrable au moyen des bits **CC1** et **CC0** du registre de configuration du timer TACR2 pour le timer A et TBCR2 pour le timer B.

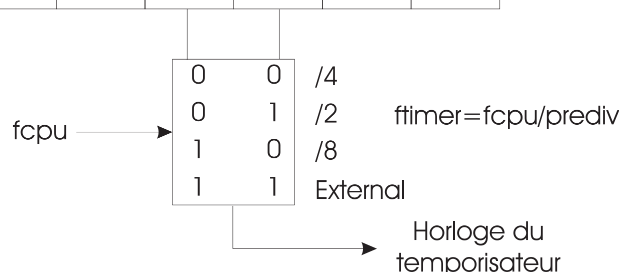
2 registres de contrôle : TACR1 et TACR2

TACR1

ICIE	OCIE	TOIE	FOLVL2	FOLVL1	OLVL2	IEDG1	OLVL1

TACR2

OC1E	OC2E	OPM	PWM	CC1	CC0	IEDG2	EXEDG



1 registre de status : TASR

TASR

ICF1	OCF1	TOF	ICF2	OCF2	0	0	0

Exemple de la résolution du temporisateur avec un Quartz de 16 Mhz pour chaque taux de prédivision :

Prédiviseur Horloge de la CPU	**Prédiviseur Horloge du temporisateur**		
	1/2	1/4	1/8
1/2 (rapide)	0.25 s	0.5 s	1 s
1/32 (lent)	4 s	8 s	16 s

Les Compteurs perpétuels TACR ou TBCR (16 bits)

TACR est composé de 2 mots de 8bits chacun notés **TACHR** (HIGH : 8 bits de poids fort) et **TACLR** (LOW :8 bits de poids faible) qui s'incrémente automatiquement à la fréquence du temporisateur.

Lecture des compteurs : Attention a l'ordre de lecture !!!.
Comme on ne peut procéder qu'a 2 lectures successives il n'y a pas de simultanéité.
Toujours commencer par les poids forts TACHR. L'accès en lecture de ce registre entraîne un recopie automatique de TACLR dans un verrou, et quand on accède à TACLR c'est en fait la valeur du verrou qui est renvoyé.

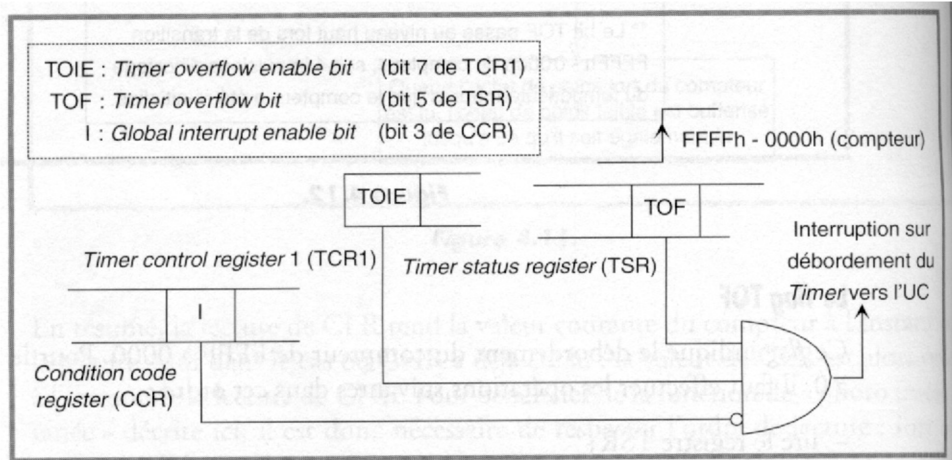
Remise à 0 du compteur (en fait pas 0 mais FFFC = -4)
Toute écriture dans TACLR remet le compteur à FFFC

a) Débordement du compteur perpétuel et son interruption associée

Flag de débordement
Le bit TOF du registre de status TASR indique le débordement de FFFF à 0000 soit 4 impulsions après l'initialisation du compteur.
Le bit TOF passe à 1 si débordement il y a eu.

Pour remettre le bit TOF à 0 il faut lire TASR et lire ou écrire dans TACLR

Interruption sur débordement
Pour générer une interruption sur débordement il faut mettre le bit TOIE du registre de control TACR1 à 1. l'interruption se déclenche alors quand les bits TOIE et TOF sont à 1 et quand le bit I du registre code condition est à 0.

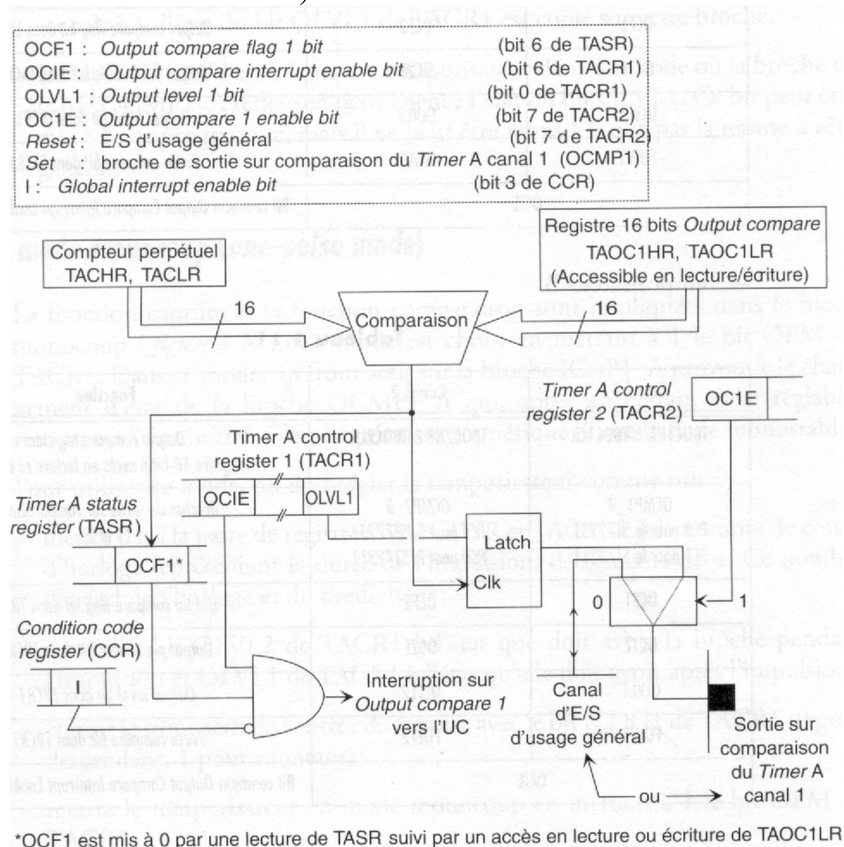


b) La fonction de comparaison et son interruption associée

Mécanisme qui produit un événement quand la valeur courante du compteur est égale à la valeur d'1 des 2 registres de comparaison TAOC1HR TAOC1LR ou TAOC2HR TAOC2LR.

On peut associer a cet évènement :

- 1 demande d'interruption
- 1 changement d'état sur 1 broche dédiée en sortie
- la remise au départ automatique du compteur perpétuel (seulemnet en mode PWM)



Si OC1E est à 1 la valeur du bit OLVL1 est copiée sur la broche au moment de l'évènement de comparaison..
Le flag OCF1 est mis à 0 par lecture de TASR puis lecture ou écriture de TAOC1LR

c) La capture et son interruption associée

Elle permet de prendre une photo de la valeur du compteur au moment d'une transition de signal détectée à une broche dédiée.

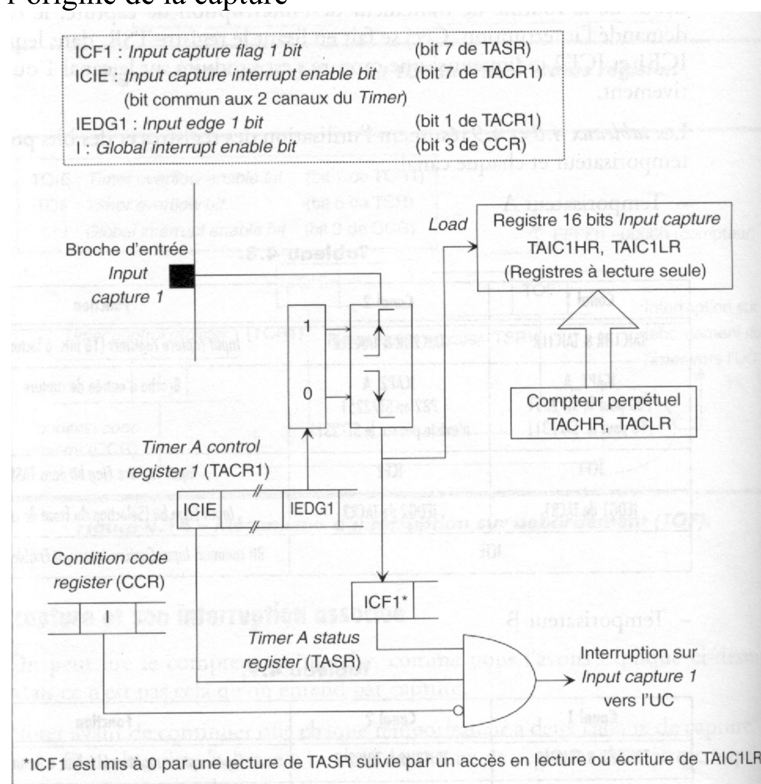
La transition de signal déclenchant la capture est définie par le bit IEDG1 : front montant (IEDG1=1) ou descendant (IEDG1=0) :

Quand la transition adéquate se produit sur la broche d'entrée input capture 1 :

- la valeur du compteur est copié dans les registres TAC1HR et TAC1LR
- le flag ICF1 du registre de status TASR est mis à 1

Pour chaque Timer il existe 2 captures possibles (2 broches dédiées, input capture 1 et input capture 2)

L'interruption de la capture
Si le bit ICIE (input capture interuption enble) est à 1 le flag ICFi (i = 1 ou 2) déclenche l'interruption
ICIE est commun aux 2 flags donc il est nécessaire d'effectuer dans la routine d'interruption un test des bits ICF1 et ICF2 du registre de status TASR pour connaître l'origine de la capture

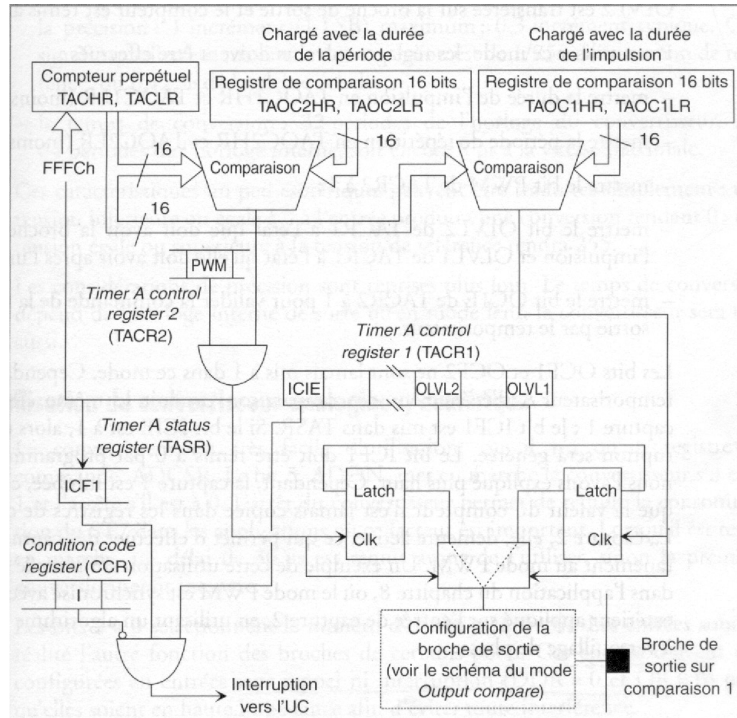


*ICF1 est mis à 0 par une lecture de TASR suivie par un accès en lecture ou écriture de TAIC1LR

Remise à 0 du flag de capture ICFi

- lire TASR
- lire ou écrire dans TAIC**i**LR

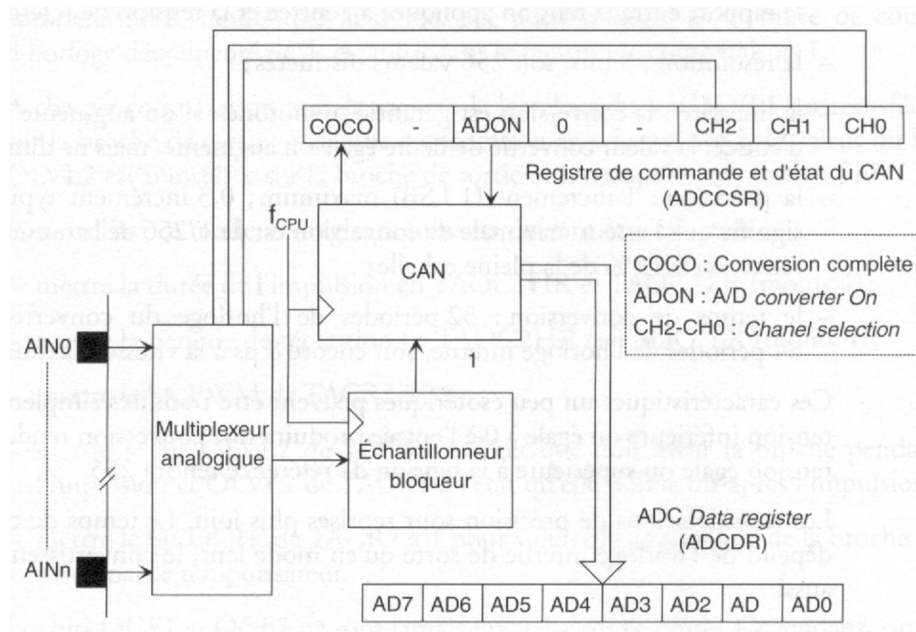
d) Le mode PWM



2.3.3. Le Convertisseur analogique/numérique (CAN)

Description **:**

- tensions d'entrées positives
- conversion radiométrique (résulat = rapport entre la tension appliquée et la tension de référence)
- 8 bits de résolution
- linéaire
- précision : 1 incrément max (1LSB) ; 0.3 incrément typique
- temps de conversion : 32 pèrode de l'ghorloge du convertisseur = 64 périodes de l'horloge interne
  (8 s à vitesse max du cpu c.a.d 16/2 Mhz)



Toute écriture dans le registre de control/statuts ADCCSR (un seul registre) stop la conversion en cours.

Les ports d'entrée/sortie AIN0 à AIN7 doivent être configurées en entrée (PADDR=0) et sans interruptions (PAOR=0) pour être en haute impédance afin d'éviter toute interférence.

Avant de récupérer la valeur de la tension dans le registre ADCDR il faut s'assurer de la fin de la conversion, pour cela il suffit d'attendre que le flag COCO passe à 1.

Avant d'utiliser le CAN il faut attendre 30 s après avoir fixer ADON à 1.

2.3.4. Les ports d'entrées/sorties parallèles

Ils conditionnent le nombre de pattes du microcontrôleur
Ils sont les plus utilisées dans les montages à microcontrôleur car se sont des lignes dont l'état logique, haut ou bas, est directement contrôlable par programme. En sorties, les lignes sont généralement chargées de commander des relais, des triacs, des LED ou des afficheurs. Par mesure d'économie ou même parfois de "réalisabilité", on cherche toujours à réduire le nombre de ports d'entrées/sorties nécessaires.

Port	Configuration d'entrée (DDR = 0)			Configuration de sortie (DDR = 1)	
	OR = 0	OR = 1	Source d'interruption externe. Option de polarité	OR = 0	OR = 1
Port A : PA0-PA7	Flottant	Flottant avec interruption	EI0 PEI0-PEI1	Drain ouvert, fort courant	(Option interdite)
Port B : PB0-PB7	Flottant	*Pull-up* avec interruption	EI1 PEI2-PEI3	Drain ouvert	Symétrique
Port C : PC0-PC5	Flottant	*Pull-up* avec interruption		Drain ouvert	Symétrique

Configuré en sortie

2 montages : symétrique (totem-pole), collecteur ouvert (open drain)

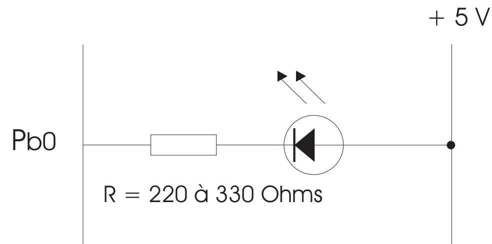
Comportant essentiellement deux transistors qui ne sont jamais passant en même temps. Il existe un autre type de montage de l'étage de sortie où seul le transistor cablé à la masse (transistor "du bas") est présent. Le signal de sortie est à prendre sur le collecteur de ce transistor, d'où l'appelation de *collecteur ouvert* pour un tel montage. Ce transistor se comporte comme un interrupteur à la masse, ouvert ou passant, correspondant respectivement aux états de sortie H ou L, dans le montage totem-pole. Pour retrouver un signal logique 5V/0V en sortie du montage à collecteur ouvert, il faut rajouter une résistance de tirage à 5V.

De nombreux microcontrôleurs proposent des ports qui peuvent servir d'entrée et de sortie par souci d'économie de place. On par exemple commander un afficheur à LED est scruter un clavier sur un unique port. La lecture du clavier peut être faite pendant des temps morts aménagés entre deux commandes d'affichage.

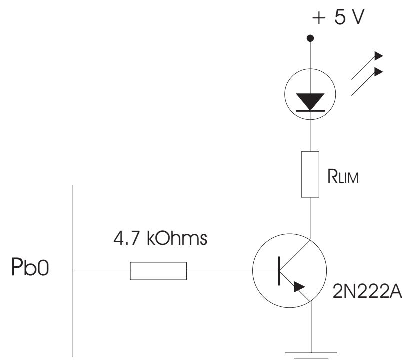
L'affichage

Si le microcontrôleur dispose de sorties à fort courant (10mA), on peut connecter directement une LED avec une résistance de limitation de courant à la broche du circuit. C'est le cas de quelques ports de la famille 6805 de Motorola ou de la famille ST7 de STMicroelectronics. Si ces ports n'existent pas, il reste la solution du banal transistor à intercaler entre la LED et le contrôleur.
Dans le cas où on utiliserait plusieurs afficheurs 7 segments à LED, on doit réaliser un multiplexage. Les afficheurs sont allumés tour à tour. Si les pas de programmation sont assez rapides, La persistance rétinienne donne alors l'impression d'un affichage continu. Les segments peuvent être connectés directement sur les sorties du microcontrôleur ou si on ne dispose pas assez de lignes de sorties, on peut faire appel à un décodeur BCD / 7 segments.
Lorsque pour des problèmes d'autonomie, de dissipation de puissance ou de consommation se posent, on préférera recourir à un afficheur à cristaux liquides. Ils sont cependant plus difficiles à mettre en œuvre. De nombreux fabricants intègrent désormais une interface spécifique pour commander ces afficheurs LCD.

LED (Sortie fort courant (High sink)) : utilisation d'une résistance de limitation



LED (Sortie normal) : utilisation d'un transistor 2N222A



Afficheurs 7 segments multiplexés

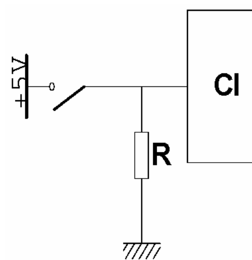
Commande de puissance

Transistor MJ 3001

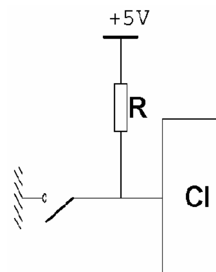
Montage Relais

Les interrupteurs – lecture de bouton poussoir

la résistance de tirage au 0 volt (**push pul**l) permet d'imposer un 0 logique sur l'entrée du circuit TTL.



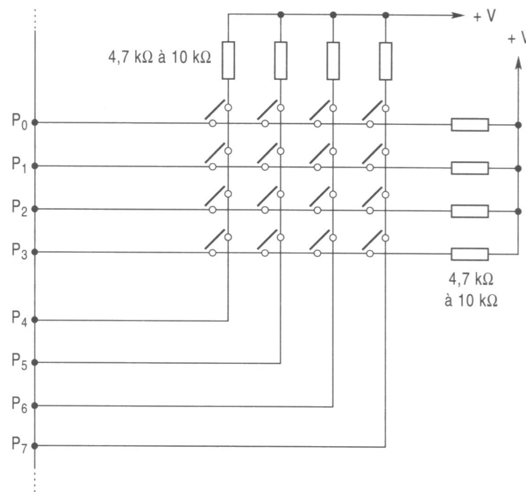
la résistance de tirage au 5 volt (**pull up**) permet d'imposer un 1 logique sur l'entrée du circuit TTL.



Les claviers

Lorsqu'il s'agit de n'utiliser que quelques boutons poussoirs, il suffit de les connecter aux lignes d'entrées (un par ligne). Cependant, il est souvent indispensable de mettre en place un clavier ; il n'est alors plus possible de réserver une ligne par bouton. Une solution consiste à faire appel a un encodeur de clavier externe qui recevra N touches en entrées et fournira un code sur P bits en sortie avec N= 2^P. Ainsi, 16 touches peuvent être codées sur 4 bits. Une autre solution, celle du clavier en matrice n'utilise pas de circuit externe. Comme le montre la figure, les touches sont placées à l'intersection des lignes et des colonnes du quadrillage réalisé par les deux groupes de quatre fils. Il suffit alors de localiser le court-circuit entre une ligne et une colonne lorsqu'une touche est enfoncée. Cette localisation se fait en deux temps (logiciel). Les lignes sont mises au niveau logique bas et les colonnes au niveau haut ; on connaît la colonne se trouve la touche utilisée. On inverse le rôle des lignes et des colonnes pour trouver la ligne. Connaissant la ligne et la colonne, on retrouve la touche enfoncée. Cette dernière solution est la plus intéressante puisqu'elle ne nécessite qu'une dizaine

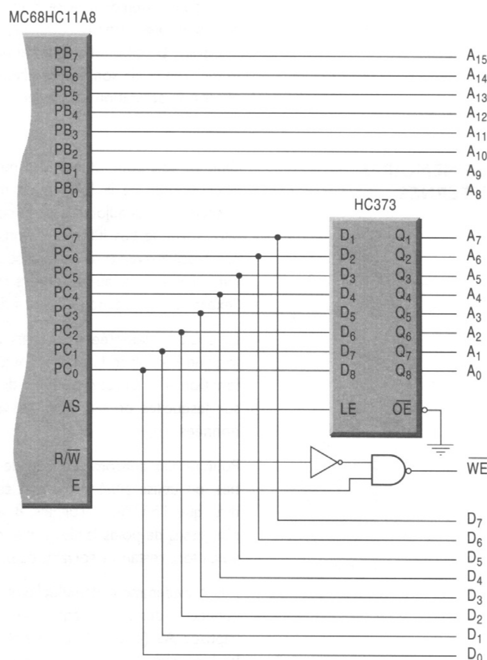
de ligne en assembleur, elle ne nécessite pas de circuit supplémentaire et réalise une économie de lignes d'entrées/sorties.



Mode de connexion d'un clavier câblé en matrice

Les mémoires externes

Que ce soit pour utiliser des microcontrôleurs sans ROM interne, pour ajouter de la RAM ou un circuit d'interface, il est parfois utile se faire sortir le bus interne. Cela n'est possible qu'avec les microcontrôleurs possédant un mode étendu. Dans ce cas, la solution la plus employée par les fabricants est de redéfinir certaines lignes d'entrées/sorties parallèles comme bus d'adresses et bus de données. Par économie, il arrive que les lignes d'adressage et de données soient multiplexées.



démultiplexage adresse/données sur un microcontrôleur Motorola

Exemple du MC68HC11A8 où le bus de données est multiplexé avec les 8 bits de poids faible du bus d'adressage sur le port C du microcontrôleur alors que les bits de poids fort du dus d'adressage sont disponibles sur le port B. La ligne AS (Address Strobe) sert à ce multiplexage.

2.4. Les interruptions

Remarque :
un seul vecteur d'interruption pour plusieurs événements possibles
Cas du timer : au niveau logiciel pour connaître l'origine précise de l'événement il faudra tester les différents fanions ou flag.

Source	Description	Registre	Adresse du Vecteur	Priorité
RESET	Reset	N/A	FFFEh-FFFFh	La plus haute
TRAP	Software	N/A	FFFCh-FFFDh	
EI0	PA0-PA7	N/A	FFFAh-FFFBh	
EI1	PB0-PB7 PC0-PC5	N/A	FFF8h-FFF9h	
	Non utilisé		FFF6h-FFF7h	
SPI	Transfer complete	SPISR	FFF4h-FFF5h	
	Mode fault			
TIMER A	Input capture 1	TASR	FFF2h-FFF3h	
	Output compare 1			
	Input capture 2			
	Output compare 2			
	Timer overflow			
	Non utilisé		FFF0h-FFF1h	
TIMER B	Input capture 1	TBSR	FFEEh-FFEFh	
	Output compare 1			
	Input capture 2			
	Output compare 2			
	Timer overflow			
	Non utilisé		FFECh-FFEDh	
	Non utilisé		FFEAh-FFEBh	
	Non utilisé		FFE8h-FFE9h	
	Non utilisé		FFE6h-FFE7h	
I2C	I2C Peripheral interrupt	I2CSR1 I2CSR2	FFE4h-FFE5h	
	Non utilisé		FFE2h-FFE3h	
	Non utilisé		FFE0h-FFE1h	La plus basse

c'est au moyen du fichier du type ***prm*** que au moment de l'édition le compilateur fait le lien entre le vecteur d'interruption et le nom de la routine d'interruption associée à l'événement et déclarée dans le programme. le vecteur d'interruption contiendra alors l'adresse du début de la routine (adresse codée sur 16 bits)

III. Utilisation du langage C

Pourquoi utiliser le langage C ?

L'intérêt principal :
Nécessité de dvper en langage évolué ; l'assembleur proche du langage machine est trop peu lisible est mal adaptés à une architecture complexe de programme.

Il existe de nombreux langage de haut niveau :
Certain dvpés pour des microcontrôleurs, d'autres largement répandus dans les ordinateurs ont étés adaptés aux besoins des microcontrôleurs.

C'est le cas du C.

Avantages :

- le C reste proche du matériel (mise à 1 ou 0 d'un bit d'un port d'entrée sortie)
- le C est structuré (prog. Facilement divisible en bloc = fonctions,..)
- 1 seule ligne de C peut produire plusieurs centaines d'instructions assembleur = simplification du code source, et fiabilité.
- Souplesse de travail de part la portabilité globale d'un programme en C (hors gestion entée/sortie)
- Portabilité entre différents types de microcontrôleurs de la même famille

Inconvénients :

- Le C destiné à l'origine pour de gros ordinateurs à mémoire centrale. Une même mémoire pour le programme, les données.
  Dans un microcontrôleur le programme dans une mémoire à lecture seule, les données dans une mémoire en lecture écriture mais volatile (modifiable par événement extérieur autre que programme). Le C n'a pas d'outils standard pour gérer ces différences.

- Les interruptions doivent et peuvent être écrites en C. les interruptions ne sont pas des concepts du langage C.

Les constructeurs fournissent les extensions pour fonctionner sur microcontrôleurs :

Modifications pour pouvoir fonctionner sur microcontrôleurs :

- Différentiation des propriétés des RAM et ROM pour intégrer la notion d'entrée/sortie

Exemple :
Le modificateur *volatile* permet à une donnée de pouvoir être modifiée par le programme, mais aussi par un évènement extérieur.
C'est le cas des registres d'entrées, des registres d'état d'un périphérique, etc..

```
volatile unsigned char PADR;      // port A data register
volatile unsigned char SPISR;     // SPI Status Register
```

- Implémentation particulière de chaque fabricant pour pouvoir mettre en œuvre les spécificités des microcontrôleurs.

Exemple :
La directive #pragma TRAP_PROC qui permet de déclarer les routines d'interruptions.
Les pointeurs *near* et *far* dans la définition des pointeurs pour optimiser l'adressage.

La première catégorie est normalisée et commune à tous les compilateurs.
La seconde est propre à chaque compilateur.
Plusieurs compilateurs existent pour le ST7.
Nous choisissons le compilateur Hiware.

3.1. Organisation mémoire du ST7 et options de compilation

Le programme est écrit en ROM, les variables sont écrites en RAM. Le programmeur doit spécifier dans un fichier de paramétrage, les adresses des zones mémoires afin que l'éditeur de liens sache placer chacun des segments composant le code.
Il doit contenir la *memory map*.

L'espace mémoire disponible est propre à chaque modèle de microcontrôleur.

Pour optimiser les accès mémoire, le compilateur Hiware propose 4 modèles d'organisation de la mémoire.

Adressage direct :
les variables sont stockées en page zero (Short Addressing) 0080h – 00FFh
c'est un adressage court sur un octet (avantage longueur du code généré et rapidité d'exécution)
Adressage étendu :
Les variables sont situées à des adresses ≥ 0100h

Modèle	**Variables locales**	**Variables globales**	**Résultat**
Small	Adressage direct	Adressage direct	Seules les données constantes ont un adressage étendu ou peuvent être adréssées par des pointeur *far* A n'utiliser que pour de petites applications
Small extended	Adressage étendu (>100h)	Adressage direct	
Large	Adressage direct	Adressage étendu	Mode par defaut, bien adapté au ST7
Large extended	Adressage étendu	Adressage étendu	Permet l'utilisation d'un grand nombre de variables, locales ou globales. Peu adapté aux memory map des ST7

3.2. Allocation des variables

3.2.1. La zone DEFAULT_RAM

par défaut les variables sont stockées dans la zone DEFAULT_RAM.
Certaine variables doivent cependant avoir des addresses fixes, commes les registres des périphériques ou autre variables forcées en page 0)

```
//ANSI C
#define VAR  (*(char *) (0x0010))     //L'Addresse de VAR est fixée à 0x0010

//ST7
#pragma DATA_SEG MY_RAM       //défini dans le fichier LINK (*.prm)
char VAR ;
```

quand elle doivent figurer en page 0 le compilateur doit le savoir

```
#pragma DATA_SEG   SHORT  MY_RAM          //défini dans le fichier LINK
char page0VAR ;
```

le linker alloue la mémoire séquentiellement dans l'ordre des déclaration et optimise l'espace mémoire quand une variable n'est pas utilisée.
Attention à la déclaration des registres matériels : dans le fichier de LMINK on doit rajouter + à chaque fichier objet

Volatile signifie que la variable peut être modifiée par le matériel

FILE.C

```
#pragma DATA_SEG SHORT REG_AREA
volatile char ADCDR;
volatile char ADCCSR;
#pragma DATA_SEG DEFAULT
….
```

FILE.PRM

```
NAMES
  FILE.O+
  ….
PLACEMENT
REG_AREA   in NO_INIT 0x0070  TO 0x0072
```

3.2.2. Les variables en page zéro

Codage sur un octet. Un nombre max de données devra être stocké en page zéro, plus particulièrement celle dont l'usage est le plus fréquent.

Pour affecter une variable en page zéro, deux méthodes :

- Création d'un segment code avec l'attribut SHORT :

```
#pragma DATA_SEG SHORT FAST_DATA
int FastVariable ;
```

- L'utilisation du segment prédéfini _ZEROPAGE. Ce segment est défini par le compilateur qui y stocke ses variables temporaires. On peut y ajouter ses propres variables.

```
#pragma DATA_SEG _ZEROPAGE
    int FastVariable ;
```

Les segments sont définis dans un fichier de paramétrage utilisé par le link et d'extension prm.

On peut revenir à tout moment au segment par défaut des variables par :

```
#pragma DATA_SEG  DEFAULT
```

3.2.3. Les constantes

En C la notion de constante n'existe pas.
On peut appliquer le modificateur **const** devant une variable

```
Const int SIZE = 100 ;
```

Ce modificateur génère un message d'erreur lors d'une tentative d'écriture sur cette constante.

```
SIZE = 200 ;  // un message d'erreur est généré
```

Inconvénient de cette méthode : consomme de la mémoire en RAM.
Le compilateur propose l'option –Cc qui permet de stocker des constantes dans un segment particulier en ROM nommé ROM_VAR, sans réserver l'espace équivalent en RAM.

Exemple :
const char HEXADECIMAL[] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'} ;
l'attribut const associé à l'option de compilation –Cc permet de ne pas encombrer la RAM avec ses 16 valeurs.

mot clé : const

```
const char table[]={…} ;
const char *table[]={…} ;
```

quand une constante doit être stockée dans une zone spécifique de la ROM, on doit utiliser la pragma **CONST_SEG**

```
#pragma CONST_SEG   MY_ROM   //placement déclaré dans le fichier de LINK
const char table[]={…} ;
```

3.2.4. Stockage en mémoire EEPROM

Permet de conserver certains paramètres lorsque le composant est mis hors tension.
Les données sont accessibles en lecture/écriture
Afin que ces données soient stockées aux addresses de L'EEPROM, un segment particulmier est créé par l'intermédiaire d'une pragma :

```
#pragma DATA_SEG  MyEEPROM
```

3.2.5. Allocation de bouts de code

```
#pragma        CODE_SEG  MY_ROM
void FctName1(void){…..}
…
#pragma        CODE_SEG   DEFAULT
void FctName2(void){…..}
```

3.2.6. Accés à la mémoire via des pointeurs

en mode LARGE (-Mi ou –Mx option)
pour accéder à une donnée en page 0 via un pointeur utiliser préférentiellement le mot clé **near**

```
char * near FunctionName (char * near ptr)
{return ( (char * near) ptr + 1 ;}
```

en mode SMALL (-Ms option)
pour accéder à une donnée en dehors de la page 0 via un pointeur utiliser préférentiellement le mot clé **far**

3.3. Les registres des Périphériques

Les registres des périphériques sont des variables localisées dans la mémoire de données en page zéro. A l'inverse des variables utilisateur ces registres sont situés à des adresses fixes prédéfinies par construction dans le µcontrôleur.

3.3.1. Déclaration des registres

Considérons le registre PADR est à l'adresse absolue 008 de la mémoire

Première méthode :

Cette méthode crée **un pointeur absolu sur le registre** au moyen d'une **macro**

```
#define PADR * ((unsigned char *)(8))
PADR = 0x10 ;               // écrire 10h (00010000) sur le port A
PADR &=  ~0x10 ;            // mise à zéro du bit 4 du port A
```

Seconde méthode :

Elle consiste à attribuer un **segment** pour chaque périphérique à l'intérieur duquel les registres sont définis comme une suite de variables du type *unsigned char*. Cette suite est ordonnée selon la position physique des registres dans la *memory map* de la RAM du composant.

Exemple du périphérique SPI :

```
// Serial Peripheral Interface
#pragma DATA_SEG SHORT SPI
extern   volatile unsigned char SPIDR;       // SPI Data Register
extern   volatile unsigned char SPICR;       // SPI Control Register
extern   volatile unsigned char SPISR;       // SPI Status Register
```

tous les registres d'un modèle de la famille ST7 seront regroupés dans un fichier source : ici **map72311.c**

Le segment DATA_SEG SHORT SPI est défini entre les adresses 21h et 23h au moment du link par le fichier de paramétrage utilisé : ***.prm (Linker Parameter File)**

```
// MEMORY LOCATION SETTING
SECTIONS
// some declarations…
ASPI   = READ_WRITE      0x21 TO 0x23 ;
// more declarations…

PLACEMENT
// some declarations…
SPI      INTO ASPI;
// more declarations…
```

ATTENTION : l'édition de lien optimise le code en éliminant, dans chacun des segments les registres non utilisés par le programme et remplit les addresses laissées vides en décalant les adresses des registres suivants.
Pour l'interdire d'optimisation on rajoute le signe + dans la partie NAMES du fichier de paramétrage.7

NAMES
    prog1.o
    map72311.o+
    start07.o
    ansi.lib
    END

3.3.2. Lecture, écriture et test d'un bit dans un registre

Test d'un bit dans un registre

prenons par exemple le bit 7 du registre TASR. Ce bit à été défini dans st72251.h par la ligne :
#define ICF1 0x07 //Input Capture Flag 1 bit
pour tester ce bit on utilise :
if (TASR & (1 << ICF1)) // test si le bit ICF1 (bit 7) est à 1

(1 << ICF1) = (1 décalé à gauche 7 fois à gauche) (10000000)=0x80
le & avec TASR permet de masquer tous les autres bit de TASR

Mise à 1 ou 0 d'un bit dans un registre

#define BIT4 0x04

SPIDR | = (1 << BIT4) ; //mise à 1 du bit 4 de SPIDR

(1 << BIT4) = (1 décalé à gauche 4 fois à gauche) (00010000)=0x10
réalise un OU logique entre SPIDR et 0x10
rappel : 1 | 0 = 1, 0 | 0 =0, 1 | 1 =1

SPIDR & = ~(1 << BIT4) ; //mise à 0 du bit 4 de SPIDR

réalise un ET logique entre SPIDR et le complément de 0x10 cad (11101111)
rappel : 1 & 0 = 0, 0 & 0 =0, 1 & 1 =1

3.3.3. Configuration des registres lors d'initialisation de périphériques

objectif : syntaxe la plus explicite en cas de relecture ou de modification du programme

dans le code source à l'initialisation des périphériques

```
TACR1 =     ( 1 << ICIE   ) |  // interruption sur capture validéé
            ( 0 << OCIE   ) |  // interruption sur comparaison
            ( 0 << TOIE   ) |  // interruption sur débordement
            ( 0 << OLVL2) |  // Démarrage cycle avec sortie niveau bas
            ( 0 << IEDG1 ) |  // Sens de la transition active
            ( 0 << OLVL1) |  // Fin cycle avec sortie niveau haut
```

dans map72311.h

```
// Timers A&B Control Register 1 bit definition

#define ICIE        0x07   // Input capture interrupt enable
#define OCIE        0x06   // Output compare interrupt enable
#define TOIE        0x05   // Timer overflow interrupt enable
#define OLVL2       0x02   // Output level 2
#define IEDG1       0x01   // Input edge 1
#define OLVL1       0x00   // Ouput level 1
```

3.3.4. Utilisation de macros pour les opérations sur les bit

Exercice

SetBit, ClrBit set/clear a selected bit in a byte variable.
ChgBit invert the selected bit value (0->1 or 1->0).
AffBit set with a chosen value a selected bit in a byte variable.
MskBit select some bits in a byte variable and copy them in another byte variable.
The "MskBit" command allows to select some bits in a source
variables and copy it in a destination var (return the value).

ValBit return the logic value of a selected bit in a byte variable.

BitSet, BitClr set/clear a defined bit in the hardware register map
BitVal return the logic value of a defined bit.

Comments :
- VAR : Name of the character variable where the bit is located.
- Place : Bit position in the variable (7 6 5 4 3 2 1 0)
- Value : Can be 0 (reset bit) or not 0 (set bit)

The "ValBit" command returns the value of a bit in a char
variable: the bit is reseted if it returns 0 else the bit is set.

Manipuler les bits à partir des variables
This method generates not an optimised code yet.

```
#define SetBit(VAR,Place)        ( VAR |= (1<<Place) )
#define ClrBit(VAR,Place)        ( VAR &= ((1<<Place)^255) )
```

```
#define ChgBit(VAR,Place)         ( VAR ^= (1<<Place) )
#define AffBit(VAR,Place,Value)   ((Value) ? \
                                  (VAR |= (1<<Place)) : \
                                  (VAR &= ((1<<Place)^255)))
#define MskBit(Dest,Msk,Src)      ( Dest = (Msk & Src) | ((~Msk) & Dest) )
#define ValBit(VAR,Place)         (VAR & (1<<Place))
```

Manipuler les bits à partir des adresses
This method generates the most optimized code.

```
#define AREA 0x00     // The area of bits begins at address 0x10

#define BitClr(BIT)  ( *((unsigned char *) (AREA+BIT/8)) &= (~(1<<(7-BIT%8))) )
#define BitSet(BIT)  ( *((unsigned char *) (AREA+BIT/8)) |=   (1<<(7-BIT%8))  )
#define BitVal(BIT)  ( *((unsigned char *) (AREA+BIT/8)) & (1<<(7-BIT%8)) )
```

3.4. Programmation des Interruptions

des pragma spécifiques pour la gestion des interruptions du ST7 ont été rajoutées.

les routines d'interruptions

La pragma **TRAP_PROC** devant la définition d'une fonction produit l'instruction IRET à la place de RET à la fin de la fonction

```
#pragma TRAP_PROC
void InterruptFct (void) {…}
```

Note : pour sécuriser les vecteurs d'interruption inutilisés il vaut mieux les diriger vers un bout de code vide (dummy_rt)

maintenant on doit associer la fonction d'interruption au vecteur d'interruption associé à l'interruption matériel :
dans le fichier d'extension .prm

```
VECTOR ADDRESS 0xFFE0 dummy_rt
VECTOR ADDRESS 0xFFE2 InterruptFct
…..
```

sauvegarde du contexte

par défaut le compilateur utilise 3 variables de stockage pour des opérations étendues. Ces variables sont _SEX (stockage indirect), _LEX (pour chargement indirect) et *RZ* (pour stockage temporaire) et sont stockées en page 0.

Ces variables peuvent être utilisées à n'importe quel moment et ne sont pas contrôlées par le programmeur,

dans un premier temps on doit avant l'utilisation d'une routine sauvegarder le contexte de ces registres logiciel.

On utilise la pragma **SAVE_REGS**

```
#pragma TRAP_PROC  SAVE_REGS
void InterruptFct (void) {…}
```

dans un second temps après avoir vérifié si le code généré de la routine d'interruption utilise ces variables, on peut supprimer ou non la pragma SAVE_REGS

Note : l'utilisation du SAVE_REGS rajoute 34 bytes au code et augmente le temps de latence de la routine d'interruption de 34cycles et la durée totale de la routine d'interruption de 79 cycles (10microsec avec fcpu=8MHz)

3.5. Langage C optimisé pour microcontrôleurs

Initialement créer pour les systèmes UNIX, maintenant largement utilisé dans les systèmes embarqués.
Le choix d'utiliser le C dans les applis micro :

- Langage structuré basé sur les types de données et des structure de contrôle évoluée.
- Ca portabilité sur différents cibles micro

L'objet de ce chapître est de montrer comment ecire un C optimisé en taille de code et de donnée. Nous verrons des instructions spécifiques rajoutées en se sens comme des instructions de compilation ou des pragma.

stratégie des variables locales et paramètres

normalement les variables locales et les paramètres sont stockés dans la pile. Pour générer un code plus optimisé, le compilateur Hiware utilise une zone ram dédiée appelée OVERLAP (chevauchement) les instructions ST7 ne permettent pas un accès simple aux données de la pile (LD A,[SP,#n]). La solution requiert deux instructions assembleur (LD X,S puis LD A,[#0x10100+n,X] avec 0x0100 bas de la pile et n l'offset de position de la donnée)

passage de paramètres et fonction

char FunctionName	(char prmN, … ,	char prm1,	char prm0)
registre A	**OVERLAP**	**registre X**	**register A**

si A et X sont utilisés les 2 dernières valeurs sont stockées dans l'OVERLAP

cela permet en fait certaine optimisation :

```
char AddChar (char p1, char p2)
{
        return(p1+p2) ;
}
```

seul p1 est stocké dans l'OVERLAP

en assembleur on a :

```
LD    p1,X
ADD   A,p1
RET
```

Objectif : lisibilité, portabilité et robustesse (nécessaire en programmation Microcontrôleur)

Astuces et écriture optimisée du C

Quand la cible de la programmation est un microcontrôleur, il faut en permanence garder à l'esprit que l'espace mémoire est limité et que la rapidité d'exécution du programme n'est pas infinie.

On va éviter la programmation approximative pas trop gênante pour un pc.
Qques règles pour fournir un code économique :

Création d'une fonction lorsqu'un groupe d'instructions est répété plusieurs fois

exemple
for (p1 = string1, p2 = string2 ; (*p2 = *p1) ; p1++ ,p2++) ;
une seule ligne d'écriture pour une tache puissante.
cette ligne de code copie une chaîne de caractère dans une autre, en recopiant chaque octet d'un emplacement mémoire vers un autre.
Si l'on doit répéter cette ligne plusieurs fois dans le programme il est avantageux d'en faire une fonction

Remplacement des multiplications ou divisions par des décalages

Les multiplications ou les divisions sont des gros consommateurs de temps machine.
Quand le multiplicateur ou le diviseur a pour valeur une puissance de 2 , il faux réaliser un decalage.

Exemple :
A = B / 16 ;
Sera remplacé par :
A = B >> 4 ;
Dans ce cas le compilateur créé un code optimisé de manipulation des bits.

Limitation de la taille des variables à leur minimum

(unsigned) char	8 bits	0	255
signed char		-128	127
unisgned int (short)	16 bits	0	65535
signed int (short)		-32768	32767
unsigned long	32 bits	0	4294967295
signed long		-2147483648	2147483647
float, double	IEEE32	1.17…E-38F	3.40…E+38F

Pas de long int quand les int suffisent
Surtout pas de float !
(les calculs en float nécessitent des routines occupant un important espace mémoire)
(de plus certaines difficultés dans les tests d'égalité.)

Attention à l'utilisation de long ou de float dans les routines d'interruptions !

Une architecture 8 bit est par construction plus efficace pour une manipulation de données de type *char*.

Limiter les calculs mathématiques (trigo, log, ..)

Prévoir des tables de valeurs pré calculées, par exemple stockées en EEPROM

Attention aux opérations inutiles !
Par exemple :
$\Sigma(Ai/b) = (\Sigma Ai)/b$

possibilité de réutilisation des variables du code…..

stratégie d'affectation des noms

MOD_FILE.c

#define MOD_MACRO (MAJUSCULE)

char Mod_**c**Var ;

int MOD_**i**Fct(void)
{…}

Macro ou fonction générique

préférer les fns génériques aux macros. Sauf si macro très courtes et optimisées

Note : tous les bloques d'instructions répétés doivent être remplacés par une fonction si possible.

Variables Statiques

Doivent être déclarées en variable statiques :

- les variables globales utilisées dans le module ou elles sont déclarées
- les fonctions utilsées uniquement dans le module où elles sont déclarées
- une variable permanente locale dans une fonction

```
static char module_var ;
static void module_fct  (void)
{…}
void exported_fct (void)
{
        char tmp_local_var ;
        static char perm_locxal_var ;
}
```

variables volatilles

le qualificateur volatile doit être ajouté devant chaque déclaration de registre hardware pouvant être modifié par le matériel. Cela permet au compilateur de désactiver l'optimisation de la mémoire.

L'Assembleur en ligne : HLI

pour certaines instruction très spécifique au ST7, le langage C peut ne pas être approprié. Pour cela le compilateur permet d'insérer des lignes de code HLI (high level inline assembleur) par exemple pour gérer de timings spécifiques….

```
asm  LD A,X ;                    1 seule instruction
asm
{
LD A,X ;
LD Y,A ;              instructions en block
}
```

tous les type de données du programme en C sont accessible dans avec l'assembleur en ligne.

```
        #define CST 10
        struct {char field1 ; caher field2 ;} str ;
        char var, array[10] ;
..
        asm
        {
                LD A, 0x0xA100        acces direct à la mémoire     LD A, 0xA100
                CP A, #CST                                          CP A, #0x0A

                LD A, str.field2                                    LD A,str :0x1
                LD A, array[4]                                      LD A,array :0x07

                LD A,#var             acces aux variables           LD A,#var
                LD A,#LOW(VAR)        par addresse                  LD A,#LOW(VAR)
                LD A,#HIGH(VAR)                                     LD A,#HIGH(VAR)
                CALL fct                                            CALL fct
        }
```

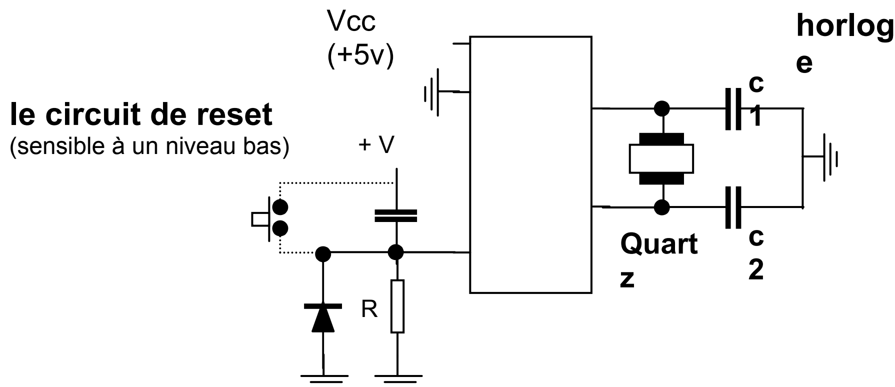
Note : le caractère # est utilisation pour concaténer des chaînes de caractères, et comme les instructions du ST7 utilise # poyr les valeures immédiates, le compilateur HIWARE C permet d'utilser la pragma **NO_STRING_CONSTR** qui désactive les ordre au preprocesseur.

Exemple :

```
#pragma NO_STRING_CONSTR
#define LDA10LDA,#10
asm LDA10;
```

IV. La Mise en œuvre

4.1. Mise en œuvre matérielle



Alimentation et horloge

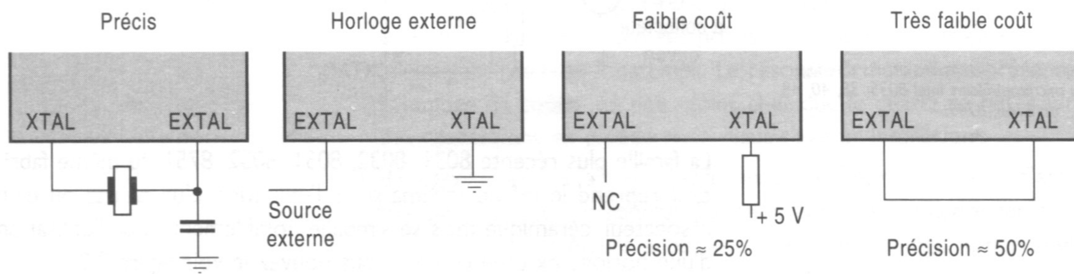
Le microcontrôleur doit être choisi en fonction de la façon dont est alimentée l'application (secteur ou pile) et de la tension d'alimentation des autres éléments du circuit. Dans la plupart des cas, les microcontrôleur utilisant la technologie CMOS et HCMOS sont les plus adaptés.
Sur tous les microcontrôleurs actuels, le générateur d'horloge est intégré. Il ne reste plus qu'à placer un composant externe pour en fixer la fréquence d'utilisation. Il faut donc se reporter à la documentation technique du constructeur pour connaître la plage d'utilisation du circuit.

Pour certains circuits, la consommation est fortement liée la fréquence d'utilisation. Ainsi, pour des raisons d'autonomie, Il est plus judicieux de réduire la fréquence d'horloge des applications qui n'ont pas besoin d'une fréquence élevée.

Selon les fabricants et les familles, l'horloge interne est configurée à partir d'un quartz, d'un résonateur céramique ou d'un générateur de signaux carrés.

Le **quartz** offre une base de temps très stable. Il est utilisé lorsqu'on a besoin d'une référence de temps très précise.

Le **résonateur céramique** est moins précis que le quartz mais coûte moins cher. Il permet cependant de rythmer précisément des opérations sur de courtes périodes.
Le générateur de signaux carrés externe est utilisé assez rarement. On le rencontre lorsqu'on a besoin de faire fonctionner plusieurs microcontrôleurs.



Il est même possible de piloter l'oscillateur d'horloge avec une résistance. C'est le cas de certains circuits de la famille 6805 de Motorola ou PIC de Microchip. La stabilité de la fréquence n'est alors que de +/- 20 %, mais est suffisante pour un grand nombre d'applications et permet de faire des économies de composants et de place.

fig.2 : *Les différentes configurations d'horloge*

4.2. Mise en œuvre logicielle

Après avoir choisi le microcontrôleur, il faut s'attaquer à la phase de programmation. Nous allons maintenant étudier le système de développement.

Assembleur et compilateur

Un système de développement comporte en premier lieu un assembleur et un ou plusieurs compilateurs adaptés au langage évolué que l'on souhaite utiliser pour programmer.

> **L'assembleur** traduit les instructions écrites en utilisant les mnémoniques du langage machine en code binaire exécutable par le microcontrôleur. L'enchaînement des mnémoniques s'appelle le listing ou code source du programme alors que le code binaire s'appelle l'objet ou l'exécutable.

Le compilateur, quant à lui, traduit les instructions écrites en langage évolué en code binaire exécutable. L'ensemble de ces instructions constitue aussi un listing ou code source et le code binaire est appelé code objet.

Les systèmes de développement pour microcontrôleur s'apparentent ainsi à ceux qui existent sur les ordinateurs. Assembleur et compilateur doivent cohabiter sans heurt. Cela est d'autant plus vrai pour les microcontrôleurs que les entrées/sorties performantes et rapides se conçoivent en assembleur. Un compilateur bien interface avec le langage machine permet de pouvoir écrire des sous programmes directement en langage machine au sein du programme en langage évolué.

Une question vient à l'esprit : pourquoi un langage pour en retranscrire un autre ? En fait, l'assembleur est pénible à utiliser lorsqu'il s'agit de faire des calculs complexes car il est constitué d'instructions élémentaires. On préfère alors utiliser un langage évolué. Par contre, l'assembleur, qui ne génère pas de lignes supplémentaires dans le code, convient parfaitement lorsqu'il s'agit de manipuler les données des registres ou de commander les entrées/sorties.
On fait appel à des machines dites "hôtes" pour faire fonctionner assembleur et compilateur. Ces machines hôtes, qui peuvent être de simples PC, assemblent ou compilent le programme de l'application pour le transcrire en binaire exécutable.
Il faut toutefois garder en mémoire qu'un microcontrôleur n'apportera jamais la puissance de calcul d'un super calculateur. Du fait que les mémoires des microcontrôleurs sont de petite taille (de 2 à 32 Ko), le langage évolué ne doit pas générer un grand nombre de ligne de code machine par ligne de langage.

Emulateur et simulateur

Avant de mette en œuvre le programme, il est plus prudent de le tester. C'est le rôle des émulateurs et simulateurs.

> **L'émulateur** est un montage éclaté du microcontrôleur qu'il remplace. Toutes les fonctions y sont séparées et reproduisent son comportement. Il est branché directement sur la maquette de l'application. Cette version éclatée permet d'avoir directement accès aux divers signaux internes. La mise au point du programme est d'autant plus facilitée que l'émulateur fonctionne aussi vite que le microcontrôleur qu'il remplace. L'inconvénient de l'émulateur est son coût relativement élevé.

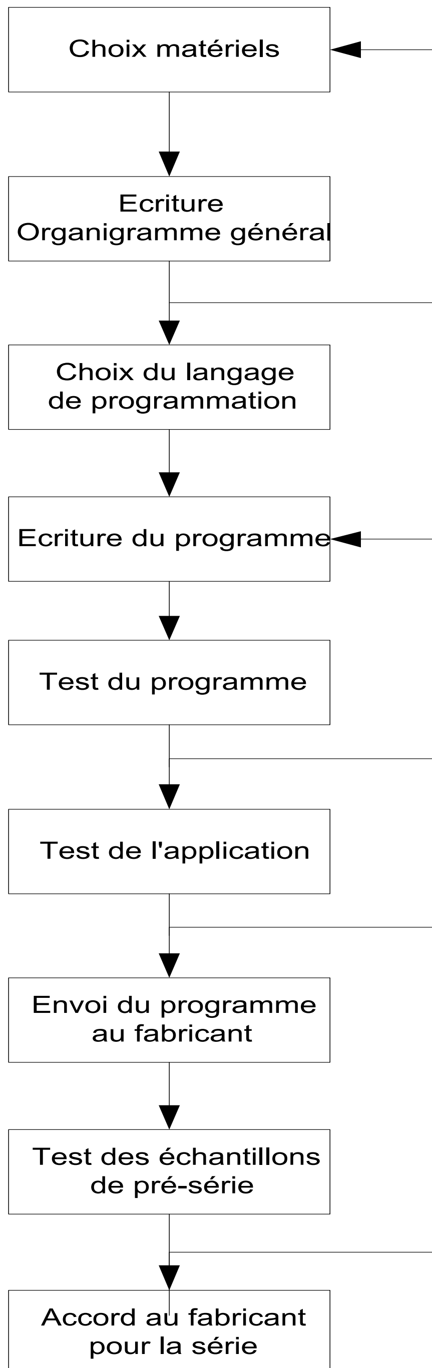
Dans certains cas, un **simulateur** suffit à tester la plupart des fonctionnalités d'un programme. L'émulateur est en fait un programme sur une machine qui reproduit le fonctionnement de l'unité centrale. Les entrés/sorties y sont représentées par des octets qu'on présente ou qu'on lit. On ne pourra pas non plus tester l'application en temps réel.

Un simulateur n'apporte donc pas toutes les possibilités qu'un émulateur, mais élimine un grand nombre d'erreurs et suffit pour des applications sans entrées/sorties particulières et sans notion de temps.

Une fois la mise au point du programme achevée, on peut faire les derniers tests selon deux cas. Si on désire utiliser des versions OTP, on programmera des EEPROM ou UVPROM équivalentes. Cela permet d’opérer d'ultimes modifications. Si par contre, on désire utiliser des versions programmées par masque, il faudra envoyer le programme au fabricant qui produira en retour une pré-série et attendra l'accord pour commencer la production en série.

Le développement d'une application

Au niveau industriel, la conception d'une application se déroule en plusieurs étapes. Après avoir choisi la solution matérielle au problème, on peut commencer à penser à un organigramme pour le programme. Puis, viennent le choix du langage et la programmation en elle-même. Par émulation ou simulation, on test ensuite le programme et l'application. On y apporte les correctifs nécessaires au bon fonctionnement avant d'envoyer le programme au fabricant.



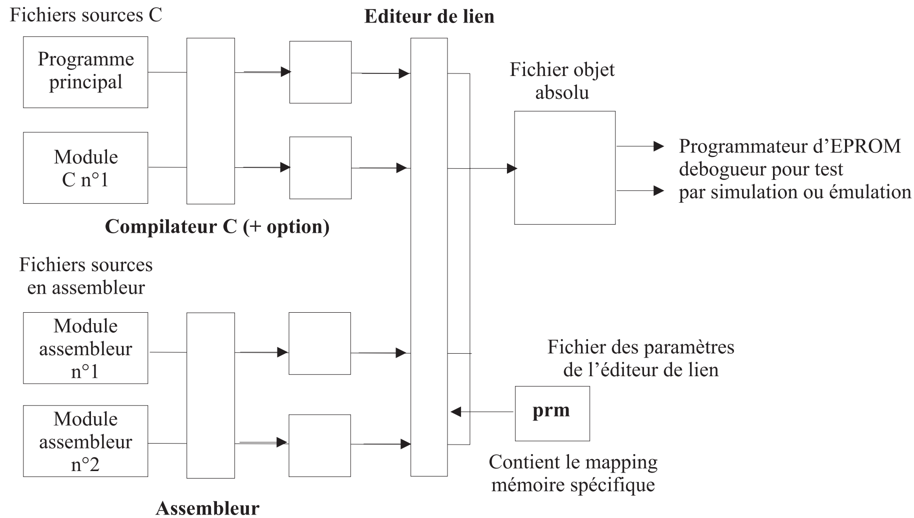
Organigramme de développement

Dans le cas où on ne disposerait que de moyens réduits ou que la quantité de pièce est jugée insuffisante pour justifier des crédits importants, on peut toujours choisir des microcontrôleurs qui possèdent une version EEPROM ou UVPROM. On fera alors des tests directement sur l'application. A l'aide des quelques lignes d'entrées/sorties, de LEDS, on peut ajouter temporairement des instructions inutiles à l'application mais qui servent à suivre l'évolution du programme et à en vérifier le bon fonctionnement.

Les constructeurs commercialisent parfois des kits de développement à des prix intéressants pour faire connaître leurs produits. Ces kits comprennent un assembleur, un simulateur, une carte de programmation et de la documentation sur le microcontrôleur.

4.3. La chaîne de développement ST7

L'éditeur de lien HIWARE



l'éditeur de lien permet une gestion complète de l'organisation de la mémoire du microcontrôleur cible. Cette gestion est contenue dans un fichier d'extension **prm**. Ce fichier doit contenir les informations suivantes :

- Le nom du fichier absolu d'extension **abs** résultant de l'édition de lien

  LINK prog1.abs

- La liste des fichiers objets devant être liés

```
NAMES
        prog1.o
        map72311.o+
        start07.o       // Tout programme en C Hiware doit être lié au fichier start07.o
        ansi.lib
END
```

- La liste des sections selon lesquelles la mémoire est organisée :
  Une section est un bloc d'adresses continues caractérisé par un nom et par ses propriétés : READ_ONLY, READ_WRITE, PAGED, NO_INIT.
  (No_INIT : permet de ne pas effacer le contenu des variables d'une section, lors du démarrage ou redémarrage du programme)
  utilité :
  Chacune de ces sections contient des parties de code objet (données ou programme)
  Les sections sont organisées en segments

```
SECTIONS
        APORTS      =     NO_INIT           0x00 TO 0x17;
        ASPI        =     READ_WRITE        0x21 TO 0x23;
```

```
    ATIMERA     =     READ_WRITE     0x31 TO 0x3F;
END
```

- La liste des segments (Placement)
  Un segment est une suite continue d'octets représentant un bloc d'instruction ou de données. Ce bloc est positionné dans la mémoire en l'attribuant à une section.

```
PLACEMENT
    PORTS                         INTO  APORTS;
    TIMERA                        INTO  ATIMERA;
    SPI                           INTO  ASPI;
END
```

  Exemple le segment SPI contient 3 octets (unsigned char) :
  SPIDR (SPI Data Register)
  SPICR (SPI Control Register)
  SPISR (SPI Status Register)

  (si un segment est de taille supérieure à la section qui lui est attribuée, l'édition de lien échouerait et le fichier .abs ne serait pas généré)

- La taille de la pile

```
STACKSIZE = 0x40 ;
```

- La liste des vecteurs d'interruption et de démarrage. Les vecteurs sont caractérisés par un nom de fonction auquel le programme d'interruption doit se brancher. Pour chaque type d'interruption la localisation mémoire est reportée.

```
VECTOR ADDRESS 0xFFEA TimerAInterupt
VECTOR ADDRESS 0xFFEC spi_rt
VECTOR ADDRESS 0xFFFE _Startup
```

! l'éditeur de lien produit, en plus du fichier .abs, un fichier .map qui résume les adresses des segments, des variables, ainsi que des informations utiles pour le débogage.

4.4. Exemple de projet

main.c : Programme principal en langage C et assembleur en ligne (HLI)

enviro.prm : fichier de link qui sert à définir les zone mémoires des périphériques et des variables du programme en fonction du maping mémoire du microcontrôleur. Définition également des vecteurs d'interruption et des nom des routines d'interruption que l'on retrouve dans le programme principal.

defaut.env : fichier d'environnement qui contient les options de compilations ainsi que les différents path du projet. (Chemin du compilateur et du linker)

enviro.mak : fichier make

4.4.1. Programme principal "main.c"

```
#include <hidef.h>
#include "map72334.h"           // ST72334 memory and registers mapping
#include "lib_bits.h"           // Predifined libraries working at bit level

#define DisableInterrupts {asm SIM;}
#define EnableInterrupts  {asm RIM;}

#pragma DATA_SEG _ZEROPAGE
unsigned long val1,val2,val3,i;

 void init(void);          // routine d'initialisation des varibles et I/O
void init_timer(void);          // routine d'initialisation de l'horloge


void main(void)
{
        DisableInterrupts;
        MISCR1=(0<<SMS)|        // mode normal fCpu=fosc/2=8Mhz
           (1<<CP0)|            // inoperant pour SMS=0
           (1<<CP1)|            // inoperant pour SMS=0
           (0<<IS20)|
           (0<<IS21)|
           (0<<MC0)|            // pas de clock ext
           (0<<IS10)|           // pour que l'interruption ei3 sur le port B soit
           (1<<IS11);           // en front descendant

        init();
        init_timer();

        EnableInterrupts;

        while(1)
        {
        }
}

void init(void)
{
//entree
```

```
        ClrBit(PBDDR,3);        //pb3 (interrupteur) en entrée interruption
        SetBit(PBOR,3);
//sortie
        SetBit(PBDDR,2);        //config LED
        SetBit(PBOR,2);
}

void init_timer(void)
{
        TACR1=(0<<ICIE)|
            (1<<OCIE)|          //output compare
            (0<<TOIE)|
            (0<<FOLV2)|
            (0<<FOLV1)|
            (0<<OLVL2)|
            (0<<IEDG1)|
            (0<<OLVL1);

        TACR2=(0<<OC1E)|
            (0<<OC2E)|
            (0<<OPM)|
            (0<<PWM)|
            (1<<CC1)|           //ftimer=fcpu/8
            (0<<CC0)|               //ftimer=1Mhz
            (0<<IEDG2)|
            (0<<EXEDG);

        TAOC1HR=0xF4;           //registres haut pour base de temps ?
        TAOC1HR=0x0F;           //registres bas pour base de temps ?
}

#pragma TRAP_PROC SAVE_REGS
void eit3_rt(void)
{
        TAOC1HR=0x1E;           //registres haut pour base de temps ?
        TAOC1LR=0x84;           //registres bas pour base de temps ?
}

#pragma TRAP_PROC SAVE_REGS
void tima_rt(void)
{
        if (ValBit(TASR,TOF))
        {
                        ChgBit(PBDR,2);
                        asm
                        {
                                ld a,TASR
                                ld a,TAOC1LR
                }
                //mise à 0 du timer perpétuel TACR
                TACLR=0;
        }
}
```

4.4.2. fichier de link “enviro.prm”

```
LINK led_i.abs

NAMES  main.o map72334.o+  start07.o  ansi.lib END

STACKTOP 0x1FF

SECTIONS
   ZRAM = READ_WRITE 0x0080 TO 0x00FF;
   RAM  = READ_WRITE 0x01FF TO 0x0200;  /* pour le J4 */
   ROM  = READ_ONLY  0xE000 TO 0xFFFF;


PLACEMENT
   PORT        INTO NO_INIT  0X0000 TO 0X0016;
   MISC1       INTO NO_INIT  0X0020 TO 0X0020;
   MISC2       INTO NO_INIT  0X0040 TO 0X0040;
   SPI         INTO NO_INIT  0X0021 TO 0X0023;
   ITC         INTO NO_INIT  0X0024 TO 0X0027;
   MCC         INTO NO_INIT  0X0029 TO 0X0029;
   WDG INTO NO_INIT  0X002A TO 0X002B;
   EEPROM      INTO NO_INIT  0X002C TO 0X002C;
   Timer_A     INTO NO_INIT  0X0031 TO 0X003F;
   Timer_B     INTO NO_INIT  0X0041 TO 0X004F;
   SCI         INTO NO_INIT  0X0050 TO 0X0057;
   ADC         INTO NO_INIT  0X0070 TO 0X0071;
   DEFAULT_ROM, ROM_VAR, STRINGS       INTO  ROM;
   DEFAULT_RAM                          INTO  RAM;
   _ZEROPAGE, _OVERLAP                  INTO  ZRAM;

END


VECTOR ADDRESS 0XFFEA tima_rt
VECTOR ADDRESS 0XFFF0 eit3_rt

/*
VECTOR ADDRESS 0XFFF2 eit2_rt
VECTOR ADDRESS 0XFFF6 eit0_rt
VECTOR ADDRESS 0XFFF0 eit3_rt
VECTOR ADDRESS 0XFFE0 dummy_rt
VECTOR ADDRESS 0XFFE2 dummy_rt
VECTOR ADDRESS 0XFFE4 eeprom_rt
VECTOR ADDRESS 0XFFE6 sci_rt
VECTOR ADDRESS 0XFFE8 timb_rt
VECTOR ADDRESS 0XFFEA tima_rt
VECTOR ADDRESS 0XFFEC spi_rt
VECTOR ADDRESS 0XFFEE dummy_rt
VECTOR ADDRESS 0XFFF0 eit3_rt
VECTOR ADDRESS 0XFFF2 eit2_rt
VECTOR ADDRESS 0XFFF4 eit1_rt
VECTOR ADDRESS 0XFFF6 eit0_rt
VECTOR ADDRESS 0XFFF8 mcc_rt
VECTOR ADDRESS 0XFFFA dummy_rt
*/

VECTOR ADDRESS 0XFFFE _Startup
```

4.4.3. fichier d'environnement "defaut.env"

```
LIBPATH=C:\HIWARE\LIB\ST7C\include
GENPATH=*C:\HIWARE\LIB\ST7C;*C:\Projets\Active vision\Asservissement\ST7-232\led334
OBJPATH=C:\Projets\Active vision\Asservissement\ST7-232\led334
ABSPATH=C:\Projets\Active vision\Asservissement\ST7-232\led334
TEXTPATH=C:\Projets\Active vision\Asservissement\ST7-232\led334

COMP=C:\HIWARE\PROG\CST7.EXE
LINK=C:\HIWARE\PROG\linker.EXE
FLAGS=-ml
ERRORFILE=
RESETSTACK=0x3FF
```

4.4.4. fichier make "enviro.mak"

```
#********************************************** EXECUTABLE COMMAND DEFINES ****
ENV = default.env
CC  = $(COMP)

#************************************************** OBJECT FILES DEFINES ****
OBJ_LIST = map72334.o main.o start07.o

#************************************ MAIN FILES COMPILE & LINK (LEVEL 3) ****
led_i.abs : $(ENV) $(OBJ_LIST) enviro.prm $(LINK) enviro.prm

main.o : $(ENV) lib_bits.h map72334.h hidef.h ansi.lib start07.h main.c $(CC) main.c

#************************************* LEVEL 0 FILE COMPILE (.o generate) ****
map72334.o : $(ENV) map72334.h
        $(CC) map72334.c
```