Université Joseph Fourier
Polytech' Grenoble

Filière Informatique Industrielle et Instrumentation

1ère Année

Cours de Microcontrôleurs

Basé sur l'utilisation du HCS12 (Motorola)

Christophe Durand
Année universitaire 2009-2010

Document écrit par JP Perrin.
Merci pour sa contribution.

PRÉAMBULE

Objectifs épistémologiques

Le but de cet enseignement est de vous familiariser avec le fonctionnement et l'utilisation des microcontrôleurs, qui sont devenus aujourd'hui des composants électroniques clé et incontournables pour tous systèmes automatisés. Cet enseignement est composé de cours théoriques (9 séances), de séances de travaux dirigés (10 séances) et de travaux pratiques (8 scéances). En fin d'année, la réalisation d'un « système embarqué » répartie sur 7 séances vous permettra de mettre en œuvre et d'approfondir vos connaissances à la fois en microcontrôleurs et en électronique. Ce sera aussi l'occasion de vous mettre en « *situation de projet* », situation comparable à votre futur métier d'ingénieur, où vous devrez faire preuve d'organisation, de travail en équipe, de créativité et de professionnalisme.

A l'issu de cet enseignement, vous serez capable de :

- **Connaître le fonctionnement** logiciel et matériel d'un microcontrôleur,
- **De programmer un microcontrôleur** à partir de différents niveaux de langage pour qu'il réalise une succession d'étapes logiques et complexes,
- **D'intégrer un microcontrôleur** dans des applications spécifiques.

Support et plan du cours

Les microcontrôleurs sont des composants intégrés qui contiennent dans un même boîtier un microprocesseur, de la mémoire, et des périphériques courants, tels que timer, liaison série asynchrone, liaison série synchrone, ports d'entrée sortie logiques, contrôleur de bus CAN, convertisseur analogique numérique, etc. Il en existe de nombreuses versions, qui diffèrent suivant les périphériques installés. Nous avons choisi comme support du cours le microcontrôleur Motorola MC9S12DP256B, construit autour d'un microprocesseur HCS12. L'ensemble est monté sur une carte de développement (appelée HCS12 T-Board) qui peut communiquer avec un PC de différentes façons. Motorola propose plusieurs microcontrôleurs autour du même microprocesseur comme les marques automobiles proposent plusieurs options de carrosserie autour du même moteur. Le modèle choisi est plutôt haut de gamme et comporte un grand choix de périphériques internes.

Dans une première partie, on s'intéressera principalement aux aspects logiciels du microcontrôleur HCS12 (registres, programmation séquentielle, ruptures de séquence).
Dans une seconde partie, on passera aux aspects matériels de la communication avec les périphériques internes ou non (bus d'adresse et de données, timing des échanges).
Dans une troisième partie, on étudiera certains contrôleurs de périphériques (pas tous, il y en a trop !) et on utilisera cette programmation pour piloter ces contrôleurs.

Nous travaillerons essentiellement en langage d'assemblage (assembleur), le but étant de comprendre comment s'y prend un microcontrôleur pour réaliser des instructions structurées.

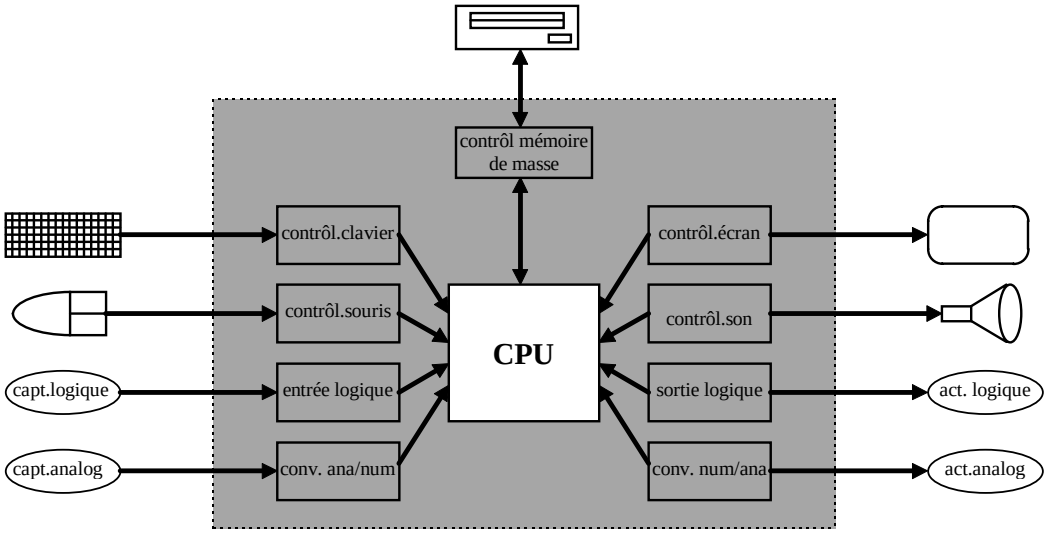
COURS n°1

Du microprocesseur au microcontrôleur

Comparaison anthropomorphique : Le cerveau s'apparente-t-il à un microprocesseur ?

Voir présentation…

Un microprocesseur muni de mémoire peut être, dans une première approche, considéré comme un « cerveau » capable d'effectuer des opérations diverses (opérations arithmétiques, logiques) et de les enchaîner dans un processus cohérent. Tel quel, un microprocesseur (on parle aussi du CPU : **C**entral **P**rocessing **U**nit) n'a aucune utilité puisqu'il ne peut communiquer avec le milieu extérieur. Pour pouvoir être efficace, un microprocesseur doit disposer d'organes d'entrée-sortie (clavier, souris, écran, haut-parleur), comme le cerveau qui dispose d'organes d'entrée (oreilles, yeux, toucher) et de sortie (paroles, gestes). Nous appelons « micro »ordinateur cette structure « micro »processeur + périphériques + liens.



Du microprocesseur au microcontrôleur…

La partie principale d'un micro-ordinateur est la « carte mère » sur laquelle sont montés le microprocesseur, les contrôleurs de périphériques et les lignes qui les connectent entre eux sous forme de circuit imprimé. Avec les progrès de la miniaturisation, et la généralisation de l'utilisation des ordinateurs, les besoins les plus courants se sont standardisés et on a pu disposer toute une carte mère au sein d'une seule et même puce, appelée microcontrôleur. L'usage de microcontrôleurs est actuellement en plein développement dans toute l'informatique industrielle, et à tous les degrés de complexité (de 8 pattes à près de 200 pattes).

Un système informatique complet regroupe autour d'un microcontrôleur un ensemble de composants qui lui permettent de fonctionner (essentiellement mémoires) et de communiquer avec le milieu extérieur (contrôleurs de périphériques). Le fonctionnement du processeur consiste à exécuter un programme, c'est à dire une suite d'instructions pointées par le compteur ordinal (pointeur de programme).

Logiciellement, les composants extérieurs au processeur sont vus comme des adresses avec lesquelles il communique par des opérations d'écriture (processeur→ composant) ou de lecture (composant→ processeur). L'exécution du programme comporte les phases suivantes :

- Lire en mémoire l'octet dont l'adresse est contenue dans le compteur ordinal,
- Analyser l'octet pour trouver le nombre d'octets qui complètent l'instruction,
- Incrémenter le compteur ordinal et lire ces octets,
- Interpréter le code et exécuter l'instruction, qui peut ou non comporter une lecture et une écriture à une adresse.

Il y a donc sans cesse communication entre le processeur et les composants périphériques.

Physiquement, le processeur utilise pour cette communication :

- Le bus de données, qui véhicule l'information (sous forme de mots dans le HC12),
- Le bus d'adresses, qui permet, après décodage, d'activer le chip select approprié,
- Des lignes de contrôle qui permettent d'assurer les échanges d'information selon des protocoles temporels précis (chronogrammes) spécifiques au microcontrôleur. Les lignes principales sont la ligne $R/\overline{W}$ qui indique le sens de transfert des données, et la ligne E, horloge interne du processeur, qui cadence toutes les opérations.

La conception d'un système complet se décompose schématiquement en plusieurs phases :

- Choix de composants compatibles avec le processeur (signaux et chronogrammes). Les fabricants de processeurs proposent en général une famille de composants périphériques directement adaptables à leur processeur. Si on choisit ailleurs, il n'est pas rare d'avoir besoin d'une mise en forme des signaux pour avoir des échanges corrects.
- Choix de la carte mémoire du système. Il s'agit d'attribuer une adresse ou une zone d'adresses à chaque composant, et de réaliser les décodages d'adresse correspondants.
- Réalisation matérielle du système : implantation des divers composants, des alims, des logiques de décodage et des bus sur un support matériel (routage).
- Mise au point des liaisons matérielles pour un fonctionnement correct de la carte.

Ces opérations représentent un investissement lourd en temps et en matière grise. En outre, on constate que les systèmes se ressemblent et utilisent les mêmes types de composants :

- De la mémoire ROM et RAM au moins pour implanter le programme de gestion et tenir à jour ses variables et ses données.
- Un moyen de communication logiciel avec l'extérieur en entrée et en sortie (transfert de données, téléchargement de programmes ou de paramètres de gestion), très souvent liaison série asynchrone ou synchrone.
- Les autres périphériques varient suivant la tâche à exécuter, mais on retrouve très souvent des entrées et sorties logiques, des entrées et sorties analogiques et une gestion du temps (timer et horloge temps réel).

Il est donc naturel qu'on ait cherché à économiser du temps de mise au point en réalisant des cartes standard répondant à des demandes standard elles aussi. Il existe depuis longtemps déjà dans les catalogues une grande variété de cartes d'entrées-sorties adaptables à une grande variété de matériels avec une grande variété de destinations et de performances.

Les progrès de l'intégration des circuits permettent actuellement de réaliser toute une carte de développement sur un seul composant. Cette idée est à l'origine de tous les microcontrôleurs.

Notions de microcontrôleur

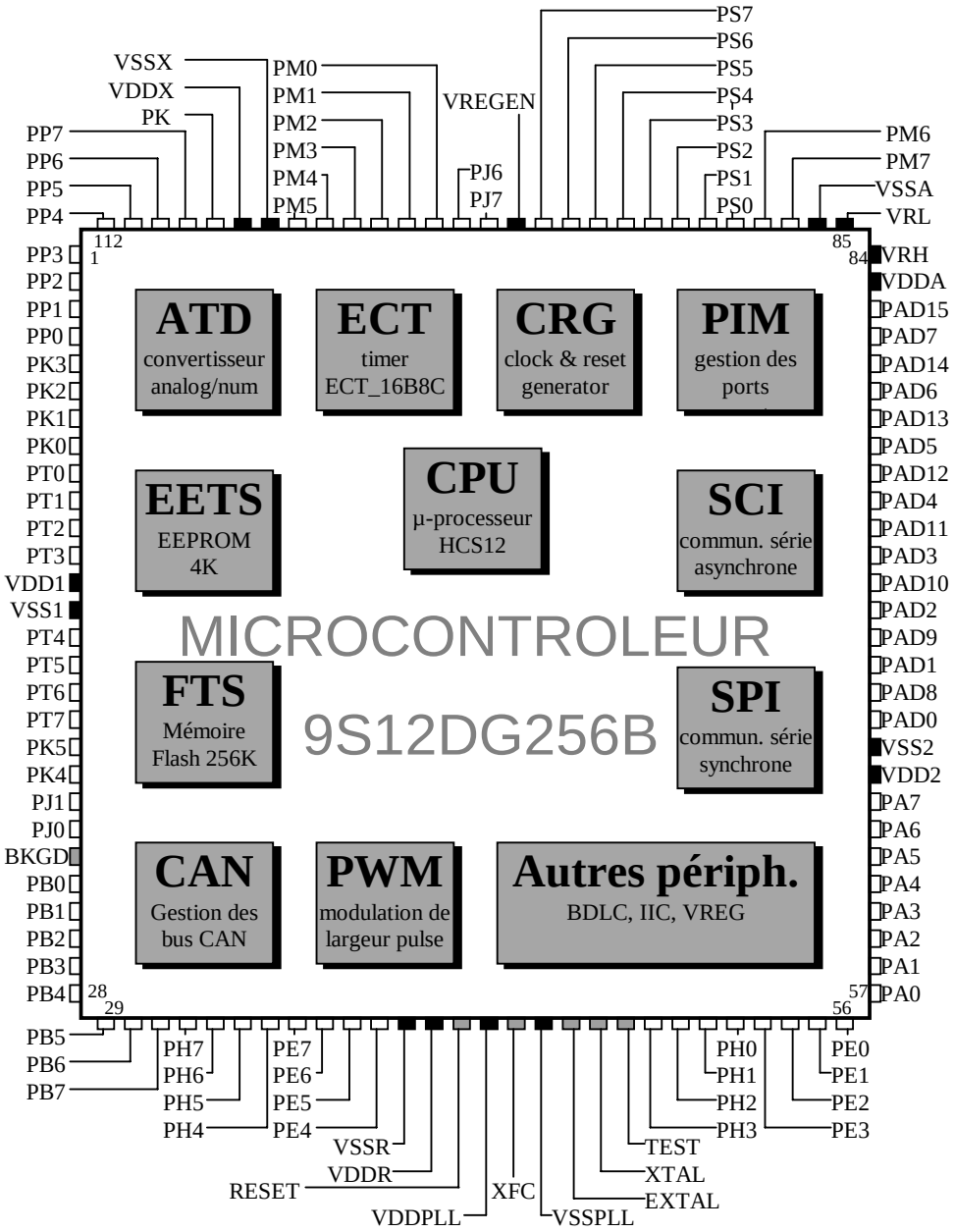
Le principe de base des microcontrôleurs repose sur l'inclusion dans le même boîtier du microprocesseur et de divers périphériques, de manière à avoir un composant autonome. Les bus de données et d'adresses sont internes au composant, ainsi que les décodages d'adresse relatifs à chaque « périphérique interne ». Les registres de contrôle des périphériques sont regroupés dans la zone d'adresse « registres du microcontrôleur ». On trouve également dans cette zone divers registres de personnalisation du microcontrôleur. Les pattes du composant se réduisent alors essentiellement à des lignes d'entrée-sortie tant logiques qu'analogiques.

L'étude d'un tel composant est beaucoup plus longue et donc beaucoup plus coûteuse que celle d'une carte. Pour être rentable, le produit doit répondre aux besoins du plus grand nombre pour un prix le plus modeste possible. Les fabricants proposent en général toute une gamme de microcontrôleurs construits à partir du même microprocesseur. De plus on peut personnaliser chaque microcontrôleur par programmation de registres internes pour encore plus de flexibilité. Toutes les fonctions ne sont pas disponibles à la fois, on aura toujours affaire à un compromis.

Les avantages d'un microcontrôleur sont les mêmes que ceux des cartes toutes faites qu'on trouve dans le commerce : une utilisation quasi immédiate, sans problème de conception, de réalisation et de mise au point. On peut y ajouter son faible encombrement, son prix modéré et une plus grande fiabilité du fait que les lignes de contrôle sont toutes internes.

Les inconvénients sont ceux du « tout compris ». On doit travailler avec les périphériques grand public choisis par le fabricant, qui ne donnent pas toujours la souplesse d'utilisation souhaitée. Les microcontrôleurs proposent en général une option qui permet un interfaçage par bus externe, comme avec les microprocesseurs. On peut ainsi rajouter au microcontrôleur un ou plusieurs composants tout en continuant à utiliser certains de ses composants internes. Bien entendu, on retrouve alors les problèmes de réalisation pratique d'une carte.

Représentation schématique du microcontrôleur 9S12DG256B



Le microcontrôleur 9S12DG256B qui équipe les kits d'évaluation de la salle de TP réunit autour d'un microprocesseur HCS12 4K d'EEPROM, 16K de RAM et 256K de mémoire Flash, ainsi que des périphériques parmi lesquels des ports d'entrée-sortie logiques (PIM), un timer (ICT), deux convertisseurs analogiques numériques 10 bits à 8 voies (ATD0 et ATD1), deux interfaces série asynchrones (SCI0 et SCI1), trois interfaces série synchrones (SPI0, SPI1 et SPI2). Il contient encore d'autres périphériques, qui ne seront pas abordés dans ce cours, faute de temps. Le microcontrôleur dispose également de possibilités d'extension de bus 8 ou 16 bits vers l'extérieur.

Pour communiquer avec l'extérieur, le microcontrôleur dispose en tout et pour tout de 112 pattes. C'est beaucoup trop peu pour répondre aux besoins de tous les modules périphériques, qui ne sont pas tous disponibles simultanément. L'organisation du microcontrôleur comprend

tout un arsenal de techniques pour choisir la configuration qui correspond le mieux au traitement d'un problème donné. Pour commencer, on va faire le tour du propriétaire pour savoir précisément de quoi on dispose.

Description physique des pattes

Alimentations : Les alimentations utilisent 15 pattes.
- Six d'entre elles groupées en trois couples, [*VDDR*, *VSSR*], [*VDDX*, *VSSX*], et [*VDDA*, *VSSA*] se partagent l'alimentation des modules du microcontrôleur.
- Six autres pattes [*VDD1*, *VSS1*], [*VDD2*, *VSS2*] et [*VDDPLL*, *VSSPLL*] alimentent le cœur du micro (CPU et son environnement immédiat). Ces tensions proviennent soit du régulateur interne, soit d'alimentations externes.
- La patte *VREGEN* valide ou non le régulateur interne.
- Les pattes *VRH* et *VRL* sont les références de tension du convertisseur A/N.

Sur la carte de TP, toutes les masses sont connectées, les trois alimentations *VDDR*, *VDDX* et *VDDA*, sont reliées à VDD = 5V. *VREGEN* est à 5V, ce qui valide le générateur interne. Les références du convertisseur sont reliées à VDD (pour *VRH*) et à la masse (pour *VRL*).

Le tableau ci-dessous récapitule les pattes d'alimentation

Nom patte	N° patte (/112)	Tension Nominale	Description
VDD1 *VDD2*	13, 65	2.5 V	Alimentations internes générées par le régulateur interne
VSS1 VSS2	14, 66	0 V	et masses associées
VDDR	41	5 V	Alimentation externe des drivers de pattes et du régulateur
VSSR	40	0 V	et masse associée
VDDX	107	5 V	Alimentation externe des drivers de pattes
VSSX	106	0 V	et masse associée
VDDA	83	5 V	Alimentation externe et masse pour les convertisseurs analogique numériques et référence du régulateur interne
VSSA	86	0 V	Permet de court circuiter l'alimentation des convertisseurs A/N indépendamment
VRL	85	0 V	Tension de référence haute des convertisseurs A/N.
VRH	84	5 V	Tension de référence basse des convertisseurs A/N.
VDDPLL	43	2.5 V	Fournit la tension et la masse nécessaires au PLL. Cela permet de court circuiter l'alimentation du PLL indépendamment. Alimentation et masse générées par le régulateur interne.
VSSPLL	45	0 V	
VREGEN	97	5 V	Validation/dévalidation du régulateur interne

Horloge : Trois pattes sont réservées à l'horloge
- *XTAL* et *EXTAL* : Ces lignes permettent la connexion avec un quartz pour le contrôle de l'oscillateur interne générateur de l'horloge E.
- *XFC* qui permet, en conjonction avec *VDDPLL* et *VSSPLL* d'activer le PLL pour générer une fréquence d'horloge plus élevée.

Sur la carte de TP, l'oscillateur interne fournit une horloge à 8 MHz. L'utilisation du PLL permet de travailler avec une horloge à 24 MHz. Pour plus de détails, se reporter au manuel du module CRG (Clock Reset Generator).

Trois pattes inclassables
- *RESET* ligne de la demande d'interruption CPU du même nom,
- *BKGD* ligne de gestion du debugging en arrière plan
- *TEST* qui ne sert à rien sauf aux tests en usine.

Les autre pattes, au nombre de 91, peuvent toutes servir d'entrées-sorties logiques d'usage général (ports). Elles ont presque toutes plusieurs affectations et leur dénomination varie généralement selon cette affectation, ce qui est une cause de confusion. Pour éviter cela, on s'efforcera de leur conserver leur nom de port. La plupart sont bidirectionnelles, c'est à dire qu'elles peuvent fonctionner en entrée ou en sortie. Seules les pattes *PAD0* à *PAD15* et les pattes *PE0* (=*XIRQ*) et *PE1* (=*IRQ*) ne fonctionnent qu'en entrée.

Modes de fonctionnement

Le microcontrôleur peut fonctionner selon 8 modes différents. Le choix de ce mode se fait au reset, suivant les niveaux de trois pattes à ce moment : *MODA* (autrement dit *PE5*), *MODB* (autrement dit *PE6*), et *MODC* (autrement dit *BKGD*). La sélection des modes est donc matérielle, bien qu'on puisse dans une certaine mesure la modifier par logiciel, comme le montre le tableau suivant.

MODC	MODB	MODA	Mode	Possibilité d'écrire dans MODx
0	0	0	Special Single Chip	Toujours dans MODC, B, A mais pas 110
0	0	1	Emulation Narrow	Jamais
0	1	0	Special Test	Toujours dans MODC, B, A mais pas 110
0	1	1	Emulation Wide	Jamais
1	0	0	Normal Single Chip	MODC jamais, MODB, A une fois (pas 110)
1	0	1	Normal Expanded Narrow	Jamais
1	1	0	Special Peripheral	Jamais
1	1	1	Normal Expanded Wide	Jamais

Seuls trois modes nous concernent en pratique : les modes dits normaux.

Dans le mode Single Chip, le MCU fonctionne comme un microcontrôleur pur, sans bus externe de données ou d'adresses. Ce mode permet une disponibilité maximum des lignes de périphériques (91), l'activité des bus se passant à l'intérieur du microcontrôleur. La carte de TP est prévue pour fonctionner dans ce mode.

Dans les deux modes étendus, le microcontrôleur libère des pattes pour assurer les échanges de données avec l'extérieur : ports A et B pour le bus mixte d'adresses et de données, port E pour les lignes de contrôle et port K pour gérer l'extension des adresses au delà de 64K. Les deux modes diffèrent par la taille des données échangées, le mode « narrow » utilisant un bus de données de 8 bits, tandis que le mode « wide » utilise un bus de données 16 bits.

L'organisation mémoire et les registres de contrôle

Dans un mode donné, les options de fonctionnement ainsi que les paramètres des différents modules sont essentiellement logiciels et passent par les registres de contrôle du microcontrôleur (A et B, X et Y, SP, PC et PCR). Les registres de contrôle occupent les adresses mémoires basses. La place qui leur est réservée va de 0 à $3FF, bien que certaines adresses ne soient pas occupées. Suivant le mode dans lequel on travaille on n'a pas besoin des mêmes registres et la carte mémoire des registres en est modifiée. Les implantations mémoire elles mêmes sont paramétrables par des registres, et il n'est pas question ici d'entrer dans tous les détails.

A chaque module est associé un bloc de registres contigus dans le plan mémoire. Le tableau ci-dessous indique la répartition mémoire des registres par module pour le microcontrôleur qui équipe la carte de TP.

Adresse	Module	Taille (octets)
0000-0017	**CORE** (Ports A, B, E, Modes, Inits, Test)	24
0018-0019	Réservé	2
001A-001B	**PARTID** Device ID register	2
001C-001F	**CORE** (MEMSIZ, IRQ, HPRIO)	4
0020-0027	Réservé	8
0028-002F	**CORE** (Background Debug Mode)	8
0030-0033	**CORE** (PPAGE, Port K)	4
0034-003F	**CRG** Clock and Reset Generator (PLL, RTI, COP)	12
0040-007F	**ECT** Enhanced Capture Timer 16-bit 8 channel	64
0080-009F	**ATD0** Analog to Digital Converter 10-bit 8 channel N°0	32
00A0-00C7	**PWM** Pulse Width Modulator 8-bit 8 channel	40
00C8-00CF	**SCI0** Serial Communications Interface N°0	8
00D0-OOD7	**SCI1** Serial Communications Interface N°1	8
00D8-00DF	**SPI0** Serial Peripheral Interface N°0	8
00E0-00E7	**IIC** Inter IC Bus	8
00E8-00EF	**BDLC** Byte Data Link Controller	8
00F0-00F7	**SPI1** Serial Peripheral Interface N°1	8
00F8-00FF	**SPI2** Serial Peripheral Interface N°2	8
0100-010F	**FTS** Flash Control Register	16
0110-011B	**EETS** EEPROM Control Register	12
011C-011F	Réservé	4
0120-013F	**ATD1** Analog to Digital Converter 10-bit 8 channel N°1	32
0140-017F	**CAN0** Motorola Scalable Can N°0	64
0180-01BF	**CAN1** Motorola Scalable Can N°1	64
01C0-01FF	**CAN2** Motorola Scalable Can N°2	64
0200-023F	**CAN3** Motorola Scalable Can N°3	64
0240-027F	**PIM** Port Integration Module	64
0280-02BF	**CAN4** Motorola Scalable Can N°4	64
02C0-03FF	Réservé	320

Le module central, appelé cœur (core), regroupe autour du CPU (Central Processing Unit) quelques sous modules : MEBI (Multiplexed External Bus Interface), BKP (Breakpoints), INT (Interrupt), MMC (Module Mapping Control), BDM (Background Debug Module) qui gèrent l'organisation de l'ensemble du microcontrôleur autour du processeur. Les $40 premières adresses lui sont réservées.

Les principaux modules qu'on abordera dans ce cours sont les ports d'entrées sorties logiques (PIM), le timer (ECT), le convertisseur analogique numérique (ATD) et les liaisons série asynchrone (SCI) et synchrone (SPI).

La carte mémoire du microcontrôleur est la suivante (bien entendu, les registres ont la priorité sur l'EEPROM entre 0 et $3FF) :

0000-0FFF	EEPROM Array	4096
1000-3FFF	RAM Array	12288
4000-7FFF	Fixed Flash EEPROM Array incl 0.5K, 1K, 2K or 4K Protected Sector at start	16384
8000-BFFF	Flash EEPROM Page Window	16384
C000-FFFF	Fixed Flash EEPROM array incl 0.5K, 1K, 2K or 4K Protected Sector at end and 256 bytes of Vector Space at $FF80-$FFFF	16384

COURS n°2

Le codage de l'information

Généralités

On distingue dans la communication et le traitement de l'information deux techniques très générales : analogiques et numériques. Les techniques analogiques traduisent l'information par des grandeurs continues (déplacement d'une aiguille, valeur d'une tension, etc.), alors que les techniques numériques la traduisent par des valeurs discrètes (nombres).

Exemples : Montre analogique (à aiguilles) ou numérique,
Voltmètres analogiques ou numériques,
Disque analogique (vinyle) ou numérique (laser).

Les perceptions des organes des sens sont de nature analogique, et il peut sembler plus naturel et plus simple de traiter l'information sous forme analogique. Actuellement, dans presque tous les domaines, les techniques numériques prennent le pas sur les techniques analogiques pour des raisons de facilité de manipulation et de traitement.

Les unités de base de l'information numérique

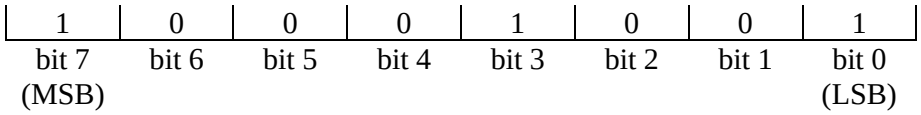
Le quantum (quantité la plus petite) d'information est représenté par un système à deux états (haut/bas, vrai/faux, ouvert/fermé, allumé/éteint, etc.). Toute information numérique peut être représentée par un ensemble de quanta d'information. On a développé diverses technologies pour représenter l'information numérique. Les calculatrices mécaniques utilisaient des systèmes à 10 états pour représenter les nombres entre 0 et 9.

A l'heure actuelle toute l'information numérique est décrite à partir de systèmes à 2 états (logique binaire), les plus faciles à réaliser et les plus fiables. L'unité associée est le bit (abréviation de l'anglais **BI**nary dig**IT** : chiffre binaire). Un des états est représenté par le chiffre 0 et l'autre par le chiffre 1. En micro électronique, le 0 est souvent associé à une tension nulle et le 1 à une tension 5V (standard TTL).

Pour décrire une information non élémentaire, on a besoin de plusieurs bits. Actuellement, ils sont regroupés au moins par 4, et le plus souvent par 8 ou des multiples de 8. Un ensemble de 4 bits est parfois appelé quartet (nibble en anglais). On appelle octet (byte en anglais) un ensemble de 8 bits, et mot (word en anglais) un ensemble de 16 bits. L'octet constitue l'unité pratique de base du traitement de l'information, plus riche que le bit tout en restant d'un maniement commode.

Les codages de l'information par des octets

Un bit pouvant prendre 2 valeurs, un ensemble de 8 bits pourra prendre 2^8 configurations différentes. Il existe donc 256 octets différents, qui constituent l'alphabet de codage de toute information. On a l'habitude de numéroter les bits de 0 à 7 et de droite à gauche. Le bit le plus à gauche est appelé bit de poids fort ou MSB (Most Significant Bit), tandis que le bit le plus à droite est appelé le LSB (Least Significant Bit) ou bit de poids faible.



Que peut représenter l'octet 10001001 ?
- 8 voyants (allumé, éteint, éteint, éteint, allumé, éteint, éteint, allumé),
- 8 commandes d'appareils (on, off, off, off, on, off, off, on),
- le caractère 'ë' dans le codage ASCII étendu,
- le nombre non signé 137,
- le nombre signé -119,
- un doigté sur un instrument de musique (piano ou flûte),
- l'instruction ADCA en mode d'adressage immédiat,
- etc..

De même qu'il est impossible de comprendre le mot '*four*' si on ignore si le texte est anglais ou français, on ne peut interpréter un octet indépendamment du contexte. Un bon exemple est celui d'un éditeur de texte : l'éditeur interprète les octets du fichier comme des caractères et les affiche comme tels à l'écran. Si le fichier a été prévu pour ça (fichier texte), on obtient un résultat lisible. Dans le cas contraire, on a un texte incompréhensible.

La notation hexadécimale

Il existe, on l'a dit, 256 octets différents, qui constituent l'alphabet de base de la micro-informatique. Il s'agit de leur donner à tous un nom facilement manipulable. Le nom '*unzérozérozérounzérozéroun*' est peu pratique à manipuler car trop long et trop monotone à l'oreille. On pourrait au contraire choisir un nom différent pour chaque octet : cette fois, ce serait trop long de mémoriser 256 noms et leur correspondance binaire.

On a choisi de donner un nom à chaque quartet :

0000 = **0** 0001 = **1** 0010 = **2** 0011 = **3** 0100 = **4** 0101 = **5** 0110 = **6** 0111 = **7**

1000 = **8** 1001 = **9** 1010 = **A** 1011 = **B** 1100 = **C** 1101 = **D** 1110 = **E** 1111 = **F**

De sorte que tout octet est représenté par deux symboles. Ainsi l'octet cité plus haut s'écrira **89** dans cette notation, appelée hexadécimale car elle fait appel à 16 symboles.

Il faut bien remarquer qu'il ne s'agit là que d'une notation permettant des échanges humains plus faciles, mais qu'il n'existe pas au niveau du processeur deux niveaux différents (hexadécimal et binaire). L'utilisateur n'a besoin de mémoriser que le tableau des 16 quartets pour reconstituer un octet binaire à partir de sa notation hexadécimale. Ainsi l'octet **35** s'écrira 0011 0101, et l'octet **BC** s'écrira 1011 1100.

Le codage des caractères

La typographie utilise les lettres de l'alphabet (26 majuscules et 26 minuscules), les chiffres (10), les ponctuations et les signes opératoires (≈24). Le total étant compris entre 64 et 128, on arrive à la conclusion qu'il suffit de 7 bits pour représenter les caractères, du moins dans les pays anglo-saxons qui ignorent les accents, les trémas et autres variations sur les caractères.

En l'absence de critère objectif, on a créé dans le passé différents codes de représentation des caractères. A l'heure actuelle, seul le code ASCII (**A**merican **S**tandard for **C**ommunication &

Information **I**nterchange) a survécu. Ce codage utilise les 96 octets de **20** à **7F** pour coder les caractères, chiffres, lettres et symboles divers d'opérations et de ponctuation. Les chiffres sont codés de **30** à **39**, les majuscules de **41** à **5A** et les minuscules de **61** à **7A**.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Les codes de **00** à **1F** ont été réservés à des usages spéciaux (caractères non imprimables) dont les plus connus sont :

08 : Back Step (BS). Touche de retour en arrière (et souvent d'effacement).
0A : Line Feed (LF). Descente d'une ligne.
0D : Carriage Return (CR). Retour en début de ligne.
1B : Escape (ESC). Touche d'échappement.

Extensions du code ASCII
- Certains des codes non imprimables ont été prévus pour des utilisations qui sont aujourd'hui totalement désuètes. C'est pourquoi on les affecte à de nouveaux caractères (as de pique, cœur, carreau, trèfle, etc.) plus ou moins bien pris en compte par les logiciels.
- Le codage sur 7 bits des caractères laisse un bit libre dans l'octet qui les représente. On utilise parfois ce bit pour des contrôles d'échanges de caractères (bit de parité).
- On peut aussi utiliser le 8^{e} bit pour coder jusqu'à 128 autres caractères (lettres accentuées, caractères graphiques, caractères spéciaux). Les codes ASCII étendus existent sous plusieurs formes et ne représentent pas un standard unique à l'heure actuelle.

Représentation ASCII d'un quartet. Comme on peut le constater, les codes des caractères de '0' à '9' vont de **30** à **39**, mais ceux des caractères 'A' à 'F' ne les suivent pas immédiatement puisqu'ils vont de **41** à **46**. Il y a un trou de 7 octets entre les deux séries de codes. C'est un défaut du code ASCII dont il faut tenir compte chaque fois qu'on veut extraire d'un quartet le code du caractère hexadécimal qui le décrit.

Le codage des nombres : Nombres non signés

Le codage le plus naturel des nombres par des octets est de considérer la représentation binaire d'un octet comme son écriture en base 2. La place du bit dans l'octet représente la puissance de 2 correspondante. Ainsi, l'octet **B5** est l'écriture en base 2 de :

B5 = 10110101 = $1\times2^7 + 0\times2^6 + 1\times2^5 + 1\times2^4 + 0\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$

= $1\times128 + 0\times64 + 1\times32 + 1\times16 + 0\times8 + 1\times4 + 0\times2 + 1\times1 = 181$

En base 2, table d'addition et table de multiplication ont toutes deux une simplicité élémentaire :

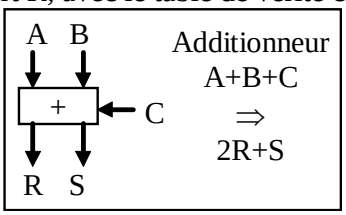
+	0	1
0	0	1
1	1	10

pour l'addition, et

×	0	1
0	0	0
1	0	1

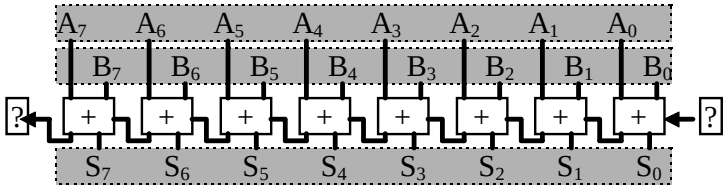
pour la multiplication,

La table d'addition se concrétise par un demi additionneur logique à deux entrées A et B et deux sorties S (chiffre des unités binaires) et R (chiffre des « dizaines »).
Pour pouvoir enchaîner une addition sur plusieurs chiffres binaires ou non, il est nécessaire de prévoir le report d'une retenue sur le chiffre suivant. L'électronique logique propose un additionneur répondant à la question. Il comporte trois entrées A, B et la retenue C, et deux sorties E et le report R, avec le table de vérité ci-contre :



A	B	C	S	R
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

On réalise un additionneur 2 octets en chaînant 8 additionneurs :



On remarque qu'il subsiste une indétermination sur le bit de report en entrée et en sortie. Ce bit porte le nom de Carry (report ou retenue). Il permet une grande souplesse d'utilisation.

Avec cette convention, un octet représente un entier de 0 (**00**) à 255 (**FF**). Le nombre 256 s'écrit en base 2 : 1 0000 0000 et n'est donc pas représentable sur un octet. Pour représenter des nombres plus grands, on doit nécessairement utiliser des bits supplémentaires. Un mot (deux octets, ou 16 bits) permet l'extension à des nombres plus grands. Il y a 2^{16} = 65536 mots différents, ce qui permet une représentation des entiers de 0 à 65535.

Remarque 1 : on peut écrire un octet en base 2 en regroupant les termes comme ci-dessous

$bit7\times2^7 + bit6\times2^6 + bit5\times2^5 + bit4\times2^4 + bit3\times2^3 + bit2\times2^2 + bit1\times2^1 + bit0\times2^0 =$

$\mathbf{[}bit7\times2^3 + bit6\times2^2 + bit5\times2^1 + bit4\times2^0\mathbf{]\ \times16} + \mathbf{[}bit3\times2^3 + bit2\times2^2 + bit1\times2^1 + bit0\times2^0\mathbf{]}$

Le premier crochet $[bit7\times2^3 + bit6\times2^2 + bit5\times2^1 + bit4\times2^0]$ représente un nombre entre 0 et 15, de même que le second $[bit3\times2^3 + bit2\times2^2 + bit1\times2^1 + bit0\times2^0]$, de sorte qu'on a écrit l'octet en. base 16. Si on appelle 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F les chiffres décrivant les nombres de 0 à 15 dans cette base, on retrouve la notation hexadécimale d'un octet.. L'avantage de l'hexadécimal est de fournir directement l'expression binaire de l'octet, (8=1000 et 9=1001), et sa valeur décimale (8×16 + 9 = 137). Alors qu'en décimal, pour retrouver le binaire de 137, il faut le décomposer soit en 128+8+1, soit par des divisions successives par 2. 137÷2=68 reste **1**(bit 0); 68÷2=34 reste **0**(bit 1); 34÷2=17 reste **0**(bit 2);17÷2=8 reste **1**(bit 3); 8÷2=4 reste **0**(bit 4); 4÷2=2 reste **0**(bit 5); 2÷2=1 reste **0**(bit 6); et 1÷2=0 reste **1(**(bit 7).

Remarque 2 : L'octet noté **23** représente le nombre décimal 35, et le nombre hexadécimal 23. Pour éviter toute confusion, on convient de faire précéder l'expression de la valeur numérique en base 16 du signe '$'. On écrira donc $23 = 35, ou $C8 = 200. Dans la suite, on confondra la représentation hexadécimale d'un octet avec l'expression en base 16 du nombre qu'il représente, c'est à dire qu'on abandonnera l'écriture en caractères gras au profit du signe '$'. En langage C, on écrit 0x23 pour préciser qu'on s'exprime en hexadécimal.

Arithmétique sur les octets. Indicateur C (Carry)

Pour ajouter deux octets, on utilisera l'additionneur 8 bits ci-dessus en mettant à 0 l'indicateur C. A la fin de l'addition, le Carry est à 0 s'il n'y a pas eu débordement et à 1 sinon.

Exemple 1 : Tout se passe bien, pas de retenue sur le bit 7. L'indicateur C de retenue est à 0.

0 0 1 1 0 1 1 0	($36=54)
+ 0 1 1 1 1 0 1 1	($7B=123)
1 0 1 1 0 0 0 1	($B1=177)

Exemple 2 : Le résultat exact dépasse la capacité de l'octet. Le résultat réduit à l'octet est donc tronqué ($5D = 93) et l'indicateur C de retenue est à 1.

1 0 0 1 1 0 0 1	($99=153)
+ 1 1 0 0 0 1 0 0	($C4=196)
1 0 1 0 1 1 1 0 1	($15D=349)

Exemple 3 : Cet exemple précise le précédent. On tourne en rond sur 256 valeurs, le successeur de 255 étant 0. L'indicateur C est bien sûr à 1.

1 1 1 1 1 1 1 1	($FF=255)
+ 0 0 0 0 0 0 0 1	($01=1)
1 0 0 0 0 0 0 0 0	($100=256)

Si on représente les nombres sur un octet, et si on veut respecter la loi d'addition binaire, tous les nombres égaux modulo 256 sont représentés par le même octet. Dans la convention des nombres non signés, les octets représentent les entiers de 0 à 255, et le carry fournit un contrôle de déroulement correct des opérations arithmétiques.

Le codage des nombres : Nombres signés (avec signe)

Le dernier exemple ci-dessus fait apparaître la lacune de la représentation précédente : les nombres négatifs ne sont pas représentés. On remédie à cela en prenant comme intervalle de définition l'intervalle symétrique (ou presque) de 256 entiers successifs (représentation en nombres signés) : [-128 à 127].

Opposé de n : c'est le nombre qui ajouté à n donne 0 :

0 1 1 0 0 1 1 1 ($67=103)	0 1 1 1 1 1 1 1 ($7F=127)	0 0 0 0 0 0 0 1 ($01=1)
+ 1 0 0 1 1 0 0 1 ($99=-103)	+ 1 0 0 0 0 0 0 1 ($81=-127)	+ 1 1 1 1 1 1 1 1 ($FF=-1)
1 0 0 0 0 0 0 0 0 ($00=0)	1 0 0 0 0 0 0 0 0 ($00=0)	1 0 0 0 0 0 0 0 0 ($00=0)

On remarque que le bit 7 des nombres négatifs est à 1 et celui des nombres positifs à 0. C'est pourquoi on appelle le bit 7 bit de signe. On note aussi que l'opposé de -128 ($80) est -128, ou plutôt que 128 et -128, qui diffèrent de 256, sont représentés par le même octet.

En nombres signés, l'indicateur C n'est pas significatif. Cette convention a besoin d'un autre indicateur (V : débordement) qui signale une anomalie au sens des nombres signés.

Exemple 1°

0 1 0 1 1 0 1 0 ($5A)	En nombres non signés : 90+143=233 OK
1 0 0 0 1 1 1 1 ($8F)	En nombres signés : 90+(-113)= -23 OK
1 1 1 0 1 0 0 1 ($E9)	C = 0 V = 0

Exemple 2°

0 1 0 0 1 1 0 0 ($4C)	En nombres non signés :76+241=61
1 1 1 1 0 0 0 1 ($F1)	En nombres signés : 76+(-15)=61 OK
0 0 1 1 1 1 0 1 ($3D)	C = 1 V = 0

Exemple 3°

0 1 0 0 1 1 0 0 ($4C)	En nombres non signés : 76+88=164 OK
0 1 0 1 1 0 0 0 ($58)	En nombres signés : 76+88=-92
1 0 1 0 0 1 0 0 ($A4)	C = 0 V = 1

Exemple 4°

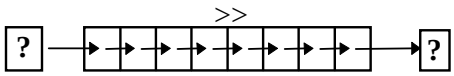
1 1 0 0 0 1 0 1 ($C5)	En nombres non signés : 197+179=120
1 0 1 1 0 0 1 1 ($B3)	En nombres signés : (-59)+(-77)=120
0 1 1 1 1 0 0 0 ($78)	C = 1 V = 1

Remarque : Le mécanisme de fonctionnement de l'addition de deux octets est indépendant de la convention choisie (nombres signés ou non), de même que le positionnement des indicateurs. Suivant le cas, on tiendra compte de C ou de V pour la validité des résultats.

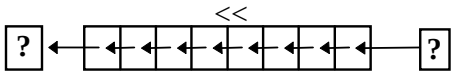
Décalages d'octets

On réalise en électronique des registres à décalage qui permettent une "translation" d'un octet vers la droite ou vers la gauche, c'est à dire une recopie simultanée de chacun des bits de l'octet sur son voisin de droite ou de gauche. Bien sûr un problème se pose pour les deux bits extrêmes, le premier qui n'a pas de précédent et dont il faut choisir l'état arbitrairement, et le dernier dont l'information n'est pas conservée dans l'octet.

décalage à droite

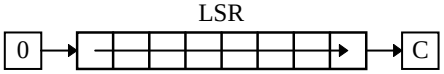


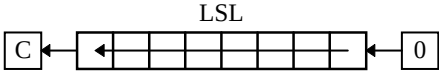
décalage à gauche



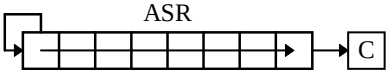
C'est encore le carry qui sert d'intermédiaire pour gérer le bit entrant ou sortant du décalage d'octet. Lorsque les octets représentent des nombres, un décalage vers la droite correspond en gros à une division par 2 et un décalage à gauche à une multiplication par 2.

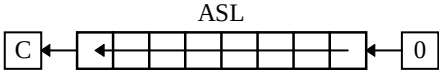
Le HCS12 a plusieurs instructions de décalage selon le choix des octets du bout. Ce sont :
- Les décalages logiques à droite ou à gauche (LSR et LSL : Logical Shift Right ou Left) dans lesquels l'octet qui sort va dans le Carry et l'octet qui rentre est un 0.



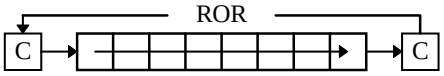


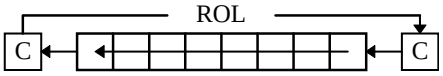
- Les décalages arithmétiques à droite et à gauche (ASR et ASL : Arithmetic Shift). Ils sont faits pour respecter la représentation des nombres signés quand c'est possible. ASL et LSL sont identiques (mêmes codes) tandis que ASR recopie le bit 7 sur lui même pour conserver le signe dans une division par 2.





- Les décalages circulaires à droite et à gauche (ROR et ROL : Rotate right ou left). Curieusement, le HC12 ne propose pas de permutation circulaire simple, mais rajoute systématiquement le Carry dans la ronde, ce qui donne :





Opérations logiques sur les octets.

Un bit représente parfaitement une variable booléenne. On définit sur ces variables l'opération unaire 'non' et les trois opérations binaires 'et', 'ou' et 'ou exclusif ' (eor) dont les tables de vérité figurent ci-dessous.

non	
0	1
1	0

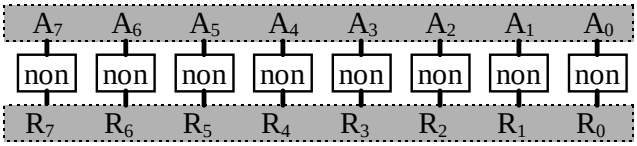
ou	0	1
0	0	1
1	1	1

et	0	1
0	0	0
1	0	1

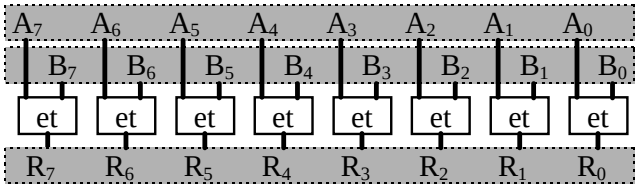
eor	0	1
0	0	1
1	1	0

On peut considérer un octet comme un ensemble de 8 variables binaires, et définir entre octets les quatre opérations **non**, **et**, **ou**, **eor** qui réalisent l'opération logique indiquée sur chacun de leurs bits homologues.

R = $\overline{A}$



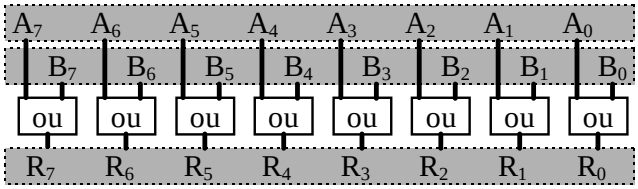
R = A et B



et	0	1
0	0	0
1	0	1

Table de vérité

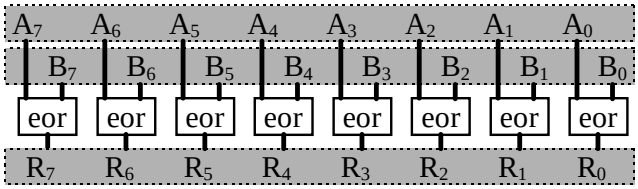
R = A ou B



ou	0	1
0	0	1
1	1	1

Table de vérité

R = A eor B



⊕	0	1
0	0	1
1	1	0

Table de vérité

par exemple		**non**	10101100	=	01010011	**non** $AC = $53
	10101100	**et**	11110000	=	10100000,	$AC **et** $F0 = $A0
	10101100	**ou**	11110000	=	11111100,	$AC **ou** $F0 = $FC
	10101100	**eor**	11110000	=	01011100	$AC **eor** $F0 = $5C

Les opérations logiques servent souvent à agir sur un seul bit d'un octet.
Pour mettre le bit 3 d'un octet à 1 sans toucher les autres, on fait un 'ou' de l'octet avec 8 :

XXXX XXXX **ou** 0000 1000 = XXXX 1XXX

Pour le mettre à 0, toujours sans toucher les autres, on fait un 'et' avec $F7 :

XXXX XXXX **et** 1111 0111 = XXXX 0XXX

Pour le changer de valeur, toujours sans toucher les autres, un 'eor' avec 8 :

XXXX XXXX **eor** 0000 1000 = XXXX $\overline{X}$ XXX

On a souvent à tester l'état d'un bit d'un octet. On réalise pour cela un 'masque' qui ne laisse 'passer' que le bit en question, par un 'et' logique : Ainsi, pour tester le bit 5 de l'octet X, on masquera l'octet X par $20, c'est à dire qu'on testera si (X et $20) est nul ou pas.

XXXX XXXX **et** 0010 0000 = 00X0 0000

COURS n°3

Les Aspects Logiciels du microcontrôleur Programmation en assembleur

Présentation du matériel

Comme on l'a indiqué en préambule à ce cours, on va étudier le microprocesseur HCS12 contenu dans le microcontrôleur MC9S12DP256B. A l'intérieur du microcontrôleur, il est connecté à de la mémoire, un timer, des contrôleurs de liaison série synchrone et asynchrone, des ports d'entrée-sortie logiques, un convertisseur analogique numérique, etc.

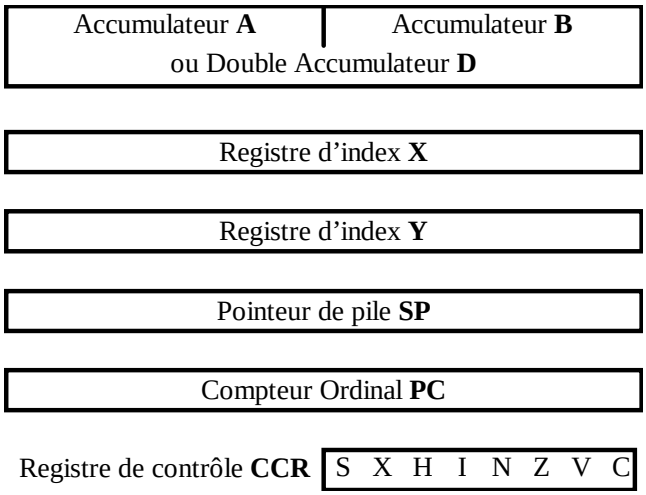
Le HCS12 (on parle aussi du CPU : **C**entral **P**rocessing **U**nit, pour la partie centrale) est de la famille Motorola. Pour les travaux pratiques on dispose d'une carte de développement (appelée HCS12 T-Board) qui peut communiquer avec un PC d'au moins deux façons différentes. L'utilisation d'un moniteur (D-BUG12) résidant dans le composant, et l'utilisation d'un debugger en ligne BDM.

On ne s'intéresse dans ce chapitre qu'au fonctionnement logiciel du microcontrôleur, c'est à dire qu'on dira « le µC ajoute », « le µC compare », « le µC range en mémoire », « le µC lit en mémoire », « le µC se branche à l'adresse... », sans se préoccuper pour l'instant du mécanisme électronique de chacune de ces opérations.

Les registres du microcontrôleur

Un microcontrôleur utilise en permanence un certain nombre de registres pour contrôler et traiter son information. On peut considérer ces registres comme des mémoires internes au CPU plus ou moins spécialisées par le jeu d'instructions qui leur sont attachées. Le nombre, les noms et les rôles des registres sont spécifiques à un microcontrôleur donné. Toutefois, on retrouve toujours, sous des noms différents et avec des modes d'utilisation variables, les mêmes types de registres dans tous les microcontrôleurs courants. Dans le cas du HCS12, les registres sont les suivants :

Registres du microprocesseur HCS12



Les accumulateurs A et B : deux registres de 8 bits utilisés systématiquement ou presque dans toutes les opérations de transfert mémoire, et la plupart des opérations arithmétiques et logiques. Les opérations d'addition par exemple se font suivant le schéma suivant :

$$A \leftarrow A + \text{valeur} \qquad \text{ou} \qquad B \leftarrow B + \text{valeur},$$

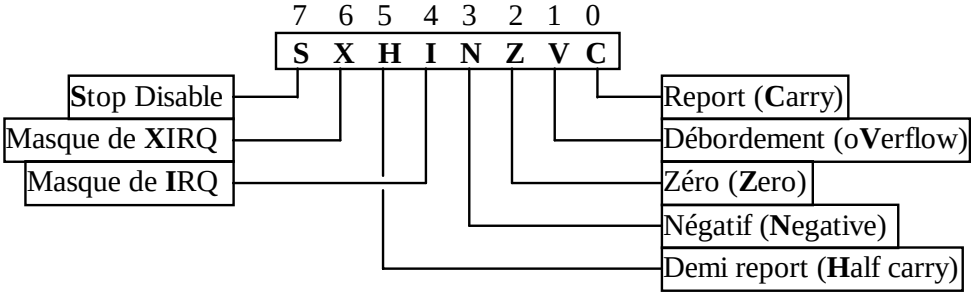
Il en est de même pour toutes les opérations à deux opérandes, ce qui explique le nom d'accumulateurs qu'on leur donne.

L'accumulateur D est le registre de 16 bits constitué par la juxtaposition des deux accumulateurs A et B (A poids fort et B poids faible) pour lequel il existe des instructions spécifiques. Il faut bien préciser que le registre D n'est pas distinct des registres A et B, mais une commodité qui permet de gérer A et B comme un seul registre 16 bits.

Les registres d'index (ou d'indice) X et Y : deux registres de 16 bits utilisables en tant que tels et spécialisés comme pointeurs de mémoire. Ils jouent un rôle fondamental dans la gestion des boucles par le mécanisme d'adressage indexé qu'on étudiera plus tard.

Le pointeur de pile SP (***S**tack **P**ointer*, pile se disant stack en anglais), registre de 16 bits réservé à la pile. La pile est une structure fondamentale dans toute l'informatique, et son importance est croissante avec la complexité des langages. On l'étudiera en temps voulu.
Le pointeur de programme PC (***P**rogram **C**ounter*) appelé aussi compteur ordinal, registre de 16 bits qui contient l'adresse de la prochaine instruction à exécuter. Dès qu'il est sous tension, le microcontrôleur exécute des instructions de programme. Avant d'exécuter une instruction, le microcontrôleur calcule l'adresse de l'instruction suivante et met à jour le PC. Son initialisation à la mise sous tension est le problème du RESET. Bien que cette mise à jour soit transparente, il ne faut pas perdre de vue que le CPU exécute sans cesse des instructions tant qu'il est sous tension.

Le registre de contrôle CCR (*Code Condition Register*). Registre de 8 bits, il constitue le tableau de bord du fonctionnement du microcontrôleur. Les bits 7, 6 et 4 sont des commandes, et les cinq autres des indicateurs (flags)



Les indicateurs

- Bit 0 : **C**arry . Positionné par la plupart des opérations arithmétiques et logiques, il permet notamment de tenir compte d'anomalies dans le résultat d'additions et de soustractions.
- Bit 1 : o**V**erflow . Positionné par certaines opérations arithmétiques et logiques, V permet de tenir compte d'anomalies dans les additions et soustractions en nombres signés.
- Bit 2 : **Z**éro . Positionné par la plupart des opérations arithmétiques et logiques, il indique
  (quand il est à 1 !) que le résultat de l'opération est 0
  (quand il est à 0 !) que le résultat de l'opération est $\neq 0$
- Bit 3 : **N**égatif . Positionné par la plupart des opérations arithmétiques et logiques, il recopie le bit 7 du résultat qui, en nombres signés, traduit le signe du résultat ('+'= 0, '-'= 1).
- Bit 5 : **H**alf Carry . Utilisation très spécifique dans les calculs BCD.

Les instructions de programmation

Le HCS12 est muni d'un certain nombre d'instructions de programmation. Ces instructions, codées sur un ou plusieurs octets, sont rangées séquentiellement en mémoire et constituent un programme. Il faut bien noter que les octets de programme (ou octets de code) n'ont rien de particulier, et peuvent être placés n'importe où dans la mémoire. Ils ne deviennent octets de code qu'au moment où le registre PC, en pointant sur eux, lance leur exécution.

On regroupe classiquement les instructions en trois familles (plus une pour les inclassables) :
- les opérations d'affectation (transfert de données),
- les opérations arithmétiques et logiques,
- les opérations de branchement,
- les opérations diverses.

Ecrire un programme consiste donc à traduire un algorithme (ou organigramme) en octets de programme et à ranger ces octets séquentiellement en mémoire. On lance l'exécution du programme en faisant pointer le PC sur le premier octet du code programme.

Pour faciliter cette écriture, on écrit un texte en utilisant un langage informatique intermédiaire doté d'une syntaxe rigoureuse (langage symbolique ou d'assemblage) où chaque instruction a un nom mnémonique. Le texte obtenu (programme source) est ensuite traduit en octets de code (assemblage) qui constituent le programme binaire. Cet assemblage peut se faire à la main à partir de la liste des instructions (notations symboliques et codes) qu'on trouve dans le manuel de référence du HCS12. Par la suite on utilise un logiciel (assembleur) qui génère automatiquement un fichier binaire de code exécutable à partir du texte source.

Les modes d'adressage

Une instruction se compose en général de plusieurs parties : une partie (opérateur) indique l'opération à effectuer (addition, comparaison, branchement, etc), et une seconde partie, éventuellement vide, indique le ou les opérandes sur lesquels va travailler l'instruction.

Exemple : l'addition du nombre constant n au contenu de la mémoire N° AD1 peut se faire de deux manières équivalentes par la succession de trois instructions :

1°/ AD1 ← AD1 + n
- a - Charger l'accumulateur A (ou B) avec le contenu de AD1 (variable)
- b - Additionner à l'accumulateur A (ou B) le nombre n (constante),
- c - Ranger le contenu de l'accumulateur A (ou B) dans la mémoire AD1

2°/ AD1 ← n + AD1
- a - Charger l'accumulateur A (ou B) avec le nombre n (constante),
- b - Additionner à l'accumulateur A (ou B) le contenu de AD1 (variable),
- c - Ranger le contenu de l'accumulateur A (ou B) dans la mémoire AD1

L'instruction a- charge l'accumulateur. Dans le cas N°1, avec le contenu d'une mémoire (une variable) et dans le cas N°2 par un nombre (une constante).
L'instruction b- additionne une valeur au contenu de l'accumulateur. Dans le cas N°1, il s'agit d'un nombre (une constante) et dans le cas N°2 du contenu d'une mémoire (une variable).
L'instruction c- range dans les deux cas le contenu de l'accumulateur dans la mémoire AD1 (affecte le résultat de l'addition à la variable).

De même que l'addition et les transferts mémoire, beaucoup de commandes on besoin d'accéder à une ou plusieurs opérandes. En général, comme dans l'exemple précédent, il existe plusieurs manières de désigner le mode d'accès à la (ou les) opérande(s). On les appelle modes d'adressage. Le codage des instructions se fait de la manière suivante : pour une même instruction, l'octet de code diffère suivant le mode d'adressage, et l'octet ou les octets suivant(s) précise(nt) comment définir l'opérande. Le HCS12 distingue 7 modes d'adressage : inhérent, immédiat, direct, étendu, relatif, indexé, multiple.

L'adressage inhérent est le plus évident : il n'y a pas d'opérande, et donc aucun adressage. L'instruction est codée sur un (ou deux) octet(s). Exemple :

mettre A à 0 s'écrit		**CLRA**	et se code	87
A ← A+B	s'écrit	**ABA**	et se code	18 06

L'adressage immédiat est celui d'une constante. C'est le cas des commandes 1°,b- et 2°,a-. L'instruction est traduite par un octet et la constante par 1 ou 2 octets suivant qu'il s'agit d'un octet ou d'un mot. Le langage symbolique traduit l'adressage immédiat par un # précédant l'opérande. Exemples :

charger A avec l'octet $30 s'écrit	**LDAA**	#$30	et se code	86 30
charger X avec le mot 5000 s'écrit	**LDX**	#5000	et se code	CE 13 88

L'adressage étendu est celui d'une (ou deux) cases mémoire dont on spécifie l'adresse dans l'opérande, sur 2 octets. C'est le cas des commandes 1°,a- et 2°,b-. Exemple :

charger A avec l'octet d'adresse $1000 s'écrit : **LDAA** $1000 et se code B6 10 00

charger X avec le mot d'adresse $1000 s'écrit : **LDX** $1000 et se code FE 10 00

Il faut noter que lors du chargement d'un registre 16 bits, deux octets sont concernés. Dans le cas précédent, l'octet d'adresse $1000 est rangé dans le poids fort de X, et l'octet d'adresse $1001 dans le poids faible de X.

L'adressage direct est un cas particulier de l'adressage étendu quand l'adresse est inférieure à $100. En réduisant l'opérande à un seul octet, cet adressage limite la taille et le temps d'exécution du code. *Dans un premier temps on peut l'ignorer*.

L'adressage relatif n'est utilisé que par les instructions de branchement et sera étudié dans le chapitre suivant (Ruptures de séquence).

L'adressage indexé fait l'objet du paragraphe suivant où on précisera ses différentes possibilités d'utilisation.

Quant à **l'adressage multiple**, il concerne des instructions à plusieurs opérandes ayant chacun son propre mode d'adressage.

Les deux exemples choisis en début de paragraphe donnent lieu aux programmes suivants (en supposant AD1 = $1000 et n = $45)

```
B6 10 00   LDAA $1000 ;étendu          86 45 LDAA #$45  ;immédiat
8B 45      ADDA #$45  ;immédiat        BB 10 00   ADDA $1000 ;étendu
7A 10 00   STAA $1000 ;étendu          7A 10 00   STAA $1000 ;étendu
      cas N°1                                cas N°2
```

On remarque que l'octet de code diffère suivant le mode d'adressage choisi.

Table 3-1. Sommaire des modes d'adressage du HCS12

Adressage	Format source	Abréviation	Description
Inhérent	**INST** (pas d'opérande exterieur)	INH	Les opérandes éventuels sont les registres CPU
Immédiat	**INST** #opr8i ou **INST** #opr 16i	IMM	L'opérande est inclus dans le flux d'instructions Taille 8 ou 16 bits suivant le contexte
Direct	**INST** opr 8a	DIR	L'opérande est les 8 bits poids faibles d'une adresse dans la zone $0000 $00FF
Etendu	**INST** opr16a	EXT	L'opérande est une adresse 16 bits.
Relatif	**INST** rel 8 ou **INST** rel 16	REL	L'instruction fournit un offset 8 ou 16 bits relatif au PC
Indexé (5 bits d'offset)	**INST** oprx5,xysp	IDX	Offset signé constant 5 bits par rapport à X, Y, SP ou PC
Indexé (pré-décrément)	**INST** oprx3,-xys	IDX	Pré décrément automatique de X,Y,SP de 1 à 8
Indexé (pré-incrément)	**INST** oprx3,+xys	IDX	Pré incrément automatique de X,Y,SP de 1 à 8
Indexé (post-décrément)	**INST** oprx3, xys-	IDX	Post décrément automatique de X,Y,SP de 1 à 8
Indexé (post-incrément)	**INST** oprx3, xys+	IDX	Post incrément automatique de X,Y,SP de 1 à 8
Indexé (offset accum)	**INST** abd, xysp	IDX	Indexé avec un offset accumulateur 8 bits (A ou B) ou 16 bits (D) par rapport à X, Y, SP ou PC
Indexé (9 bits d'offset)	**INST** oprx9,xysp	IDX1	Offset signé constant 9 bits / à X, Y, SP ou PC (les 8 bits faibles d'offset dans un octet d'extension)
Indexé (16 bits d'offset)	**INST** oprx16,xysp	IDX2	Offset signé constant 16 bits / à X, Y, SP ou PC (les16 bits d'offset dans deux octets d'extension)
Indexé indirect (16 bits d'offset)	**INST** [oprx16,xysp]	[IDX2]	On trouve le pointeur sur l'opérande à un offset constant de 16 bits par rapport à X, Y, SP ou PC (Offset 16 bits dans deux octets d'extension)
Indexé indirect (offset accu D)	**INST** [D,xysp]	[D,IDX]	On trouve le pointeur sur l'opérande à X, Y, SP ou PC plus la valeur dans D

Les adressages indexés

Dès que l'on veut décrire et utiliser une variable structurée, parcourir un tableau, on a besoin d'un opérande d'adresse variable. Les registres d'index ont été créés pour cela, et leur utilisation la plus simple, pour une instruction de chargement par exemple, peut se définir comme suit : « charger A avec le contenu de la mémoire d'adresse X »

La richesse de langage d'un microcontrôleur se mesure en partie à la variété de ses modes d'adressage indexés. Le HCS12 est bien pourvu de ce côté là et propose un grand choix. Un des points forts est que tous les registres de 16 bits (X, Y, mais aussi SP et PC) peuvent servir de base d'adressage indexé de manière (presque) interchangeable. On va simplement décrire les possibilités offertes, le détail de l'utilisation de tous les modes étant une affaire de pratique.

Adressage avec offset constant : supposons qu'on veuille charger A avec le contenu de l'adresse $1041, et que X ait la valeur $1000. On utilise alors l'instruction : LDAA $41,X, qui signifie « charger A avec le contenu de la mémoire N° X+$41 ». $41 est appelé offset. Avec

le HCS12, Motorola propose trois types d'offset constant :

sur 5 bits, signé, de -16 à +15, qui assure le maximum de rapidité,
sur 9 bits, signé, de -256 à +255
sur 16 bits, qui assure une portée maximum

Adressage avec offset accumulateur : Il est souvent très utile d'avoir un offset variable, et le HCS12 dispose d'adressages indexés avec A, B ou D comme offset. Exemple : LDAA D, SP

Adressage avec auto incrémentation ou auto décrémentation : Lorsque l'on décrit un tableau, on a souvent besoin d'incrémenter régulièrement le pointeur. Exemple : LDD 2,X+ signifie « charger D avec la mémoire d'adresse X (en fait, les deux mémoires d'adresses X et X+1), puis incrémenter X de 2.

Adressage indirect indexé : c'est le plus perfectionné, et le plus délicat, des modes d'adressage. On comprendra son fonctionnement sur un exemple : LDAA [20,X] signifie : lire l'adresse écrite en X+20, X+21 et charger A avec le contenu de cette adresse.
Le tableau suivant précise les détails et les codages de tous les modes indexés.

Table 3-2. Sommaire des opérations indexées

Code xb (Post octet)	Syntaxe du code **source**	**Commentaires** **rr ; 00=X, 01=Y,10=SP, 11=PC**
rr0nnnnn	,r n,r -n,r	**Offset constant 5 bits** n=-16 à +15 r peut spécifier X, Y, SP ou PC
111rr0zs	n,r -n,r	**Offset constant** (9 ou 16 bits signés) z- 0 = 9 bits avec signe dans LSB du post-octet -256≤n≤255 1 = 16 bits -32768≤n≤32767 si z = s = 1, indirect indexé 16 bits d'offset (voir plus bas) r peut spécifier X, Y, SP ou PC
111rr011	[n,r]	**Indirect indexé 16 bits d'offset** rr peut spécifier X, Y, SP ou PC -32768≤n≤32767
rr1pnnnn	n,-r n,+r n,r- n,r+	**pré-décrément, pré-incrément, post- incrément, post-décrément automatique** p = pre-(0) ou post-(1), n = -8 à -1, +1 à +8 r peut spécifier X, Y, SP. (PC n'est pas un choix valide) +8 = 0111 ... +1 = 0000 - 1 = 1111 ... - 8 = 1000
111rr1aa	A,r B,r D,r	**Offset accumulateur** (8 bits ou 16 bits non signés) aa - 00 = A 01 = B 10 = D (16 bits) 11 = voir offset indirect indexé sur l'accumulateur D r peut spécifier X, Y, SP ou PC
111rr111	[D,r]	**Offset indirect indexé sur l'accumulateur D** r peut spécifier X, Y, SP ou PC

Les opérations d'affectation générales

Ces opérations affectent une valeur à un registre ou une mémoire. L'origine de cette valeur peut être diverse et dépend du mode d'adressage, mais à l'exception des instructions MOVB et MOVW qui passent directement d'adresse à adresse, toutes ces instructions passent par un registre du CPU.

CLR,**CLRA**,**CLRB**	Mise à 0
LDAA,**LDAB**,**LDD**,**LDS**,**LDX**,**LDY**,**LEAS**,**LEAX**,**LEAY**	Chargement
STAA,**STAB**,**STD**,**STS**,**STX**,**STY**	Rangement
MOVB,**MOVW**	Transferts mémoire
SEX,TAB,TBA,TAP,**TFR**,TPA,TSX,TSY,TXS,TYS	Transferts de registres
EXG,XGDX,XGDY	Echanges de registres

Tous les modes d'adressage ne sont pas disponibles pour toutes les instructions. Chaque instruction a sa syntaxe propre qu'on trouvera dans la notice détaillée.

Opérations arithmétiques et logiques

Les opérations arithmétiques addition et soustraction se font toujours sur un accumulateur (A, B ou D). Il y a deux types d'addition et de soustraction, suivant qu'on tient compte ou non de la retenue (carry). Elles positionnent pratiquement toutes les indicateurs.

ADCA,**ADCB**,**ADDA**,**ADDB**,**ADDD**	Addition
SBCA,**SBCB**,**SUBA**,**SUBB**,**SUBD**	Soustraction

Les multiplications et les divisions sont sans opérande (voir mode d'emploi détaillé) de même que l'incrémentation (+1) la décrémentation (-1) et quelques opérations entre registres.

MUL,**EMUL**,**EMULS**,**EMACS**	Multiplication
EDIV,**EDIVS**,**FDIV**,**IDIV**,**IDIVS**	Division
DEC,**DECA**,**DECB**,DES,**DEX**,**DEY**	Décrémentation
INC,**INCA**,**INCB**,INS,**INX**,**INY**	Incrémentation
ABA,ABX,ABY,**SBA**	Diverses

Les décalages ont une importance fondamentale car ils représentent soit une multiplication par 2 (à gauche), soit une division par 2 (à droite). Il y a deux indéterminations : que devient le bit qui sort et quelle valeur a celui qui entre. C'est la raison pour laquelle il existe trois types de décalage sur le HCS12 : rotation, arithmétique et logique (voir mode d'emploi détaillé).

ASL,**ASLA**,**ASLB**,**ASLD**	Arithmétique à gauche
ASR,**ASRA**,**ASRB**,**ASRD**	Arithmétique à droite
LSL,**LSLA**,**LSLB**,**LSLD**	Logique à gauche
LSR,**LSRA**,**LSRB**,**LSRD**	Logique à droite
ROL,**ROLA**,**ROLB**	Rotation à gauche
ROR,**RORA**,**RORB**	Rotation à droite

Les opérations logiques se font presque exclusivement sur les accumulateurs A et B (à part la complémentation qui peut être directement sur une mémoire). Ce sont les opérations :

ANDA,**ANDB**,**ANDCC**,**ORAA**,**ORAB**,**ORCC**,**EORA**,**EORB**	Et,Ou,Ouexclusif
NEG,**NEGA**,**NEGB**	Négation
COM,**COMA**,**COMB**	Complémentation

Certaines opérations servent exclusivement à positionner les indicateurs sans modifier les registres sur lesquels elles réalisent des opérations logiques fictives.

BITA,BITB	Test de bits
CMPA,CMPB,CPD,CPS,CPX,CPY,CBA	Comparaison
TST,TSTA,TSTB	Test (comp à 0)

Notions sur la programmation en Assembleur

Le mot « assemblage » signifie traduction d'un programme source (en langage mnémonique) en un programme objet (une séquence d'octets) interprétable par le microprocesseur. Cette traduction est très généralement bijective, et on appelle désassemblage la fonction inverse qui interprète une séquence d'octets comme un programme dont elle reconstitue le mnémonique. Ces opérations peuvent se pratiquer à la main, en utilisant comme dictionnaires les tables de codage du langage de programmation du microprocesseur.

Assembleurs désassembleurs en ligne.
L'assemblage et le désassemblage sont des fonctions facilement automatisables, et il existe des programmes élémentaires capables de réaliser ces fonctions, généralement incorporés aux moniteurs de mise au point (Buffalo pour le kit HC 11). Un désassembleur a pour entrée une suite d'octets et fournit en sortie, quand c'est possible, un texte constitué de mnémoniques d'instructions. Un assembleur en ligne, lui, reçoit en entrée des chaînes de caractères décrivant l'instruction en langage mnémonique (codes mnémoniques et leurs opérandes qui sont soit des nombres, soit des adresses suivant le mode d'adressage), et, après analyse de la chaîne fabrique la suite d'octets correspondants. Qui dit analyse de chaîne dit obligatoirement syntaxe, syntaxe qu'il faut respecter dans l'écriture de la chaîne sous peine d'incompréhension de l'assembleur en ligne.

Assembleurs
La pratique de l'assembleur en ligne met vite en évidence un certain nombre de défauts impossibilité d'insérer simplement une ligne oubliée et manque d'étiquettes formelles sont les deux principaux.(Il est plus lisible d'écrire qu'on divise la mémoire NUMERATEUR par la mémoire DENOMINATEUR plutôt que la mémoire $543B par la mémoire $235F). Ces notions, qui peuvent très bien se manipuler lors de l'assemblage à la main, ne passent pas avec un assembleur en ligne. On utilise alors des outils plus évolués, un éditeur de texte qui prépare les chaînes à assembler, avec les mêmes facilités d'écriture qu'un éditeur normal, et un assembleur qui, à partir du fichier texte créé (fichier source, souvent noté ASM), fabrique la séquence d'octets de code dans un autre fichier (fichier objet, souvent noté S19).

Nouveautés par rapport au langage d'assemblage : essentiellement l'introduction d'identificateurs ou d'étiquettes, et l'utilisation de directives d'assemblage.

Syntaxe d'une ligne de texte destiné à l'assembleur. Chaque instruction occupe une ligne. On distingue quatre champs dans la ligne, tous facultatifs. Ce sont :
Champ étiquette : commence à la première colonne Une étiquette commence par une lettre
Champ instruction : séparé de la fin de l'étiquette par au moins un espace ou un ':'
Champ opérande : séparé du champ instruction par au moins un espace. Un opérande peut être une constante, une adresse ou une expression qui sera évaluée lors de l'assemblage.
Champ commentaire : commence par un point virgule. Toute la partie droite de la ligne est un commentaire; Est également commentaire une ligne commençant à gauche par une '*'

```
Etiquette      Instruction Opérande     Commentaire

DEBUT          LDAA #$45          ;ACCA = $45 (chargement de A immédiat)
               STAA COMPT         ;COMPT = ACCA (rangement de A à l'adresse
              COMPT)
               CLRA
```

*Les trois lignes précédentes constituent des exemples possibles de lignes d'assembleur

Nécessité de pseudo-instructions, ou directives d'assemblage.

Le choix des emplacements mémoire des octets ne posait pas de problème lors d'un assemblage en ligne. C'est l'opérateur qui allait lui même implanter ses octets ou bon lui semblait. Il doit maintenant communiquer les adresses d'implantation choisies à l'assembleur sous forme de la directive (ou pseudo-instruction) **ORG,** installée dans une ligne à part dans le champ instruction et suivie en opérande de l'adresse choisie.

Pour des raisons de confort, on veut donner un nom à une constante qui revient souvent dans un programme (par exemple l'octet $IB, code ASCII de la touche d'échappement). Cela se fait à l'aide de la directive **EQU,** dans le champ instruction, précédée de l'étiquette ESC et suivie de l'opérande $IB

Enfin si dans un problème on a besoin de variables nommées, par exemple une variable MASQUE d'un octet, une variable TEMPO de deux octets, la façon habituelle de procéder est de choisir une origine pour la zone des données, et d'y réserver 1 octet sous le nom de MASQUE et deux octets sous le nom de TEMPO, selon le schéma

```
            ORG  $1000
MASQUE      DS.B  1        ;Réservation d'un octet de mémoire
TEMPO       DS.W 1         ;Réservation de deux octets de mémoire
```

Cela consiste à prévenir l'assembleur que l'identificateur MASQUE = $1000 et l'identificateur TEMPO = $1 001 pour tout l'assemblage. La différence avec

```
MASQUE      EQU $1000
TEMPO       EQU $1 001
```

ne saute pas toujours aux yeux. L'utilisation de la seconde écriture ne nous assure pas d'une indépendance des adresses, et on peut accidentellement avoir deux étiquettes au même nom

```
char    MASQUE
int     TEMPO
```

Ces variables peuvent être initialisées, comme en C. On utilisera alors les directives **DC.B** et **DC.W**

Exemples

Ex N°1 Ajouter le nombre de 32 bits contenu dans les adresses `$1100`, `$1101`, `$1102` et `$1103` au nombre contenu dans les adresses `$1200`, `$1201`, `$1202` et `$1203`. On range le résultat aux adresses `$1300`, `$1301`, `$1302` et `$1303`. Autrement dit, effectuer l'addition suivante :

```
 [$1100] [$1101] [$1102] [$1103]
+[$1200] [$1201] [$1202] [$1203]
=[$1300] [$1301] [$1302] [$1303]
```

Résolution technique

Opérations successives		Instruction HC12
[$1103]+[$1203]	= [$1303] avec $carry_0$	**ADDA**
[$1102]+[$1202]+$carry_0$	= [$1302] avec $carry_1$	**ADCA**
[$1101]+[$1201]+$carry_1$	= [$1301] avec $carry_2$	**ADCA**
[$1100]+[$1200]+$carry_2$	= [$1300] avec $carry_3$	**ADCA**

Programmation HC12

Code instruction	Mnémonique	Description de l'action
1000 B6 1103	**LDAA** $1103	;A←octet d'adresse $1103
1003 BB 1203	**ADDA** $1203	;A=A+ octet d'adresse $1203
1006 7A 1303	**STAA** $1303	;A rangé à l'adresse $1303
1009 B6 1102	**LDAA** $1102	;A←octet d'adresse $1102
100C B9 1202	**ADCA** $1202	;A=A+C+octet d'adresse $1202
100F 7A 1302	**STAA** $1302	;A rangé à l'adresse $1302
1012 B6 1101	**LDAA** $1101	;A←octet d'adresse $1101
1015 B9 1201	**ADCA** $1201	;A=A+C+octet d'adresse $1201
1018 7A 1301	**STAA** $1301	;A rangé à l'adresse $1301
101B B6 1100	**LDAA** $1100	;A←octet d'adresse $1100
101E B9 1200	**ADCA** $1200	;A=A+C+octet d'adresse $1200
1021 7A 1300	**STAA** $1300	;A rangé à l'adresse $1300

Ex N°2 Le tableau de notes des élèves est constitué de la manière suivante : pour chaque élève, les notes des quatre épreuves figurent dans quatre cases mémoires successives. Une cinquième case est prévue pour y inscrire la moyenne finale. Si X contient l'adresse de la première case mémoire d'un élève, écrire le programme qui calcule et range la moyenne dans le tableau.

Résolution technique

On initialise une variable somme à 0. On y ajoute ensuite les différentes notes de l'élève pointées par X, X+1, X+2 et X+3. On divise le résultat par 4 à l'aide de deux décalages à droite, ce qui donne la moyenne (par défaut !) qu'on range dans la mémoire X+4.

Programmation HC12

Code instruction	Mnémonique		Description de l'action
1000 87	**CLRA**		;Au départ, Somme nulle
1001 AB 00	**ADDA**	0,x	;A=A+première note
1003 AB 01	**ADDA**	1,x	;A=A+seconde note
1005 AB 02	**ADDA**	2,x	;A=A+troisième note
1007 AB 03	**ADDA**	3,x	;A=A+p quatrième note
1009 47	**ASRA**		;A = A/2
100A 47	**ASRA**		;A = A/2
100B 6A 04	**STAA**	4,x	;Moyenne dans le tableau

Ex N°3 : La notation hexadécimale associe un caractère à chaque quartet. Elle est définie par un tableau de 16 octets, codes ASCII des 16 caractères hexadécimaux. Ce tableau H figure en mémoire à partir de l'adresse $1100 :

```
1100  30 31 32 33-34 35 36 37-38 39 41 42-43 44 45 46
0123456789ABCDEF
```

Ecrire un programme à qui on communique dans A un nombre de 0 à 15 et qui retourne le caractère correspondant, toujours dans A

```
LDX     #$1100   ;X pointe sur le début du tableau H
LDAA    A,X      ;A←H[A]. A est chargé avec l'élément N° A du tableau
```

COURS n°4

Les ruptures de séquence logicielles

Généralités

Habituellement les programmes du microcontrôleur s'exécutent de manière séquentielle. Plus précisément, le compteur ordinal PC contient l'adresse du premier octet de l'instruction à exécuter. Le microcontrôleur décode l'instruction, évalue le nombre d'octets de code qui la constituent et fait pointer le PC sur la prochaine instruction du programme. Il termine alors l'interprétation et l'exécution de l'instruction en cours avant de passer à l'instruction suivante.

On dit qu'il y a rupture de séquence si l'adresse de l'instruction suivante n'est pas celle calculée au cours du décodage. L'origine d'une telle rupture peut être logicielle (c'est l'instruction elle même qui modifie le contenu du compteur ordinal) ou matérielle (en général un changement d'état d'une ligne logique) auquel cas on parle d'interruption (→voir chapitre sur les Interruptions).

On distingue trois types de ruptures de séquence logicielles :
- **Ruptures inconditionnelles** : la modification du contenu du PC est systématique.
- **Ruptures conditionnelles** : la modification du PC est soumise à des conditions, exprimées par les bits d'état du registre de contrôle CCR.
- **Appel et retour de sous-programmes** : la modification est systématique comme dans un branchement inconditionnel, mais avec mémorisation de l'adresse de retour.

Instructions de rupture inconditionnelle

Le HCS12 possède trois instructions de rupture inconditionnelle de séquence :
- **JMP** suivi d'une adresse, qui charge cette adresse dans le PC.
- **BRA** suivi d'un déplacement (offset) sur un octet, qui ajoute ce déplacement à l'adresse contenue dans le PC. L'addition se fait en nombres signés. On dit que le branchement est relatif, car l'adresse de destination n'est pas absolue comme dans le cas de **JMP**. C'est un branchement à faible portée, puisqu'il ne peut déplacer le PC que de ±$80.
- **LBRA** suivi d'un déplacement (offset) sur deux octets, qui ajoute ce déplacement à l'adresse contenue dans le PC. Il s'agit encore d'un branchement relatif, mais à longue portée, puisqu'il peut déplacer le PC sur tout l'espace adressable de 0 à $FFFF.

Exemple : On trouve à l'adresse $1000 les octets $20 (code de **BRA**) et $35. Quelle est l'adresse de destination ? Lorsque le PC pointe sur $1000, le microcontrôleur lit le code $20, reconnaît que c'est un code à 2 octets et met à jour le PC à $1002. Puis il exécute l'instruction et ajoute $35 au PC, qui contiendra donc en fin d'instruction, $1037. Si le second octet avait été $C2 par exemple, la même opération aurait conduit à l'adresse $FC4.

$1000	$20 $35	**BRA**	$1037

$1000	$20 $C2	**BRA**	$FC4

Remarque 1 : l'instruction de saut **JMP** est absolue et permet une rupture de séquence à une adresse fixe. Les instructions **BRA** et **LBRA** sont relatives. Elles génèrent du code translatable, c'est à dire pouvant fonctionner tel quel indépendamment de son implantation.
Remarque 2 : l'instruction **LBRA** de branchement à une adresse relative lointaine est équivalente à un saut à une adresse indexée sur le registre PC :

LBRA Adresse ⇔ **JMP** Adresse, PCR

Instructions de rupture conditionnelle

Les ruptures conditionnelles n'existent dans le HCS12 que sous forme de branchement (long ou court), c'est à dire qu'elles sont relatives. Elles fonctionnent de la façon suivante : si la condition est réalisée, le branchement a lieu (ajout du déplacement au PC), sinon, l'instruction est ignorée (pas d'ajout au PC qui pointe sur l'instruction suivante).

Les branchements conditionnels sont déclenchés par les quatre indicateurs fondamentaux du registre d'état CCR : carry C, débordement V, zéro Z et signe N. Il existe 8 instructions de branchement conditionnel sur l'état (1 ou 0) de chacun des quatre indicateurs. Il faut y ajouter 6 branchements plus complexes relatifs aux nombres signés et déclenchés par certaines combinaisons des différents indicateurs. Les 14 instructions de branchement conditionnel se groupent deux par deux, déclenchées par des conditions contraires : Ex :

BCC : Branch if Carry Clear : branchement si le Carry est nul (0)

BCS : Branch if Carry Set : branchement si le carry est à 1 (1).

La plupart des instructions arithmétiques et logiques positionnent les indicateurs. Dans la mesure du possible, on s'efforce de placer l'instruction qui positionne les indicateurs actifs pour le branchement juste avant l'instruction de branchement. Si on a besoin d'insérer une ou plusieurs instructions entre l'instruction 'déclenchante' et le branchement, il est fondamental de s'assurer que ces instructions ne modifient pas les indicateurs concernés.

En général, les instructions qui positionnent les indicateurs (instructions arithmétiques ou logiques) modifient également le contenu des registres. Pour les positionner sans modifier les registres, le HCS12 dispose de trois familles d'instructions :
- Les comparaisons (**CMPA,CMPB,CPD,CPX,CPY,CBA**) positionnent les indicateurs comme une soustraction de l'opérande sans effectuer la soustraction.
- Les tests (**TST,TSTA,TSTB**) positionnent les indicateurs comme une comparaison à 0.
- Les tests de bits (**BITA,BITB**) positionnent les indicateurs comme un et logique avec l'opérande sans effectuer l'opération.

Exemple : écrire un programme à qui on donne dans A la valeur n d'un quartet et qui retourne le caractère hexa qui lui correspond, toujours dans A. (exercice traité au chapitre précédent avec un tableau). Ici, on n'utilise pas de tableau et on calcule le code ASCII en distinguant deux cas : n < 10 et le code est égal à n+$30, ou le n ≥ 10 et le code est égal à n + $37

```
        CMPA    #10     ;effectue fictivement la soustraction A=A-10. Positionne N
        BLO     PLUS    ;Si l'indicateur N est mis (A<10), on saute
        ADDA    #7      ;sinon (A≥10), A = A+7
PLUS    ADDA    #$30    ;De toute façon, on ajoute $30
ICI     BRA     ICI     ;Le programme boucle sur lui même
```

Le programme associe aux nombres 0 à 9 les codes ASCII $30 à $39 (caractères '0' à '9'), et aux nombres 10 à 15 les codes ASCII de $41 à $46 (caractères 'A' à 'F')

Déroulement d'un programme lu par le microcontrôleur

La suite ci-dessous représente un programme :

```
1000  79 20 04 B6 - 20 00 7A 20 - 05 87 C7 74 - 20 01 24 03
1010  F3 20 04 78 - 20 05 75 20 - 04 F7 20 01 - 26 ED 7C 20
1020  02 20 FE
```

On se propose de suivre son exécution à partir du moment où PC = $1000.

`PC = 1000` `[PC]=79` : code sur 3 octets
Lecture des 2 octets restants (`20 04`) et mise à jour de PC
décodage et exécution : mémoire N°2004 ← 0 (**`CLR`** `$2004`)

`PC = 1003` `[PC]=B6` : code sur 3 octets
Lecture des 2 octets restants (`20 00`) et mise à jour de PC
décodage et exécution : ACCA ← mémoire 2000 (**`LDAA`** `$2000`)

`PC = 1006` `[PC]=7A` : code sur 3 octets
Lecture des 2 octets restants (`20 05`) et mise à jour de PC
décodage et exécution : ACCA → mémoire 2005 (**`STAA`** `$2005`)

`PC = 1009` `[PC]=87` : code sur 1 octet
Mise à jour de PC : `PC = 100A`
décodage et exécution : ACCA ← 0 (**`CLRA`**)

`PC = 100A` `[PC]=C7` : code sur 1 octet
Mise à jour de PC : `PC = 100B`
décodage et exécution : ACCB ← 0 (**`CLRB`**)

`PC = 100B` `[PC]=74` : code sur 3 octets
Lecture des 2 octets restants (`20 01`) et mise à jour de PC
décodage et exécution : décalage à droite de 2001 (**`LSR`** `$2001`)

`PC = 100E` `[PC]=24` : code sur 2 octets
Lecture de l'octet restant (`03`) et mise à jour de PC : `PC = 1010`
décodage et exécution : si Carry = 0, PC ← PC + 3 (**`BCC`** `$1013`)

`Si carry`

`PC = 1010` `[PC]=F3` : code sur 3 octets
Lecture des 2 octets restants (`20 04`) et mise à jour du PC
décodage et exécution : D ← D + mémoires 2004 et 2005 (**`ADDD`** `$2004`)

`PC = 1013` `[PC]=78` : code sur 3 octets
Lecture des deux octets restants (`20 05`) et mise à jour du PC
décodage et exécution : décalage à gauche de `2005` (**`ASL`** `$2005`)

`PC = 1016` `[PC]=75` : code sur 3 octets
Lecture des deux octets restants (`20 04`) et mise à jour du PC
décodage et exécution : décalage à gauche de 2004 avec Carry (**`ROL`** `2004`)

`PC = 1019` `[PC]=F7` : code sur 3 octets
Lecture des deux octets restants (`20 01`) et mise à jour du PC
décodage et exécution : positionnement des flags suivant `$2001` (**`TST`** `$2001`)

`PC = 101C` `[PC]=26` : code sur 2 octets
Lecture de l'octet restant (`ED`) et mise à jour du PC : `PC = 101E`
décodage et exécution : si Z = 0 , PC ← PC + ED =000B (**`BNE`** `$000B`)

`Si Z`

`PC = 101E` `[PC]=7C` : code sur 3 octets
Lecture des 2 octets restants (`20 02`) et mise à jour du PC

décodage et exécution : D → mémoires 2002 et 2003 (**STD** $2002)

PC = 1021 [PC]=20 : code sur 2 octets

Lecture de l'octet restant (FE) et mise à jour du PC

décodage et exécution : PC ← PC + FE = $1021 (**BRA** $1021)

La pile et le pointeur de pile SP

Compte tenu du nombre réduit de registres, il arrive fréquemment que l'on ait besoin dans un calcul de mémoriser des valeurs intermédiaires. Le programmeur doit choisir pour cela des mémoires auxiliaires, dont il assure la gestion. Cette gestion devient vite fastidieuse et complexe si on veut utiliser au mieux l'espace mémoire. C'est entre autres pour répondre à ce besoin qu'on a imaginé la structure de pile, qui existe et fonctionne de la même façon sur tous les microcontrôleurs.

La pile est une zone de mémoire quelconque, pour laquelle on a prévu un pointeur spécial SP (Stack Pointer) et des instructions d'accès rapides (Push et Pull). Les instructions Push (**PSHA, PSHB, PSHC, PSHD, PSHX, PSHY**) servent à ranger les registres dans la pile alors que les instructions Pull (**PULA, PULB, PULC, PULD, PULX, PULY**) permettent de les récupérer. Toute l'astuce de la structure vient de ce qu'on n'a pas besoin de préciser l'adresse de rangement, automatiquement gérée par le microcontrôleur avec le pointeur de pile SP. On utilise parfois pour décrire la pile le terme LIFO (Last In First Out), pour bien préciser que le premier terme à sortir de la pile est le dernier entré.

Détail des instructions PUSH d'empilement (Avant l'instruction, SP pointe sur la valeur SP_0).

-1 octet : **PSHA**	SP = SP-1	décrémentation du pointeur de pile SP(SP_0 - 1)
	STAA [SP]	rangement de A à l'adresse N° SP (SP_0 - 1)
-1 mot : **PSHX**	SP = SP-2	décrémentation du pointeur de pile SP (SP_0 - 2)
	STX [SP]	rangement de X à l'adresse N° SP (SP_0-2, SP_0-1)

Détail des instructions PULL de dépilement (Avant l'instruction, SP pointe sur la valeur SP_0).

-1 octet : **PULA**	LDAA [SP]	chargement de A avec le contenu de SP (SP_0)
	SP = SP+1	incrémentation du pointeur de pile SP (SP_0+1)
-1 mot : **PULX**	LDX [SP]	chargement de X avec [SP_0, SP_0+1]
	SP = SP+2	incrémentation du pointeur de pile SP (SP_0+2)

Remarque : Tout programme doit initialiser le pointeur de pile à une valeur choisie. Si on utilise un moniteur résident, cette initialisation est assurée par le moniteur et on finit par l'oublier, car le mécanisme est transparent au programmeur qui se contente d'utiliser la zone de pile en mémoire. Il lui faut simplement tenir soigneusement la comptabilité de sa pile pour être sûr de récupérer les bonnes données. En cas de branchements conditionnels, il arrive fréquemment que la pile ne soit pas correctement gérée dans tous les cas. C'est une source d'erreurs redoutables à détecter quand certaines éventualités se produisent très rarement.

Exemples d'utilisation de la pile :

L'instruction TFR X,Y de transfert de X dans Y et l'instruction EXG X,Y d'échange des deux registres n'existent pas sur le HC11. On les réalise à l'aide de la pile selon le schéma suivant :

Ces exemples illustrent le fonctionnement de la pile, mais il s'agit d'utilisations marginales. Cette structure trouve sa pleine signification avec les appels de sous-programmes. On verra aussi plus tard qu'elle joue un rôle déterminant dans le passage des paramètres aux fonctions et à la définition de variables locales à ces fonctions.

Les appels de sous-programme

Le HCS12 dispose de deux instructions d'appel de sous-programme :
- **JSR** (Jump to SubRoutine), instruction d'appel à une adresse absolue,
- **BSR** (Branch to SubRoutine), instruction d'appel à une adresse relative proche,

Ces instructions fonctionnent de manière analogue aux instructions **JMP** et **BRA**, sauf qu'elles mémorisent avant le branchement l'adresse d'où l'appel est donné. Il existe bien sûr une instruction permettant la récupération de cette adresse : **RTS** (retour de sous programme), placée en fin de sous-programme.

La mémorisation de l'adresse de retour se fait par la pile : à la fin du décodage de l'instruction d'appel, PC pointe sur l'instruction suivante. L'instruction d'appel effectue deux actions : l'empilement de la valeur du PC et le saut à l'adresse demandée. A la fin du sous-programme, l'instruction RTS se borne à récupérer dans la pile l'adresse sauvegardée.
On peut décrire **JSR**, **BSR,** et **RTS** à l'aide des instructions fictives suivantes :

```
JSR  Adresse     ⇔    PUSH PC + JMP Adresse
BSR  Adresse     ⇔    PUSH PC + BRA Adresse
RTS              ⇔    PULL PC
```

Remarque 1 : un sous-programme peut a priori être appelé à toute étape du programme, et donc dans n'importe quel contexte (état des registres) du CPU. Si on veut ne pas perturber ce contexte, on sauvera au préalable les valeurs des registres utilisés, et il faudra les restituer en fin de sous-programme. La pile est là pour ça. *Remarque : il s'agit d'une possibilité, nullement d'une obligation. On peut aussi ne rien préserver. C'est le programme appelant qui le fait.*

Remarque 2 : l'instruction d'appel à une adresse relative lointaine, équivalente à **LBRA**, qui pourrait être **LBSR** comme cela existe sur d'autres processeurs Motorola, n'existe pas. Elle est remplacée par un appel indexé sur le registre PC :

```
LBSR  Adresse     ⇔    JSR   Adresse, PCR
```

*Pour être complet, il existe également une instruction **CALL** d'appel de sous-programmes en mémoire étendue, associée à l'instruction **RTC** (retour de Call). On n'en parle pas ici.*

Exemple 1 : Temporisation logicielle fixe : Il s'agit d'attendre un temps fixé en décrémentant le registre IX de $1000 à 0. Le sous-programme est installé à l'adresse 1100 :

```
1100 34        PSHX          ;Mise à l'abri du contenu de IX dans la pile
1101 CE 10 00  LDX    #$1000 ;IX=1000
1104 09        DEX           ;IX=IX-1 (positionne l'indicateur Z)
1105 26 FD     BNE    $01104 ;Bouclage si non Z (X ≠ 0)
1107 30        PULX          ;Sinon, c'est fini, on récupère IX en pile
1108 3D        RTS           ;Retour à l'instruction suivant l'appel
```

Exemple 2 : Sous programme de temporisation logicielle paramétrable : on peut réutiliser le programme ci-dessus, en supprimant les deux instructions **PSHX** et **PULX**, et en utilisant la valeur initiale de X comme paramètre. L'appel du sous-programme se fait en deux temps :

```
1000 CE 12 34 LDX    #$1234  ;Passage de la durée de la tempo par IX
1003 BD 01 00 JSR    $1100   ;Appel du sous programme de temporisation
1006 ........ ...    ...     ;Et cetera

1100 09       DEX            ;IX=IX-1 (positionne l'indicateur Z)
1101 26 FD    BNE    1100    ;Bouclage si non Z (X ≠ 0)
1103 39       RTS            ;Retour à l'instruction suivant l'appel
```

Dans la mesure où le X utilisé par le sous-programme vient du programme principal, on se dispense en général de le sauvegarder en entrée du sous-programme pour le récupérer en sortie, en laissant ce soin au programme appelant, s'il le désire.

Exemple 3 : écrire un sous-programme OCTHEX qui extrait d'un octet la chaîne de deux caractères hexadécimaux qui le décrivent. En entrée, l'octet est dans A, et X pointe sur l'adresse de rangement.

On veut pouvoir l'utiliser de la manière suivante

```
        LDAA    #$B4    ;A contient l'octet 10110100
        LDX     #$2000  ;et X l'adresse de la chaîne $2000
        BSR     OCTHEX  ;Appel du sous programme OCTHEX
        ...     ...

OCTHEX  PSHA            ;Mise à l'abri de l'octet à traiter
        ANDA    #$F     ;Extraction du quartet faible
        BSR     CARQ    ;Caractère hex décrivant le quartet faible
        TFR     A,B     ;rangé dans B
        PULA            ;Récupération de l'octet initial
        LSRA            ;|
        LSRA            ;|Extraction du quartet fort
        LSRA            ;|
        LSRA            ;|
        BSR     CARQ    ;Caractère hex décrivant le quartet fort
        STD     0,X     ;Rangement des deux caractères aux adresses X et X+1
        RTS
CARQ    CMPA    #10     ;|
        BLO     PLUS    ;|
        ADDA    #7      ;|Voir plus haut dans ce chapitre
PLUS    ADDA    #$30    ;|
        RTS             ;|
```

Le sous-programme CARQ a déjà été étudié dans ce chapitre et dans le précédent. Si on prend la version utilisant un tableau, le sous-programme CARQ va utiliser le pointeur X et donc perdre la valeur affectée à X à l'appel du sous-programme OCTHEX. Dans ce cas, elle doit être sauvegardée

- Soit au début de OCTHEX, pour être récupérée avant l'instruction **STD** 0,X
- Soit dans CARQ qui prend alors l'allure suivante :

```
PSHX            ;Mise à l'abri du X du programme appelant
LDX     #$1100  ;X pointe sur le début du tableau H
LDAA    A,X     ;A←H[A]. A est chargé avec l'élément N° A du tableau
PULX            ;Récupération de la valeur préservée
RTS             ;
```

Imbrication des sous-programmes

La gestion de la pile permet d'imbriquer des sous-programmes les uns dans les autres. La seule précaution à prendre est de s'assurer que la valeur du pointeur de pile est la même au moment d'exécuter le RTS qu'au moment où on est entré dans le sous-programme. A titre d'exemple, on a suivi l'évolution de la pile au cours de l'exécution du programme ci-dessous :

```
            org         $1000

main        ldaa        #3
            ldab        #4
            bsr         addi
            bra         *

addi        aba
            bsr         addi2
            rts

addi2       asla
            aba
            rts
```

PC			SP0-4	SP0-3	SP0-2	SP0-1	A	B	SP
1000	LDAA	#03	xx	xx	xx	xx	xx	xx	SP0
1002	LDAB	#04	xx	xx	xx	xx	03	xx	SP0
1004	BSR	1008	xx	xx	xx	xx	03	04	SP0
1008	ABA		xx	xx	10	06	03	04	SP0-2
100A	BSR	100D	xx	xx	10	06	07	04	SP0-2
100D	ASLA		10	0C	10	06	07	04	SP0-4
100E	ABA		10	0C	10	06	0E	04	SP0-4
1010	RTS		10	0C	10	06	0E	04	SP0-4
100C	RTS		10	0C	10	06	0E	04	SP0-2
1006	BRA	1006	10	0C	10	06	0E	04	PILI

Le tableau de droite trace l'exécution du programme. SP0 est la valeur de départ du pointeur de pile, représenté par un trait vertical gras dans la pile [SP0-4, SP0]

Ecriture d'un sous-programme

On part presque toujours d'un programme qui fournit le résultat recherché. En général, si ce programme n'existe pas on prend le temps de l'écrire. Ensuite, il s'agit de partager ce programme en deux parties : le programme « appelant », qui accèdera au sous programme par un **JSR** ou un **BSR**, et le sous-programme appelé qui se conforme à la syntaxe de l'appelant.

On peut considérer l'appelant comme un client et le sous-programme comme le travail d'un sous-traitant. Dans ce type de relations, c'est le client qui a toujours raison et c'est au sous-traitant de se plier à ses exigences. La difficulté est que le programmeur est souvent son propre client et qu'il modifie la syntaxe de l'échange pour faciliter son travail. La méthode recommandée est donc de rechercher le maximum de commodité dans le rôle du client, d'établir un cahier des charges et ensuite de s'y tenir dans le rôle du sous-traitant.

Une fois le sous-programme écrit, mis au point et testé, il reste à le documenter (paramètres en entrée, en sortie, profondeur de pile nécessaire, registres modifiés) car le client ne veut pas entrer dans le détail de l'écriture du sous-programme. C'est sur cette documentation et sur elle seule qu'il s'appuiera pour développer son ou ses programmes appelants.

COURS n°5

Les ports d'Entrée/Sortie logiques

Généralités

On l'a vu dans le chapitre précédent, sur les 112 pattes du microcontrôleur, 91 peuvent servir d'entrées sorties d'usage général. Ces pattes, regroupées par ports, sont gérées par trois modules différents :

Port A, *Port B*, *Port E* et *Port K* sont gérés par le module MEBI, partie du cœur du microcontrôleur. Les ports *PAD* ne fonctionnent que comme entrées. Ils sont gérés par les modules ATD. Enfin le reste des ports (*Port H*, *Port J*, *Port M*, *Port P*, *Port S* et *Port T*) sont gérés par le module spécifique des entrées / sorties logiques PIM.

Le nombre de pattes disponibles (91) n'étant pas un multiple de 8, certains ports ont moins de 8 bits : *Port K* en a 7 et *Port J* en a 4. Malgré toutes ces différences, les lignes d'entrée logiques ont un certain nombre de points communs qu'on va maintenant préciser.

Propriétés communes des ports

La philosophie de Motorola en ce qui concerne les entrées sorties logiques d'usage général est de leur accorder une large autonomie par rapport à leurs voisines. Dès lors, leur regroupement en « ports » est plutôt d'ordre administratif que purement logique. Cela permet d'associer à un groupe de 8 pattes des groupes de 8 bits qui définissent chacun une propriété de la patte correspondante.

Registre de direction de donnée : (DDR) Chacun de ses bits permet de choisir la direction de transfert d'une patte : à 0, en entrée, à 1 en sortie.

Registre d'entrée-sortie de données : (IOR)

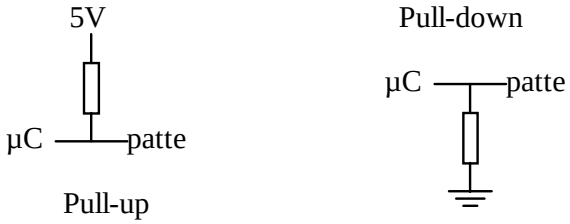
Si la patte est programmée en sortie, le bit correspondant fixe son niveau.
Si la patte est programmée en entrée, le bit correspondant traduit son niveau.

Registre d'entrée de données : (IR) Chacun de ses bits reflète le niveau de la patte associée, quelle que soit la direction programmée dans le registre de direction de données, et qu'il y ait ou non un module connecté.

Registre de réduction de sortance : (RDR) Un bit à 1 dans ce registre diminue la puissance maximum que peut « sortir » la patte associée dans un rapport 3 environ. Ce bit n'a de sens que si la patte est programmée en sortie. Il est ignoré dans le cas contraire.

Registre de validation de pull : (PE) Un bit à 0 laisse la patte correspondante « en l'air ». S'il est à 1, il valide le choix d'une résistance de tirage, pour fixer son niveau logique dans tous les cas. Ce bit est utilisé pour une patte programmée en entrée. Un pull-up (ou pull-down) étant généralement inutile en sortie.

Registre de choix de polarité du pull : (PS) Ce registre n'est utile que si le précédent a choisi une résistance de tirage. A 0, il s'agit d'un pull-up, et à 1, il s'agit d'un pull-down. Le schéma ci-dessous rappelle les schémas pull-up et pull-down.



Différenciation des ports : déclenchement d'interruptions

Trois ports (H, J et P) offrent la possibilité de déclencher des interruptions sur front de chacune des lignes d'entrée. Ils sont dans ce but munis d'un registre PIF de huit indicateurs (un par patte). L'indicateur passe à 1 sur front actif (montant ou descendant) de l'entrée associée.

Le front actif est défini par le choix du mode de tirage. Le pull-up maintient un niveau haut en absence de signal. C'est donc un passage au niveau bas (front descendant) qui sera le front actif. Au contraire, le pull-down maintient un niveau bas en l'absence de signal. C'est un passage au niveau haut (front montant), qui sera l'événement actif.

Pour remettre à 0 l'indicateur monté à 1 sur un front actif, il faut y écrire un 1. L'écriture d'un 0 est sans effet. Cette façon de faire déroute au premier abord, mais elle permet une écriture simple des choses. Si on attend par exemple une montée du bit 5, on utilise le masque %00100000 appliqué au registre PIF. Pour le remettre à 0 sans toucher les autres, il suffit d'écrire le masque dans PIF puisque l'écriture des bits nuls est sans effet.

Chacun des ports H, J et P dispose de son vecteur d'interruption. Il dispose également d'un registre PIE de validation d'interruption calqué sur le registre d'indicateurs PIF. Un bit à 1 de ce registre permet de déclencher une interruption sur montée de l'indicateur correspondant.
Ainsi, si le registre PIE contient %00001000, la montée du flag 3 de PIF déclenche une demande d'interruption du CPU. La montée des autres ne déclenche pas d'interruption.

Vue d'ensemble

	PIM						MEBI				ATD	
Registre	**H**	**J**	**M**	**P**	**S**	**T**	**A**	**B**	**E**	**K**	**Pad0**	**Pad 1**
(IOR) Entrée sortie du port	+	+	+	+	+	+	+	+	+	+	+	+
(IR) Entrée du port	+	+	+	+	+	+						
(DDR) Direction de données du port	+	+	+	+	+	+	+	+	+	+		
(RDR) Réduction de sortance du port	+	+	+	+	+	+						
(PER) Validation du pull du port	+	+	+	+	+	+						
(PSR) Choix de polarité du port	+	+	+	+	+	+						
(IER) Validation d'interruptions du port	+	+		+								
(IFR) Indicateurs d'interruptions du port	+	+		+								

Le partage des pattes

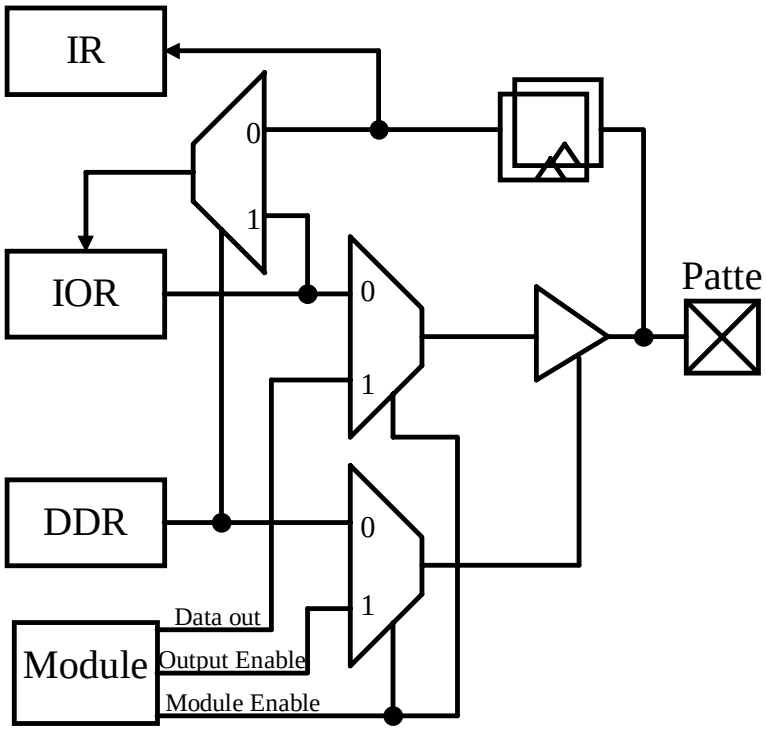
Presque toutes les pattes des ports ont plusieurs affectations possibles. Ainsi, quand on travaille en mode étendu, *Port A* et *Port B* deviennent bus d'adresses et bus de données multiplexés, les lignes du *Port E* deviennent les lignes de gestion du bus externe et le *Port K* sert à gérer l'extension de la mémoire adressable au delà de 64K.

Les pattes des deux ports *PAD*, qui fonctionnent uniquement en entrée sont le plus souvent utilisées comme canaux d'entrée analogique vers les convertisseurs.

Quant aux autres ports, ils peuvent prêter leurs pattes aux modules suivants (entre autres) :

Port H	Liaisons série synchrones SPI1 et SPI2
Port J	Bus IIC, CAN4
Port M	Bus CAN (CAN0, CAN1, CAN2, CAN3) et/ou BDLC
Port P	Variateur de largeur d'impulsion (PWM)
Port S	Liaisons série synchrone et asynchrone (SPI0, SCI0 et SCI1)
Port T	Timer (ECT).

La cohabitation entre ports d'entrée sortie logiques et modules se règle par des registres liés aux modules. Un principe simple, mais qui souffre peut-être quelques exceptions, est que la patte perd ses propriétés d'entrée sortie logique dès qu'elle est affectée à un module. Ainsi, le module peut l'utiliser comme sortie même si le bit du DDR est à 0. Le DDR n'a simplement plus cours dans ces conditions. La figure ci-dessous précise les relations entre registres de ports et modules.



Le bit IR ne peut être que lu. Il reproduit toujours l'état physique de la patte.

Le bit IOR reproduit l'état physique de la patte s'il est en entrée (DDR = 0). En sortie (DDR = 1), il lit le buffer

Si module enable = 0 (pas de module), et DDR = 1 (sortie), la patte recopie IOR.

En présence de module, IOR et DDR sont court-circuités et la patte prend la donnée fournie par le module. On remarque bien que le transfert de la donnée depuis le module vers la patte se fait même si DDR = 0.

Exemples d'utilisation des ports

Exemple 1 : programmer les lignes du port A en sortie sauf la ligne *PA7* en entrée. Afficher en permanence en *PA0* le niveau lu en *PA7*.

```
86 7F          LDAA  #%01111111  Programmation de la direction des lignes
5A 02          STAA  DDRA        Toutes en sortie sauf PA7
86 80  RECOM   LDAA  #%10000000  Récupération du niveau
94 00          ANDA  PORTA       de la ligne PA7
27 06          BEQ   ZERO        Si c'est 1,
86 01          LDAA  #1          On prépare le registre A
9A 00          ORAA  PORTA       pour écrire un 1 dans PA0
20 04          BRA   PLUS        Si c'est 0,
86 FE  ZERO    LDAA  #%11111110  On prépare le registre A
94 00          ANDA  PORTA       pour écrire un 0 dans PA0
5A 00  PLUS    STAA  PORTA       Mise de PA0 au niveau sans toucher les autres
20 EC          BRA   RECOM       Et bouclage
```

Exemple 2 : Compter le nombre de fronts descendants sur la patte *Port H0* par scrutation.

```
79 2000        CLR   CPT         ;Compteur à 0 au départ
86 FF          LDAA  #$FF        ;Pull-up pour tout le port H
7A 0264        STAA  PERH        ;(ou front descendant actif)
86 01          LDAA  #1          ;test du bit 0 de PIFH
B5 0267  ATT   BITA  PIFH        ;(attente d'une transition
27 FB          BEQ   ATT         ;active de la ligne PH0).
72 2000        INC   CPT         ;Quand c'est le cas, on incrémente le compteur
7A 0267        STAA  PIFH        ;et on abaisse le flag (en écrivant un 1 dedans)
20 F3          BRA   ATT         ;et on reboucle
```

Exemple 3 : Compter le nombre de fronts descendants sur la patte *Port H0* par interruption.

```
79 2000        CLR   CPT         ;Compteur à 0 au départ
86 FF          LDAA  #$FF        ;Pull-up pour tout le port H
7A 0264        STAA  PERH        ;(ou front descendant actif)
86 01          LDAA  #1          ;
7A 0267        STAA  PIFH        ;On abaisse le flag (rien ne s'est encore passé)
14 10          SEI               ;Invalidation des interruptions au niveau du CPU
B5 0266        STAA  PIEH        ;Validation d'inter sur front descendant de PH0
CE 101E        LDX   #ETDUN      ;Installation de la tâche en écrivant le
7E 3E4C        STX   $3E4C       ;vecteur d'interruption à l'adresse RAM
10 EF          CLI               ;active de la ligne PH0).
20 FE          BRA   *           ;Programme principal

72 2000  ETDUN INC   CPT         ;On incrémente le compteur
86 01          LDAA  #1          ;et on abaisse le flag
7A 0267        STAA  PIFH        ;(en écrivant un 1 dedans)
20 F3          RTI               ;Fin de tâche
```

COURS n°6

Utilisation de la Pile

La pile et le pointeur de pile SP

Compte tenu du nombre réduit de registres, il arrive fréquemment que l'on ait besoin dans un calcul de mémoriser des valeurs intermédiaires. Le programmeur doit choisir pour cela des mémoires auxiliaires, dont il assure la gestion. Cette gestion devient vite fastidieuse et complexe si on veut utiliser au mieux l'espace mémoire. C'est entre autres pour répondre à ce besoin qu'on a imaginé la structure de pile, qui existe et fonctionne de la même façon sur tous les microcontrôleurs.

La pile est une zone de mémoire quelconque, pour laquelle on a prévu un pointeur spécial SP (Stack Pointer) et des instructions d'accès rapides (Push et Pull). Les instructions Push (**PSHA, PSHB, PSHC, PSHD, PSHX, PSHY**) servent à ranger les registres dans la pile alors que les instructions Pull (**PULA, PULB, PULC, PULD, PULX, PULY**) permettent de les récupérer. Toute l'astuce de la structure vient de ce qu'on n'a pas besoin de préciser l'adresse de rangement, automatiquement gérée par le microcontrôleur avec le pointeur de pile SP. On utilise parfois pour décrire la pile le terme LIFO (Last In First Out), pour bien préciser que le premier terme à sortir de la pile est le dernier entré.

Détail des instructions PUSH d'empilement (Avant l'instruction, SP pointe sur la valeur SP_0).

-1 octet : **PSHA**	SP = SP-1	décrémentation du pointeur de pile SP(SP_0 - 1)
	STAA [SP]	rangement de A à l'adresse N° SP (SP_0 - 1)
-1 mot : **PSHX**	SP = SP-2	décrémentation du pointeur de pile SP (SP_0 - 2)
	STX [SP]	rangement de X à l'adresse N° SP (SP_0-2, SP_0-1)

Détail des instructions PULL de dépilement (Avant l'instruction, SP pointe sur la valeur SP_0).

-1 octet : **PULA**	LDAA [SP]	chargement de A avec le contenu de SP (SP_0)
	SP = SP+1	incrémentation du pointeur de pile SP (SP_0+1)
-1 mot : **PULX**	LDX [SP]	chargement de X avec [SP_0, SP_0+1]
	SP = SP+2	incrémentation du pointeur de pile SP (SP_0+2)

Remarque : Tout programme doit initialiser le pointeur de pile à une valeur choisie. Si on utilise un moniteur résident, cette initialisation est assurée par le moniteur et on finit par l'oublier, car le mécanisme est transparent au programmeur qui se contente d'utiliser la zone de pile en mémoire. Il lui faut simplement tenir soigneusement la comptabilité de sa pile pour être sûr de récupérer les bonnes données. En cas de branchements conditionnels, il arrive fréquemment que la pile ne soit pas correctement gérée dans tous les cas. C'est une source d'erreurs redoutables à détecter quand certaines éventualités se produisent très rarement.

Exemples d'utilisation de la pile :
L'instruction TFR X,Y de transfert de X dans Y et l'instruction EXG X,Y d'échange des deux registres n'existent pas sur le HC11. On les réalise à l'aide de la pile selon le schéma suivant :

Ces exemples illustrent le fonctionnement de la pile, mais il s'agit d'utilisations marginales. Cette structure trouve sa pleine signification avec les appels de sous-programmes. On verra aussi plus tard qu'elle joue un rôle déterminant dans le passage des paramètres aux fonctions et à la définition de variables locales à ces fonctions.

Les appels de sous-programme

Comme nous l'avons déjà vu, le HCS12 dispose de deux instructions d'appel de sous-programme :
- **JSR** (Jump to SubRoutine), instruction d'appel à une adresse absolue,
- **BSR** (Branch to SubRoutine), instruction d'appel à une adresse relative proche,

Ces instructions fonctionnent de manière analogue aux instructions **JMP** et **BRA**, sauf qu'elles mémorisent avant le branchement l'adresse d'où l'appel est donné. Il existe bien sûr une instruction permettant la récupération de cette adresse : **RTS** (retour de sous programme), placée en fin de sous-programme.

La mémorisation de l'adresse de retour se fait par la pile : à la fin du décodage de l'instruction d'appel, PC pointe sur l'instruction suivante. L'instruction d'appel effectue deux actions : l'empilement de la valeur du PC et le saut à l'adresse demandée. A la fin du sous-programme, l'instruction RTS se borne à récupérer dans la pile l'adresse sauvegardée.

On peut décrire **JSR**, **BSR,** et **RTS** à l'aide des instructions fictives suivantes :

```
JSR  Adresse    ⇔   PUSH PC + JMP Adresse
BSR  Adresse    ⇔   PUSH PC + BRA Adresse
RTS             ⇔   PULL PC
```

Remarque 1 : un sous-programme peut a priori être appelé à toute étape du programme, et donc dans n'importe quel contexte (état des registres) du CPU. Si on veut ne pas perturber ce contexte, on sauvera au préalable les valeurs des registres utilisés, et il faudra les restituer en fin de sous-programme. La pile est là pour ça. *Remarque : il s'agit d'une possibilité, nullement d'une obligation. On peut aussi ne rien préserver. C'est le programme appelant qui le fait.*

Exemple d'utilisation de la pile

Exemple 1 : Temporisation logicielle fixe : Il s'agit d'attendre un temps fixé en décrémentant le registre IX de $1000 à 0. Le sous-programme est installé à l'adresse 1100 :

```
1100 34        PSHX          ;Mise à l'abri du contenu de IX dans la pile
1101 CE 10 00  LDX    #$1000 ;IX=1000
1104 09        DEX           ;IX=IX-1 (positionne l'indicateur Z)
1105 26 FD     BNE    $01104 ;Bouclage si non Z (X ≠ 0)
1107 30        PULX          ;Sinon, c'est fini, on récupère IX en pile
1108 3D        RTS           ;Retour à l'instruction suivant l'appel
```

Exemple 2 : Sous programme de temporisation logicielle paramétrable : on peut réutiliser le programme ci-dessus, en supprimant les deux instructions **PSHX** et **PULX**, et en utilisant la valeur initiale de X comme paramètre. L'appel du sous-programme se fait en deux temps :

```
1000 CE 12 34  LDX    #$1234 ;Passage de la durée de la tempo par IX
1003 BD 01 00  JSR    $1100  ;Appel du sous programme de temporisation
1006 ........  ...    ...    ;Et cetera

1100 09        DEX           ;IX=IX-1 (positionne l'indicateur Z)
1101 26 FD     BNE    1100   ;Bouclage si non Z (X ≠ 0)
1103 39        RTS           ;Retour à l'instruction suivant l'appel
```

Dans la mesure où le X utilisé par le sous-programme vient du programme principal, on se dispense en général de le sauvegarder en entrée du sous-programme pour le récupérer en sortie, en laissant ce soin au programme appelant, s'il le désire.

Imbrication des sous-programmes

La gestion de la pile permet d'imbriquer des sous-programmes les uns dans les autres. La seule précaution à prendre est de s'assurer que la valeur du pointeur de pile est la même au moment d'exécuter le RTS qu'au moment où on est entré dans le sous-programme. A titre d'exemple, on a suivi l'évolution de la pile au cours de l'exécution du programme ci-dessous :

```
            org       $1000

main        ldaa      #3
            ldab      #4
            bsr       addi
            bra       *

addi        aba
            bsr       addi2
            rts

addi2       asla
            aba
            rts
```

PC			SP0-4	SP0-3	SP0-2	SP0-1	A	B	SP
1000	LDAA	#03	xx	xx	xx	xx	xx	xx	SP0
1002	LDAB	#04	xx	xx	xx	xx	03	xx	SP0
1004	BSR	1008	xx	xx	xx	xx	03	04	SP0
1008	ABA		xx	xx	10	06	03	04	SP0-2
100A	BSR	100D	xx	xx	10	06	07	04	SP0-2
100D	ASLA		10	0C	10	06	07	04	SP0-4
100E	ABA		10	0C	10	06	0E	04	SP0-4
1010	RTS		10	0C	10	06	0E	04	SP0-4
100C	RTS		10	0C	10	06	0E	04	SP0-2
1006	BRA	1006	10	0C	10	06	0E	04	PILI

Le tableau de droite trace l'exécution du programme. SP0 est la valeur de départ du pointeur de pile, représenté par un trait vertical gras dans la pile [SP0-4, SP0]

COURS n°7

Les Ruptures de séquences matérielles
Les Interruptions

Les ruptures de séquence matérielles : interruptions

Une interruption, comme un sous programme, est une rupture temporaire de la séquence d'instructions en cours pour exécuter une action définie, avant de reprendre la séquence où on l'avait laissée. A la différence d'un sous-programme où la rupture est déclenchée par une instruction logicielle (BSR ou JSR), donc parfaitement prévue, la rupture de séquence d'une interruption est commandée par matériel (niveau ou transition d'une ligne) ou de façon interne (interruptions du MCU). Elle peut donc se produire à n'importe quel stade de déroulement d'un programme.

Pour un sous-programme, l'adresse de branchement est indiquée dans l'appel. Pour une interruption, cette adresse (vecteur d'interruption) figure dans une table placée en haut de mémoire, chaque ligne d'interruption ayant son adresse propre (adresse d'interruption).

Le microcontrôleur dispose de trois lignes de demande d'interruptions externes :
- La **ligne *RESET*** ne peut être masquée, c'est à dire que toute demande d'interruption sur cette ligne (niveau bas) parvient au processeur où elle reçoit un traitement immédiat.
- La **ligne *XIRQ*** peut être masquée par le bit X du CCR, mais une fois démasquée reste active jusqu'au prochain RESET. A l'origine, il s'agissait d'une NMI (**N**on **M**askable **I**nterrupt). On a introduit le masque X pour résoudre des conflits avec le RESET au démarrage.
- La **ligne *IRQ*** peut être masquée par le bit I du CCR. Quand ce bit est à 1, les demandes d'interruption sur la ligne sont bloquées et ne parviennent pas au microprocesseur.

En plus de ces trois lignes, le microcontrôleur dispose de quelques tâches d'interruption internes qui peuvent s'activer indépendamment des lignes physiques. Ce sont notamment les interruptions pour défaut d'alimentation ou d'horloge, l'interruption de mise en route initiale, l'interruption pour code opératoire illégal, et l'interruption logicielle **SWI**. De plus, pratiquement tous les périphériques internes au microcontrôleur disposent d'au moins une ligne d'interruption personnalisée.

En fait, la ligne *RESET* est réservée à la réinitialisation du programme de base du processeur, et toutes les demandes d'interruption venant des périphériques externes au CPU passent par les deux lignes *IRQ* et *XIRQ*. Le premier travail de chacune des deux tâches d'interruption est d'identifier le demandeur parmi ceux connectés à l'entrée concernée, avant de s'aiguiller vers le traitement spécifique. Chacun des demandeurs doit donc disposer d'un indicateur de demande d'interruption que la tâche puisse tester.

Au début d'une tâche d'interruption, tous les registres sont préservés dans la pile et le masque I est mis pour ne pas imbriquer les interruptions. Le programmeur peut passer outre et valider une imbrication de tâches par l'instruction CLI, à ses risques et périls. L'interruption étant par nature imprévisible, cette sauvegarde du contexte est obligatoire. Cela interdit tout passage de paramètre par les registres et oblige à n'utiliser que des variables globales.

A la fin de la tâche d'interruption, la demande physique ne doit plus être active, sinon on repart immédiatement dans la même tâche. Il appartient donc à la tâche de faire le nécessaire pour qu'il en soit ainsi (acquittement de la demande). La plupart du temps, il suffit de remettre à 0 l'indicateur responsable.

Points communs avec les sous-programmes : Comme pour un sous-programme, il s'agit de détourner le programme principal pour effectuer une tâche précise (la tâche d'interruption), et d'y revenir une fois la tâche accomplie. Comme pour un sous-programme, on mémorise dans la pile l'adresse de retour (qui suit l'adresse de l'instruction interrompue) adresse que l'on dépile en fin de tâche. On doit également sauvegarder le contexte. Cette mesure est plus radicale dans le cas d'une interruption car on ignore à quel endroit du programme intervient la rupture de séquence (IRQ = Interrupt ReQuest). On préserve dans la pile tous les registres en début de tâche, pour tous les dépiler en fin de tâche.

Différences avec les sous-programmes : Le programmeur n'est pas maître de la séquence de programme qui va être interrompue. Il lui est donc impossible d'écrire par programme l'adresse de la tâche d'interruption dans le registre PC, comme c'était le cas avec les ruptures de séquence logicielles. Cette adresse doit être prise ailleurs, par une action matérielle déclenchée par l'activation de la ligne d'interruption. Il existe dans tout microcontrôleur une zone réservée de mémoire non volatile (adresses d'interruption) qui contient les adresses des tâches associées (vecteurs d'interruption) aux différentes lignes de demande. Sur le HCS12, les vecteurs d'interruption sont notés en en haut de mémoire A titre d'exemple, aux trois lignes citées plus haut sont associés les adresses :

IRQ→$FFF2,$FFF3
XIRQ→$FFF4,$FFF5
RESET→$FFFE,$FFFF

La plupart des interruptions sont masquables, c'est à dire qu'on peut programmer le microcontrôleur pour tenir ou non compte de la demande d'interruption. Le bit I du CCR masque l'interruption IRQ ainsi que toutes les interruptions issues des périphériques internes. Le bit X du CCR masque l'interruption XIRQ.

Fonctionnement d'une interruption

Prenons par exemple l'interruption IRQ. Le CPU est en train d'exécuter un programme, et la ligne IRQ passe au niveau bas. Si elle n'est pas masquée, elle positionne un drapeau interne, que le processeur consulte entre chaque exécution d'instruction. Une fois terminée l'exécution de l'instruction en cours, le CPU reconnaît une demande d'interruption. Il exécute alors un empilement de tous les registres, y compris le PC qui permet le retour au programme en fin de tâche. Sachant que la demande vient de IRQ, il va ensuite lire dans l'adresse $FFF2,$FFF3 le vecteur d'interruption de IRQ qu'il recopie dans le compteur de programme PC. La tâche d'interruption commence alors. Elle fonctionne comme un sous-programme normal, et se termine par l'instruction **`RTI`**. Comme **`RTS`**, elle permet le retour au programme courant, mais en plus, elle dépile tous les registres empilés au moment de l'appel.

Une tâche d'interruption présente cependant quelques particularités importantes qui la distinguent d'un sous-programme :

- Elle doit prévenir le dispositif demandeur qu'elle a bien reconnu la demande, et faire en sorte qu'il cesse de maintenir IRQ au niveau bas. En général, le dispositif demandeur dispose d'un bit d'état qui monte à 1 pour déclencher une demande. Il appartient à la tâche de remettre ce bit à 0 (acquittement de l'interruption).
- Elle ne peut communiquer avec le programme principal que par des variables globales, en aucun cas par les registres qui sont automatiquement sauvegardés.

Les interruptions, une fois installées, s'exécutent de façon transparente pour le programmeur. Elles sont souvent si transparentes qu'elles se font oublier. Le programmeur doit toujours penser qu'elles existent et qu'elles prennent du temps. Il est donc fortement conseillé, lors de l'écriture d'une tâche d'interruption, de respecter les règles suivantes :

- Une interruption ne doit jamais être bloquante (attente d'un événement)
- Une interruption doit durer le moins longtemps possible (traitement minimum des données dans la tâche), sinon on court le risque de ralentir considérablement le programme principal, voire même de le bloquer si les demandes sont rapprochées.

La notion d'interruption a rencontré en informatique autant de succès que celle de pile, et elle s'y est montrée tout aussi féconde. Sa fécondité a même été si grande qu'il existe des 'interruptions logicielles', autrement dit des interruptions déclenchées par une instruction logicielle, ou plus précisément un sous programme ! Toutes les routines des BIOS et des OS sur les ordinateurs compatibles PC sont ainsi structurées en interruptions logicielles.

Les interruptions des périphériques internes du microcontrôleur

La plupart des modules internes du microcontrôleur (ports, timer, liaisons série SCI et SPI, convertisseurs, etc) peuvent formuler des demandes d'interruption au MCU. Ces demandes utilisent la ligne IRQ interne (reliée physiquement à la patte *IRQ*). Comme chaque module comporte plusieurs sources possibles d'interruption, on arrive à 50 sources possibles de demandes d'interruption internes sur la ligne IRQ. Pour ne pas perdre de temps en début de tâche IRQ pour identifier le demandeur, le microcontrôleur réserve un vecteur à chacune de ces sources.

Au niveau du module, tout événement demandeur est signalé par la montée d'un indicateur. A chaque indicateur est associé un bit qui valide (1) ou masque (0) une demande d'interruption sur montée du bit d'état. Comme les bits d'état sont en général regroupés dans un registre d'état, à chaque registre d'état est associé un registre de validation de demande d'interruption. Quelques exemples :

- Module timer : L'interruption du canal **TCx** sur montée du bit CxF est validée si CxI = 1

$004C	**C7I**	**C6I**	**C5I**	**C4I**	**C3I**	**C2I**	**C1I**	**C0I**	**TIE**
$004E	**C7F**	**C6F**	**C5F**	**C4F**	**C3F**	**C2F**	**C1F**	**C0F**	**TFLG1**

- : L'interruption sur débordement timer (TOF) est validée si TOI = 1

$004D	**TOI**	0	0	0	TCRE	PR2	PR1	PR0	**TSCR2**
$004F	**TOF**	0	0	0	0	0	0	0	**TFLG2**

- Module ATD0 : L'interruption sur séquence complète (ASCIF) est validée si ASCIE = 1

$0082	ADPU	AFFC	AWAI	ETRIGLE	ETRIGP	ETRIGE	**ASCIE**	**ASCIF**	**ATD0CTL2**

Module SCI0 : l'interruption sur donnée présente (RDRF) est validée si RIE = 1

$00CB	**TIE**	**TCIE**	**RIE**	**ILIE**	TE	RE	RWU	SBK	**SCI0CR2**
$00CC	**TDRE**	**TC**	**RDRF**	**IDLE**	OR	NF	FE	PF	**SCI0SR1**

Module SPI0 : L'interruption sur montée de SPIF est validée si SPIE = 1

$00D8	**SPIE**	SPE	**SPTIE**	MSTR	CPOL	CPHA	SSOE	LSBFE	**SPICR1**
$00DB	**SPIF**	0	**SPTEF**	MODF	0	0	0	0	**SPISR**

Module PIM (portH) Un front actif sur PHx (PIFHx = 1) déclenche une inter si PIEHx = 1

$0266	**PIEH7**	**PIEH6**	**PIEH5**	**PIEH4**	**PIEH3**	**PIEH2**	**PIEH1**	**PIEH0**	**PIEH**
$0267	**PIFH7**	**PIFH6**	**PIFH5**	**PIFH4**	**PIFH3**	**PIFH2**	**PIFH1**	**PIFH0**	**PIFH**

Ecriture d'une tâche d'interruption

La plupart du temps, une tâche d'interruption évite une attente qui bloque le programme dans une boucle de scrutation. Une bonne façon de procéder est d'écrire d'abord le programme (ou le sous-programme) de scrutation et d'en extraire la tâche d'interruption, sa validation, et la modification éventuelle des passages de paramètres.

Exemple : On veut équiper le microcontrôleur d'un compteur du nombre de tours du timer. On va utiliser l'indicateur TOF (**T**imer **O**verflow **F**lag) du registre **TFLG2** qui passe à 1 sur débordement du timer (passage de $FFFF à 0). Le corps du programme est une boucle de scrutation de TFLG2 qui attend la montée de TOF. Dès qu'elle s'est produite, le programme fait redescendre l'indicateur pour pouvoir attendre le prochain débordement et incrémente le compteur, puis boucle indéfiniment.

```
* Initialisations
        clr     cpt     ;Initialisation à 0
        clr     cpt+1   ;du compteur 16 bits
        ldaa    #$80    ;Lancement du compteur
        staa    TSCR1   ;TCNT du timer
* Boucle principale
recom   ldaa    #$80    ;On veut tester le bit 7
att     bita    TFLG2   ;Attente de sa montée
        beq     att     ;(bit 7 = TOF)

        staa    TFLG2   ;Redescente de TOF
        ldx     cpt     ;|
        inx             ;|Incrémentation du compteur
        stx     cpt     ;|
        bra     recom   ;et bouclage
```

Il s'agit de transformer ce programme pour incrémenter le compteur par interruptions. L'utilisation de l'interruption supprime l'attente, c'est à dire les trois lignes à partir de recom. La tâche d'interruption se compose donc des lignes restantes de la boucle, suivies d'un rti.

```
avance  ldaa    #$80    ;|
        staa    TFLG2   ;|Redescente de TOF
        ldx     cpt     ;|
        inx             ;|Incrémentation du compteur
        stx     cpt     ;|
        rti
```

Pour qu'elle puisse s'effectuer, il faut que :

- le vecteur d'interruption de débordement contienne l'adresse de la tâche avance,
- la montée de TOF entraîne une demande d'interruption du timer (bit TOI = 1),
- l'interruption soit prise en compte par le MCU (bit I à 0).

Les trois conditions sont réalisées dans le programme ci-dessous :

```
sei             ;Masquage des IRQ
ldaa    #$80    ;Validation de l'inter
staa    TSCR2   ;au niveau du timer
ldx     #avance ;Installation de avance comme
stx     $FFDE   ;tâche de débordement timer
cli             ;Démasquage des IRQ
```

En outre, il faut maintenir les initialisations du programme précédent.

Particularités du traitement des interruptions sous D-Bug12

Lorsqu'on met un microcontrôleur sous tension, il va directement exécuter la tâche de RESET, dont il trouve le vecteur d'interruption en $FFFE, $FFFF. Ces deux adresses doivent impérativement correspondre à une mémoire permanente (ROM ou EPROM ou encore Flash comme c'est le cas sur la carte de TP). Pour cette raison, les adresses hautes sont constituées de mémoire permanente et la ligne

```
stx   $FFDC ;tâche de débordement timer
```

est inefficace puisqu'on ne peut pas écrire à l'adresse indiquée.

Le moniteur D-Bug12 utilisé propose une redirection vers des adresses d'interruption en RAM, de manière à permettre à un utilisateur de créer ses propres vecteurs d'interruption. Il n'est pas nécessaire d'entrer dans tous les méandres de la redirection. On peut sans inconvénient majeur considérer que les adresses d'interruption occupent l'intervalle de $3E0C à $3E7F au lieu de l'intervalle de $FF8C à $FFFF. Pour que le programme d'interruption fonctionne, il suffit de remplacer la ligne " `stx $FFDE` " par la ligne " `stx $3E5E` "
Une autre particularité du moniteur D-Bug12 est qu'il utilise pour sa communication avec le microcontrôleur le module de liaison série SCI0, programmé en interruptions en entrée et en sortie. Il interdit donc l'accès aux interruptions de ce module.

Conclusions

On a vu jusqu'à présent quelques considérations techniques pour écrire correctement et initialiser une tâche d'interruption. Il est parfois plus délicat de décider de l'opportunité de traiter un problème avec ou sans interruption. Quelques principes de base peuvent permettre d'éclairer les choix :

- Une interruption ne doit jamais être bloquante (par exemple faire appel à une entrée clavier)

- Une interruption doit durer le moins longtemps possible. Eviter d'y inclure des sous-programmes longs et lents (affichages notamment sur SCI)

- Une interruption est faite pour se faire oublier. Ne pas oublier tout de même qu'elle prend du temps et qu'elle risque d'accaparer le temps du CPU si on l'appelle trop souvent.

- Lorsqu'on cherche une vitesse d'exécution maximum, ne pas oublier qu'une tâche d'interruption empile, puis dépile tous les registres et qu'il vaut mieux souvent travailler en scrutation.

COURS n°8

Le Timer

Introduction

On peut répartir les fonctions du timer en 3 catégories principales :

- Introduction de délais, ou temporisations,
- Répétition périodique d'actions à intervalles précis,
- Mesure de la durée entre des actions.

Dans la mesure où on connaît exactement la période du cycle de l'horloge E qui cadence le processeur, ainsi que le nombre de cycles utilisés par chaque instruction, l'écriture d'une boucle de programme de durée précise ne pose (en principe) aucun problème. Par exemple, avec une horloge E à 24 MHz, le sous-programme suivant donne une tempo de 1/100^{e} s :

[04]	34		**JSR**	TEMPO	Appel du sous programme

[02]	3C	TEMPO	**PSHX**		Sauvegarde du registre IX
[02]	CE EA5C		**LDX**	#59996	Nombre de boucles d'attente
[01]	09	ATTEN	**DEX**		Décrémentation de IX
[03]	26 FD		**BNE**	ATTEN	jusqu'à arriver à 0
[03]	30		**PULX**		Récupération du registre IX
[05]	3D		**RTS**		Retour du sous programme

La durée de la boucle est de 4 cycles d'horloge. La boucle est effectuée 59996 fois, ce qui représente 4×59996 = 239984 cycles. En plus, le sous programme comprend des instructions exécutées une seule fois, qui durent 2(PSHX) + 2(LDX) + 3(PULX) + 5(RTS) = 12 cycles. Si on inclut la durée de l'appel (4), on arrive à un total de 240000 cycles, soit un centième de seconde exactement.

La question se complique si on veut effectuer une action périodique de période précise. En effet, il faut décompter de la temporisation la durée des instructions de l'action. S'il s'agit d'une action séquentielle, le problème est facilement soluble, mais si elle présente des aiguillages, il faut évaluer la durée de chaque chemin possible pour calculer la temporisation à lui ajouter. Cela donne des logiciels très lourds à mettre au point, et surtout pratiquement impossibles à modifier.

Si on veut mesurer logiciellement la durée écoulée entre deux événements, le logiciel doit contrôler deux actions simultanément : l'avance d'un compteur chronomètre et la surveillance de l'occurrence de l'événement. Par exemple, pour mesurer la largeur d'un créneau appliqué à l'entrée PA_0 du port A, on peut utiliser le programme suivant, où X sert de compteur :

[02]	CE 0000		**LDX**	#0	Initialisation du compteur (X)
[01]	86 01		**LDAA**	#1	
[03]	95 00	ATTEN	**BITA**	PORTA	Attente du passage à 1
[03]	27 FC		**BEQ**	ATTEN	de la ligne 0 du port A
[01]	08	ATTE2	**INX**		Incrémentation du compteur
[03]	95 00		**BITA**	PORTA	Attente du passage à 1
[03]	26 FB		**BNE**	ATTE2	de la ligne 0 du port A
[01]	09		**DEX**		Décrémentation de X (on a compté 1 de trop)

Le compteur IX est incrémenté au sein de la boucle d'attente du second front, dont la durée est de 1 + 3 + 3 = 7 cycles machine. Le résultat de la mesure est 7×X cycles, avec une incertitude de 7 cycles. Cette incertitude est inévitable du fait de la boucle d'attente ATTE2. S'il s'agit d'un test plus compliqué, la gestion commune devient vite très pénible.

Ces exemples donnent une idée des problèmes posés par les temporisations logicielles. Dans les cas cités, c'est difficile mais pas insurmontable. Mais le problème devient impossible pour deux raisons principales : l'évolution des processeurs dont les vitesses d'horloge s'accélèrent, supprimant ainsi toute portabilité des logiciels, et le fonctionnement en parallèle avec des interruptions dont on ignore la durée et la fréquence puisqu'on n'en connaît souvent même pas les sources. Il devient alors absolument nécessaire de disposer d'une horloge autonome, qui gère l'incrémentation de son (ou ses) compteurs indépendamment du logiciel du CPU. On appelle ce module périphérique timer (ou compteur de temps).

Présentation

Un timer est un composant qui incrémente ou décrémente un registre à chaque front appliqué sur sa ligne d'entrée CLK (horloge). Si ce signal est périodique (horloge du microprocesseur ou oscillateur indépendant), le registre du timer fonctionne lui même comme une horloge dans laquelle le programme peut aller lire l'heure à tout moment. Si les fronts de la ligne d'entrée ne sont pas périodiques, le timer fonctionne alors en compteur d'événements (compteur Geiger par exemple).

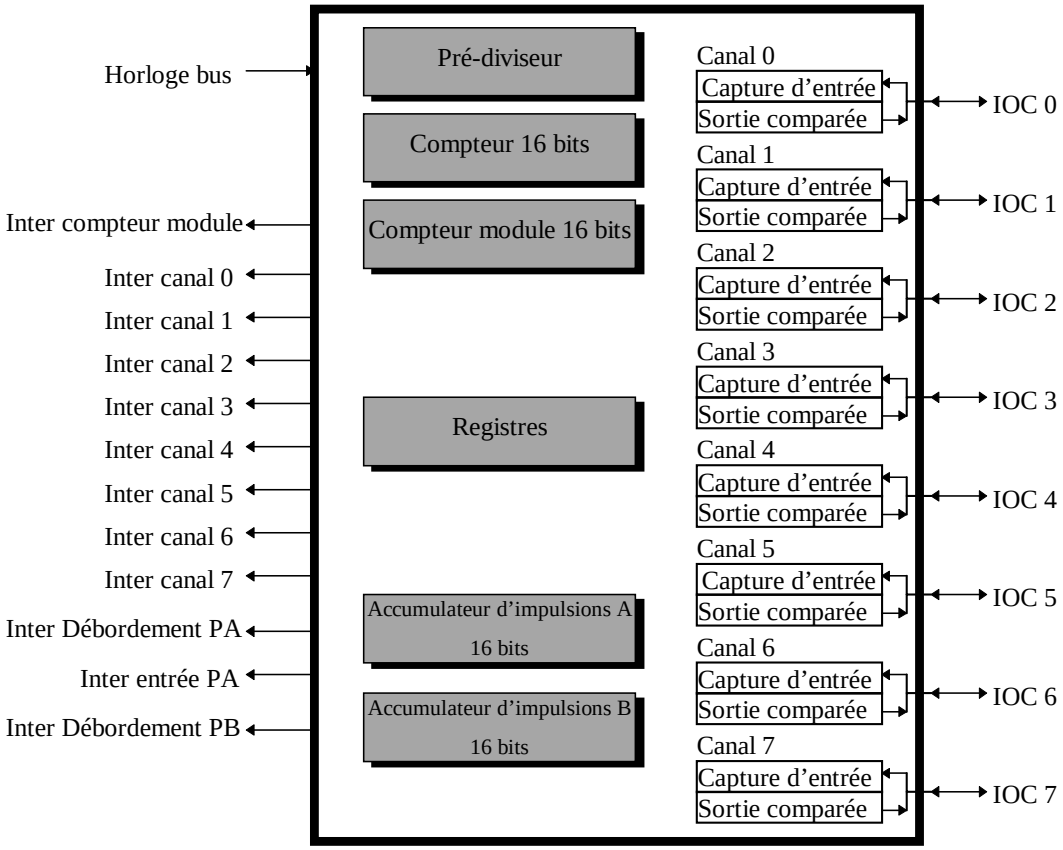
Les composants timers usuels sont le 8254 dans la famille INTEL et le 6840 dans la famille MOTOROLA. Tous deux réunissent sur un même composant 3 compteurs de 16 bits, qu'on peut lancer ou arrêter, ainsi que divers modes de fonctionnement qui leur donnent une grande souplesse d'utilisation : possibilité de commander un timer par un autre, fonctionnement en cascade, etc. Le timer dont on dispose est assez différent et correspond au bloc diagramme ci-dessous. On y distingue trois parties :

- Un compteur principal de 16 bits précédé d'un pré-diviseur qui permet d'avoir sous la main une base de temps permanente. Le système de huit registres associés de capture d'entrée ou de sortie comparée permet une utilisation optimum de cette ressource.

- Quatre accumulateurs d'impulsions de 8 bits, cascadables en deux accumulateurs de 16 bits fournissent un accessoire pratique pour les comptages d'événements et les chronométrages. De nombreux gadgets équipent cette option pour l'adapter au mieux à une grande variété de problèmes concrets.

- Un décompteur de 16 bits, muni lui aussi d'un pré-diviseur indépendant, permet la génération d'une autre base de temps. Bien qu'il ait été prévu pour faciliter l'utilisation des accumulateurs, on peut parfaitement l'utiliser comme un timer indépendant.

L'exploitation de tous les accessoires du timer nécessite de nombreux registres. Motorola en a prévu $40 (de $40 à $7F). On va décrire maintenant les principales fonctions du timer avec les registres associés.



Registre de comptage du timer

$0044	TCNT15	TCNT14	TCNT13	TCNT12	TCNT11	TCNT10	TCNT9	TCNT8	**TCNT**
$0045	TCNT7	TCNT6	TCNT5	TCNT4	TCNT3	TCNT2	TCNT1	TCNT0	

Situé aux adresses $44 et $45, **TCNT** est le compteur de base du timer. En lecture seule, il s'incrémente à chaque impulsion d'un pré-diviseur qui divise l'horloge E du micro (24 MHz sur la carte de TP) par 2^N, N allant de 0 à 7.

Deux commandes agissent sur ce compteur :

Le bit TEN (timer enable), bit 7 du registre **TSCR1** qui l'active (à 1) ou l'arrête (à 0).

$0046	TEN	TSWAI	TSFRZ	TFFCA	0	0	0	0	**TSCR1**

Les trois bits poids faibles du registre **TSCR2** qui définissent la valeur de N.

$004D	TOI	0	0	0	TCRE	PR2	PR1	PR0	**TSCR2**

Quand il arrive à FFFF, il repasse à 0, en mettant à 1 l'indicateur TOF (Time OverFlow) du registre d'état **TFLG2**.

$004F	TOF	0	0	0	0	0	0	0	**TFLG2**

On peut distinguer deux types d'utilisation du compteur :

- Réaliser des temporisations, pour répéter périodiquement des actions, logicielles ou matérielles. Le principe est simple : on lit le compteur, on y ajoute la durée de temporisation et on attend que le compteur atteigne la valeur calculée pour exécuter l'action, et on recommence le processus. Ce type de fonctionnement (déclenchement d'un 'top' ou d'une autre action à un instant précis) est appelé 'Sortie Comparée' (Output Compare).

-.Mesurer la durée entre plusieurs 'tops' matériels. Il s'agit de saisir au vol la valeur du compteur immédiatement après chaque événement pour faire les calculs ensuite. Ce type de fonctionnement est appelé 'Capture d'Entrée' (Input Capture).

Pour réaliser ces deux types d'action le timer dispose de huit registres de 16 bits **TC0** à **TC7**, qui peuvent à la demande servir à des captures d'entrée ou des sorties comparées, suivant le registre **TIOS** : bit IOSx à 1 : TCx en sortie comparée. bit IOSx à 0 : TCx en capture d'entrée.

$0040	IOS7	IOS6	IOS5	IOS4	IOS3	IOS2	IOS1	IOS0	**TIOS**

Les sorties comparées (OCx : Output Compare N°x)

Il s'agit de déclencher à un instant donné une action logicielle ou matérielle ('top' sur une ligne physique). On pourrait imaginer une boucle logicielle d'attente de l'égalité du registre de comptage **TCNT** et d'une valeur donnée. Par exemple, si la valeur est contenue dans le registre double D, on pourrait écrire la séquence suivante :

```
RECOM   CPD   TCNT
        BNE   RECOM
```

La durée de la boucle est de 3 + 3 = 6 cycles machine. Si le timer est incrémenté à chaque cycle, l'égalité cherchée risque fort d'avoir lieu entre deux comparaisons successives, et donc de n'être pas détectée. La solution de faire un test d'inégalité est elle aussi malcommode puisque le timer reboucle sur lui-même. Le timer évite ces ennuis en utilisant un comparateur matériel, qui positionne l'indicateur CxF quand le compteur passe par la valeur du registre **TCx** (utilisé comme sortie comparée). Ces 8 indicateurs composent le registre **TFLG1**.

$004E	C7F	C6F	C5F	C4F	C3F	C2F	C1F	C0F	**TFLG1**

La montée du bit CxF entraîne une interruption si le bit correspondant CxI de **TIE** est à 1

$004C	C7I	C6I	C5I	C4I	C3I	C2I	C1I	C0I	**TIE**

Dans le cas précédent, l'événement de sortie comparée est logiciel : montée d'un indicateur. Le timer offre en plus la possibilité de déclencher une action matérielle ('top' sur une ligne) sur une sortie comparée. Les lignes utilisées sont les lignes du port T, *PTx* étant associée au registre **TCx**. Le choix de la forme du top se fait dans les registres **TCTL1** et **TCTL2**.

$0048	OM7	OL7	OM6	OL6	OM5	OL5	OM4	OL4	**TCTL1**
$0049	OM3	OL3	OM2	OL2	OM1	OL1	OM0	OL0	**TCTL2**

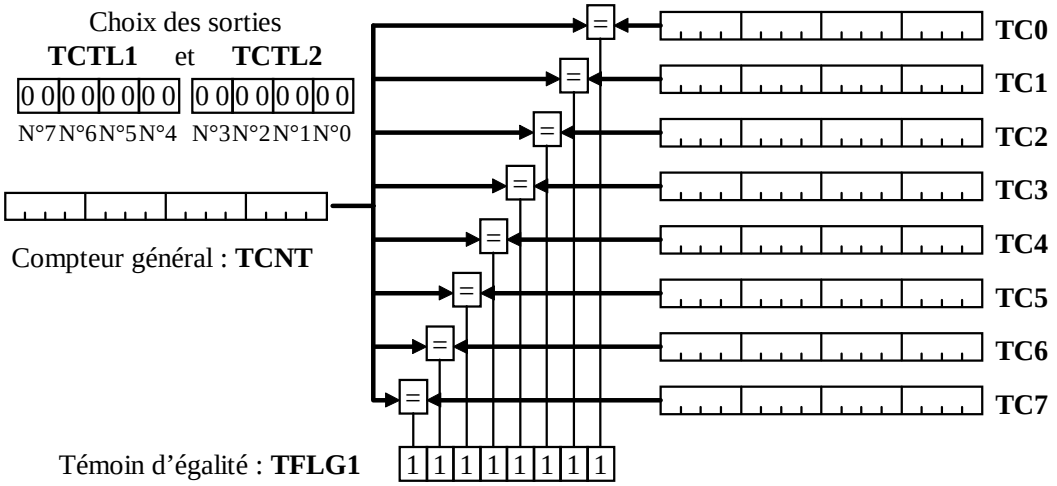
Exemple : soit à répéter l'action définie par le sous-programme ACTION tous les millièmes de seconde. L'événement OC0, signalé par la montée à 1 du bit C0F du registre d'état, signifie que le compteur est passé par la valeur affichée dans le registre **TC0**. Le programme attend cet événement en scrutant le bit C0F. Quand il s'est produit, on exécute l'action, puis on prépare les registres du timer pour la prochaine scrutation, c'est à dire en ajoutant la période à TC0 et en remettant l'indicateur C0F à 0. On a la séquence :

```
86 40          LDAA   #$40
95 4E   RECOM  BITA   TFLG1   Attente de l'événement OC0
27 FC          BEQ    RECOM   (TCNT = TC0)
16 xxxx        JSR    ACTION  Exécution de l'action
DC 50          LDD    TC0     Préparation de TC0
C3 5DC0        ADDD   #24000  pour le prochain coup
5C 50          STD    TC0     (TC0 = TC0 + 24000)
86 40          LDAA   #$40    Remise à zéro de
5A 4E          STAA   TFLG1   l'indicateur C0F
20 EC          BRA    RECOM   Bouclage
```

Le programme est indépendant de la durée du sous-programme ACTION, pourvu qu'elle soit plus courte que la période. La boucle de scrutation dure 6 cycles, ce qui donne une incertitude de 6 cycles machine sur l'instant d'exécution du sous-programme ACTION. Mais les OC0 sont toujours exactement séparés par une période. Il n'y a pas accumulation d'incertitude.

En résumé :

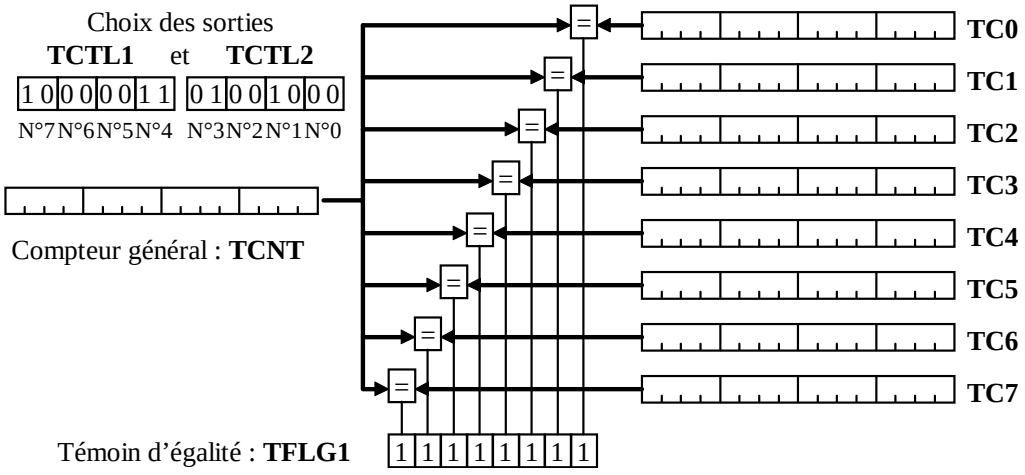
A / Aucune sortie activée. Les lignes physiques que le timer partage avec le port T restent la « propriété » du port T



B/ Quatre sorties activées

Quand TCNT = TC7, la patte associée (*PT7*) passe au niveau bas
Quand TCNT = TC4, la patte associée (*PT4*) passe au niveau haut
Quand TCNT = TC3, la patte associée (*PT3*) change de niveau
Quand TCNT = TC1, la patte associée (*PT1*) passe au niveau bas

Les pattes *PT6*, *PT5*, *PT2* et *PT0* restent des pattes du port T



Les captures d'entrée (ICx : Input Capture N°x)

Il s'agit de récupérer l'instant où se produit une action matérielle ('top' sur une ligne physique, c'est à dire front montant ou descendant). On peut imaginer une boucle logicielle d'attente du front. Par exemple, pour attendre un niveau haut sur la ligne *PH1* du port H, on pourrait écrire la séquence suivante :

```
        LDAA # 2
RECOM   BITA  PORTH
        BEQ   RECOM
```

La durée d'une telle boucle est de 4 + 3 = 7 cycles machine. Elle représente l'incertitude sur la mesure du temps. On peut l'éviter en reliant physiquement la ligne au mécanisme de capture.

Chaque capture utilise une ligne physique et un registre 16 bits. Ces lignes et ces registres sont les mêmes que pour les sorties comparées, à savoir les lignes du port T, la ligne *PTx* étant associée au registre **TCx**. Le fonctionnement d'une capture est le suivant. L'événement sur la ligne *PTx* déclenche une recopie du registre de comptage **TCNT** dans le registre de capture **TCx** (choisi comme capture d'entrée). Cette capture est signalée logiciellement par la montée du bit CxF du registre d'état **TFLG1.** Cette montée entraîne une interruption si le bit correspondant CxI de **TIE** est à 1.

Le type de front qui définit l'événement actif sur chaque ligne est choisi par deux bits consécutifs dans les registres **TCTL3** et **TCTL4**. Il peut s'agir soit d'un front montant, soit d'un front descendant, soit des deux. Si on n'en choisit aucun, la capture d'entrée est dévalidée et la ligne correspondante du port T recouvre ses droits.

$004A	EDG7B	EDG7A	EDG6B	EDG6A	EDG5B	EDG5A	EDG4B	EDG4A	**TCTL3**
$004B	EDG3B	EDG3A	EDG2B	EDG2A	EDG1B	EDG1A	EDG0B	EDG0A	**TCTL4**

Exemple : la séquence de programme suivante récupère dans X l'instant où s'est produit une transition active de la ligne *PT1*. (La séquence ne définit pas la transition active).

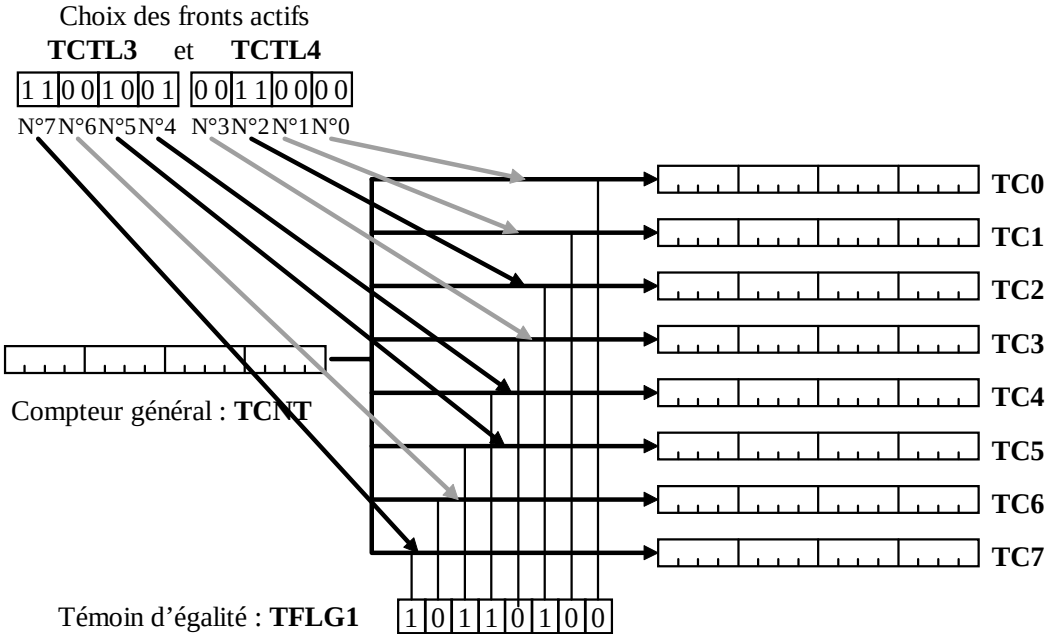
```
86 02          LDAA  #2      Mise à zéro de
5A 4E          STAA  TFLG1   l'indicateur C1F
95 4E  RECOM   BITA  TFLG1   Boucle d'attente de
27 FC          BEQ   RECOM   la transition active
DE 52          LDX   TC1     Récupération du temps dans IX
```

La fonction capture d'entrée a été complétée par des registres de rangement qui permettent plus de souplesse dans les mesures de temps où on doit faire la différence entre deux entrées capturées. Seuls les canaux 0, 1, 2 et 3 sont équipés de ce gadget.

En résumé,

Dans l'exemple qui suit, le contenu des registres **TCTL3** et **TCTL4** indique que seules les pattes *PT7*, *PT5*, *PT4* et *PT2* peuvent déclencher des captures, *PT7* et *PT2* sur front quelconque, *PT5* sur front descendant et *PT4* sur front montant. Les quatre autres pattes restent des pattes d'entrée-sortie du port T

Il s'ensuit que les bits 6, 3, 1 et 0 du registre **TFLG1** restent constamment à 0



Accumulateurs d'impulsions.

Le module timer comporte quatre accumulateurs d'impulsions de 8 bits, associés aux pattes *PT0* à *PT3* du port T. Ces quatre accumulateurs bénéficient aussi de registres de rangement.

$0062	PACNT7/15	PACNT6/14	PACNT5/13	PACNT4/12	PACNT3/11	PACNT2/10	PACNT1/9	PACNT0/8	**PACN3**
$0063	PACNT7	PACNT6	PACNT5	PACNT4	PACNT3	PACNT2	PACNT1	PACNT0	**PACN2**
$0064	PACNT7/15	PACNT6/14	PACNT5/13	PACNT4/12	PACNT3/11	PACNT2/10	PACNT1/9	PACNT0/8	**PACN1**
$0065	PACNT7	PACNT6	PACNT5	PACNT4	PACNT3	PACNT2	PACNT1	PACNT0	**PACN0**

Ces accus sont validés par un bit du registre **ICPAR**

$0068	0	0	0	0	PA3EN	PA2EN	PA1EN	PA0EN	**ICPAR**

Le type d'impulsion déclenchante est déterminé par les deux registres **TCTL3** et **TCTL4** déjà utilisés pour les captures d'entrée.

Les deux accumulateurs N° 0 et 1 peuvent se cascader pour former l'accumulateur 16 bits **PACB**, dont le poids faible est **PACN0** et le poids fort **PACN1**. La patte associée à **PACB** est la patte *PT0*. La notice précise bien comment choisir entre deux accus 8 bits et 1 accu 16 bits.

Les deux accumulateurs N° 2 et 3 peuvent se cascader pour former l'accumulateur 16 bits **PACA**, dont le poids faible est **PACN2** et le poids fort **PACN3**. La patte associée à **PACA** est la patte *PT7*. La notice précise bien comment choisir entre deux accus 8 bits et 1 accu 16 bits.

Tous les accumulateurs (8 bits et 16 bits) fonctionnent en accumulateur d'impulsion, depuis qu'ils sont activés jusqu'à ce qu'ils soient invalidés. L'accu **PACA** dispose en plus d'un autre mode de fonctionnement, le mode chronomètre, où il compte le nombre de tops d'horloge entre deux fronts appliqués. Seule une lecture détaillée du mode d'emploi permettra d'en tirer toutes les possibilités.

COURS n°9

Les autres modules

A : LES CONVERTISSEURS ANALOGIQUES / NUMERIQUES

Généralités

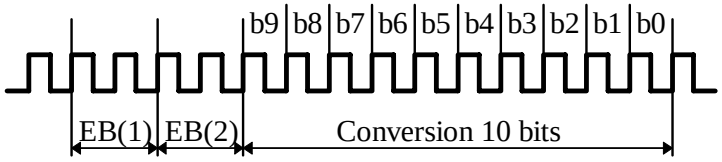
Le microcontrôleur étudié contient deux modules identiques de conversion analogique numérique de type ATD_10B8C. Ces deux modules sont désignés par ATD0 (Analog To Digital N° 0) et ATD1 (Analog To Digital N° 1). On se borne ici à décrire ATD0, ATD1 ne différant que par les pattes physiques et l'adresse de ses registres. Et pour alléger l'écriture, on omettra le 0 pour parler simplement du module ATD.

Il s'agit d'un convertisseur analogique / numérique par approximations successives sur 10 bits, à huit canaux d'entrée multiplexés. Bien que ce ne soit pas une obligation (voir références électriques du module) on peut considérer que la précision de la conversion est de l'ordre d'un bit poids faible, soit 1/1024 pleine échelle. Il s'agit d'une précision respectable si on travaille à pleine échelle (5 mV sur 5V) mais vite faible si on n'utilise qu'une partie de la gamme possible.

Le bloc ATD présente les caractéristiques suivantes :

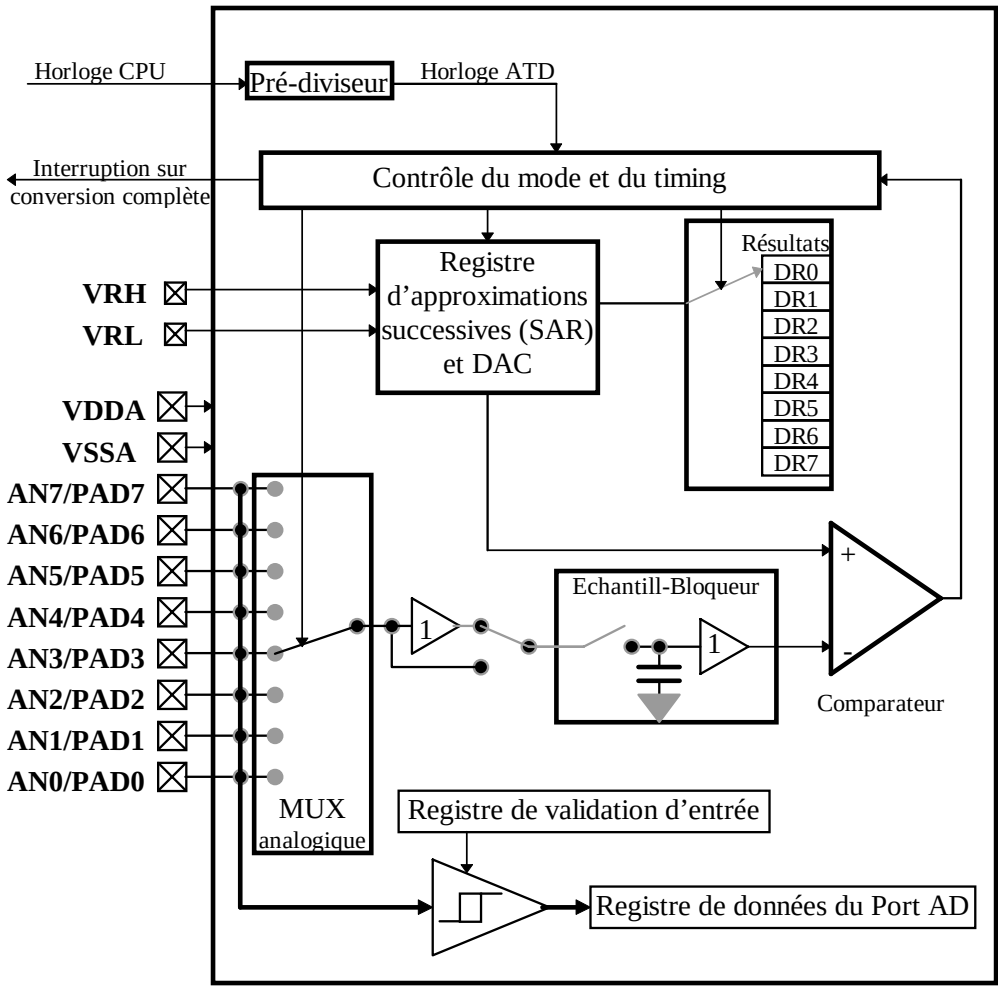
- Choix de la résolution (8 ou 10 bits).
- Choix de la vitesse de conversion.
- Echantillonneur bloqueur. Choix du temps d'échantillonnage.
- Résultats justifiés à gauche ou à droite, signés ou non.
- Possibilité de contrôle extérieur du déclenchement.
- Possibilité de génération d'une interruption sur conversion complète.
- Multiplexeur analogique pour 8 canaux d'entrée analogique.
- Séquences de conversions de 1 à 8.
- Mode de conversion continu.
- Scrutation de canaux multiples.

Bien que programmable, le temps de conversion est réglé par une horloge propre à l'ATD établie par une division de l'horloge CPU. Les valeurs permises pour l'horloge ATD ne dépassent pas 2 MHz. Une conversion sur 10 bits prendra donc 10 cycles, soit 5µs, sans compter le temps d'échantillonnage. Celui-ci, programmable, occupe au moins 2 cycles (première phase) et 2 à 16 cycles (seconde phase), soit un minimum de 4 cycles et donc 2µs.
Au total, la conversion dure un minimum de 14 cycles, soit 7 µs.



Temps de conversion minimum (EB = 2)

Description du fonctionnement



Le convertisseur peut mesurer de 1 à 8 signaux analogiques différents, appliqués aux entrées analogiques *AN0* à *AN7*, c'est à dire aux pattes *PAD0* à *PAD7*. Les entrées analogiques sont multiplexées, les conversions se faisant successivement sur chaque canal.

Une mesure consiste en un échantillonnage / blocage qui fige la tension à mesurer pour la durée de mesure, puis à des comparaisons successives de la valeur bloquée avec les tensions successives (VRH-VRL)/2, (VRH-VRL)/4... (VRH-VRL)/2^N fournies par un DAC intégré, qui construisent le nombre de N bits correspondant à la tension analogique. Le rangement de la mesure se fait dans un des registres 16 bits de résultats de DR0 à DR7.

L'ordonnancement des conversions et de leur rangement est assuré par le bloc de contrôle et de timing, programmable grâce à de nombreux registres. Le principe de base est de procéder par séquence, c'est à dire un enchaînement aussi rapproché que possible de 1 à 8 mesures sur le même canal ou des canaux différents, les résultats d'une même séquence étant rangés dans les registres de résultats.

Principaux registres concernés

Pour fonctionner, l'ATD doit être alimenté. C'est le rôle du bit ADPU du registre **ATDCTL2**. On trouve également dans ce registre des bits relatifs au déclenchement extérieur.

$0082	ADPU	AFFC	AWAI	ETRIGLE	ETRIGP	ETRIGE	ASCIE	ASCIF	**ATD0CTL2**

Le choix du timing de la conversion dépend du registre **ATDCTL4**, qui définit la fréquence de l'horloge ATD et le nombre de cycles d'échantillonnage blocage.

$0084	SRES8	SMP1	SMP0	PRS4	PRS3	PRS2	PRS1	PRS0	**ATD0CTL4**

Le choix du nombre d'éléments de la séquence (1 à 8) se fait dans le registre **ATDCL3**. Les éléments sont rangés dans les registres de résultats dans l'ordre d'exécution de la séquence, sauf si on est en mode FIFO où les résultats s'affichent sans cesse à la suite en bouclant.

$0083	0	S8C	S4C	S2C	S1C	FIFO	FRZ1	FRZ0	**ATD0CTL3**

Le registre **ATDCTL5** est essentiel car c'est une écriture dans ce registre qui déclenche une nouvelle séquence de conversion, en arrêtant au besoin celle en cours. Il fixe le N° du canal à convertir en premier. En mode MULT, la séquence se poursuit par celle des canaux suivants. En mode simple, la séquence répète la mesure sur le même segment. Le mode SCAN relance une séquence dès la fin de la précédente, sinon l'ATD attend une nouvelle écriture pour lancer une nouvelle séquence. Ce registre fixe en outre le format de sortie (cadrage à droite ou à gauche, valeur signée ou non).

$0085	DJM	DSGN	SCAN	MULT	0	CC	CB	CA	**ATD0CTL5**

L'ATD dispose de deux registres d'état. **ATDSTAT0** et **ATDSTAT1** Le premier est fondamental par son bit 7 (SCF) qui indique la fin d'une séquence de conversion. Les autres bits de ce registre ainsi que ceux de l'autre registre fournissent des informations plus fines sur le déroulement d'une séquence.

$0086	SCF	0	ETORF	FIFOR	0	CC2	CC1	CC0	**ATDSTAT0**

Sur chacun des 8 registres de rangement 16 bits on doit ranger des nombres de 10 bits. Il y a possibilité de différents cadrages. On trouvera le détail dans la notice complète.

Pour être complet il faut rappeler que les lignes d'entrée des canaux analogiques peuvent aussi servir de lignes d'entrées logiques. Il y a des registres prévus pour valider ou non cette possibilité et bien sûr il existe le registre de port PAD dont les bits traduisent le cas échéant les niveaux logiques des pattes *PAD0* à *PAD7*.

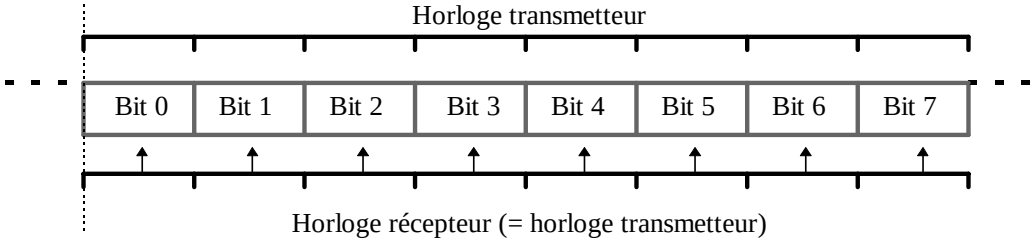
B : LES LIAISONS SERIE SCI ET SPI

Comparaison entre transmission série synchrone et asynchrone

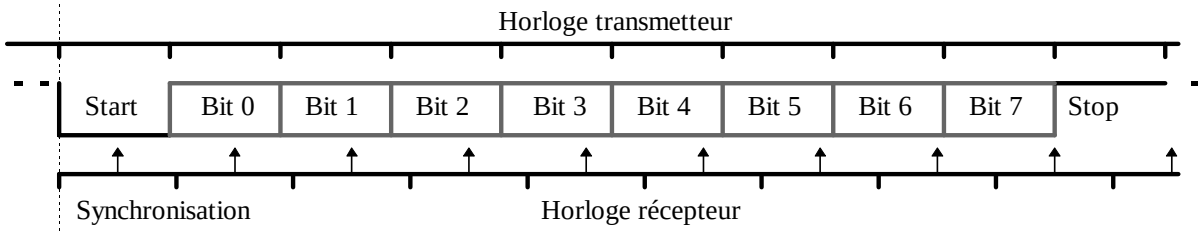
Les liaisons série ont en commun le transfert d'octets bit à bit sur une même ligne. Un bit est représenté par un niveau haut ou bas de la ligne pendant un certain temps défini par le nombre de **bauds** (bits par seconde) et réglé par une horloge. On distingue deux types de transmissions suivant les horloges utilisées pour découper un octet et le recomposer.

Dans les deux cas, chaque bit est émis entre deux tops d'horloge* de l'émetteur, pour être reconstitué à la réception. Le récepteur se place juste entre deux tops de son horloge (flèches) pour tomber au milieu de chaque bit et reconstituer l'octet. Cela implique bien sûr que les horloges de l'émetteur et du récepteur soient synchronisées.

Dans le cas d'une liaison série synchrone, les deux partenaires utilisent la même horloge (celle du maître, qui peut être récepteur ou transmetteur). La synchronisation est toujours assurée, mais le transfert exige un fil supplémentaire pour transmettre l'horloge..



Dans le cas d'une liaison série asynchrone le récepteur utilise son horloge qui a <u>à peu près</u> la même fréquence que l'émetteur et il la synchronise au début de chaque octet, à la réception du front descendant du bit start sur la ligne de donnée. Si les fréquences sont peu différentes, la flèche de réception tombe bien sur le bit, mais pas tout à fait au milieu. Si elles diffèrent de plus de 5% environ, la reconstitution peut omettre un bit ou compter deux fois le même.



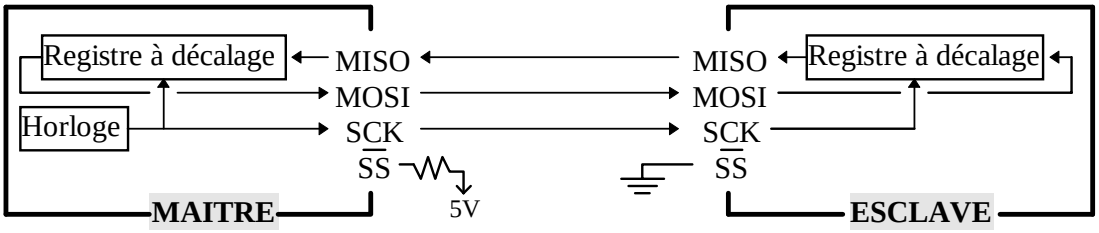
Le MC9S12DG256B dispose de trois modules de liaison série synchrone : SPI0, SPI1 et SPI2 et de deux modules de liaison série asynchrone : SCI0 et SCI1. Les trois modules SPI sont rigoureusement identiques de même que les deux modules SCI. Ils ne diffèrent que par les pattes physiques qui les relient à l'extérieur.

** Pour faciliter la lecture des schémas, on n'a fait figurer que les 'tops' d'horloge. Ce signal est en fait un signal carré, et les tops représentent un type de front (montant ou descendant).*

Liaison série synchrone SPI (Serial Peripheral Interchange)

Le microcontrôleur dispose de trois modules SPI rigoureusement semblables. On décrit SPI0, SPI1 et SPI2 ne différant que par les pattes physiques et l'adresse des registres. Et pour alléger l'écriture, on omettra le 0 pour parler simplement du module SPI.

Le fonctionnement de la liaison SPI est a priori dissymétrique du fait de l'utilisation d'une horloge commune. Le générateur de l'horloge commune est appelé maître et l'autre partenaire de l'échange est appelé esclave.



En mode maître ou en mode esclave, la liaison SPI utilise quatre pattes du microcontrôleur. Le SPI0 les emprunte au port S : *PTS4 = MISO* (**M**aster **I**n **S**lave **O**ut), *PTS5 = MOSI* (**M**aster **O**ut **S**lave **I**n), *PTS6 = CLK* (horloge) et *PTS7 = SS* (**S**lave **S**elect). La figure ci-dessus précise la structure de la liaison synchrone et les principaux registres afférents. La détection du signal se fait avec l'horloge du maître, et la communication des données se fait par échange systématique des contenus des registres **SPIDR** de données du maître et de l'esclave.

$00DD	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0	**SPIDR**

Pour le contrôle des échanges le SPI dispose
- d'un registre de bauds **SPIBR** fixant la vitesse de transmission :

$00DA	0	SPPR2	SPPR1	SPPR0	0	SPR2	SPR1	SPR0	**SPIBR**

- d'un registre d'état **SPISR** comportant trois indicateurs :

$00DB	SPIF	0	SPTEF	MODF	0	0	0	0	**SPISR**

- de 2 registres de contrôle pour choisir parmi les différentes options proposées (cf notice) :

$00D8	SPIE	SPE	SPTIE	MSTR	CPOL	CPHA	SSOE	LSBFE	**SPICR1**
$00D9	0	0	0	MODFEN	BIDIROE	0	SPISWAI	SPCO	**SPICR2**

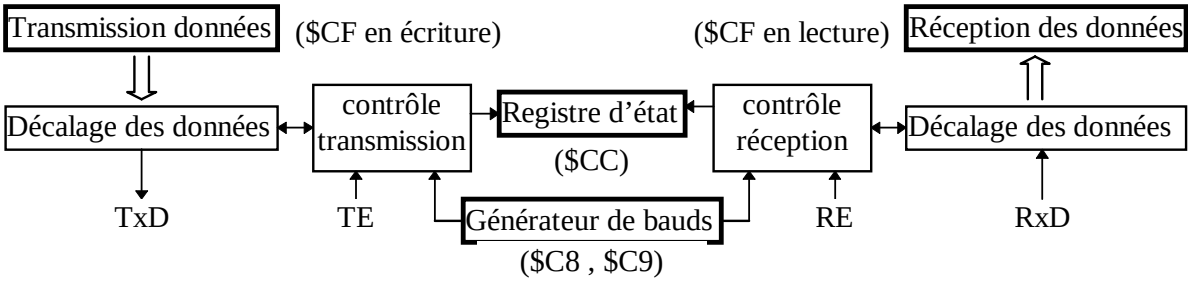
On n'entre pas plus dans le détail de la liaison SPI, dont l'étude et la manipulation se font dans le cadre du cours d'informatique industrielle de deuxième année.

Liaison série asynchrone SCI (Serial Communication Interface)

Le microcontrôleur dispose de deux modules SCI rigoureusement semblables. On décrit SCI0, SCI1 ne différant que par les pattes physiques et l'adresse des registres. Et pour alléger l'écriture, on omettra le 0 pour parler simplement du module SCI.

On a déjà présenté ce type de liaison dans le début du cours. On y a vu notamment que deux horloges différentes ne peuvent pas rester synchronisées, ce qui impose l'ajout d'un bit de start et d'un bit de stop pour une synchronisation au début de chaque octet transmis et un contrôle après le dernier bit.

La liaison SCI utilise deux pattes du microcontrôleur. Pour le SCI0, ce sont les pattes du port S : *PTS0 = RxD* (**R**éception de **D**onnées), *PTS1= TxD* (**T**ransmission de **D**onnées). Le SCI1 utilise quant à lui les pattes *PTS2* et *PTS3*. La figure ci-dessous illustre le fonctionnement d'un module SCI. Le module se comporte en fait comme deux modules pratiquement indépendants, un module de réception et un module de transmission.



Le module de transmission comporte principalement un registre à décalage qui assure le découpage et la transmission sur la ligne *TxD* de l'octet qu'il contient. Pour envoyer un octet sur le SCI il suffit de l'écrire dans le registre de transmission du SCI. Il est alors recopié et traité dans le registre à décalage. Bien entendu, des indicateurs permettent de contrôler le bon fonctionnement du système et d'éviter des erreurs (notamment de bourrage).

Le module de réception comporte lui aussi un registre à décalage qui assure la reconstruction de l'octet à partir du signal qui lui arrive par la patte *RxD*. Dès que c'est fait, l'octet est recopié dans le registre de réception, et un indicateur signale son arrivée. Bien entendu, le travail de réception est contrôlé et divers indicateurs signalent les anomalies dans le transfert.

Ces modules ont en commun le registre de bauds **SCIBD** qui permet sur 13 bits de choisir la vitesse de transmission **et** de réception selon la relation :

$$\text{baud} = \text{horloge}/(16 \times \text{SBR}).$$

Avec une horloge micro à 24 MHz, le nombre de bauds peut aller de 183 (SBR = $1FFF) à 1.500.000 bauds (SBR = 1). Sur la carte de TP, le moniteur D-Bug12 initialise SBR à $9C, ce qui donne environ 9615 bauds.

$00C8	0	0	0	SBR12	SBR11	SBR10	SBR9	SBR8	**SCI0BDH**
$00C9	SBR7	SBR6	SBR5	SBR4	SBR3	SBR2	SBR1	SBR0	**SCI0BDL**

Ils ont également en commun l'adresse du registre de donnée **SCIDRL.** Bien qu'il s'agisse de deux registres distincts, on peut leur attribuer la même adresse puisque le registre de réception n'est accessible qu'en lecture et le registre de transmission n'est accessible qu'en écriture.

$00CF	R7/T7	R6/T6	R5/T5	R4/T4	R3/T3	R2/T2	R1/T1	R0/T0	**SCI0DRL**

Dans les deux registres de commande, un seul est important en première utilisation, c'est **SCICR2**. Le bit TE (**T**ransmit **E**nable) établit l'affectation de la patte *PTS1* au module SCI en tant que *TxD* et le bit RE (**R**eceive **E**nable) établit l'affectation de la patte *PTS0* au module SCI en tant que *RxD*. Les quatre bits poids fort servent à valider les demandes d'interruption sur montée des indicateurs du registre d'état.

$00CB	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK	**SCI0CR2**

Le SCI propose en option la possibilité d'échanger un 9^{e} bit de donnée. C'est la fonction du registre **SCIDRH**. Dans un premier temps, on peut l'ignorer.

Le registre d'état du SCISR

$00CC	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF	**SCI0SR1**

Ce registre contient tous les indicateurs qui contrôlent le bon fonctionnement du module, en transmission comme en réception. Deux indicateurs sont fondamentaux :
- TDRE (**T**ransmit **D**ata **R**egister **E**mpty) indiquant qu'on peut envoyer un nouvel octet sur la sortie série, le précédent étant parti
- RDRF (**R**eceive **D**ata **R**egister **F**ull) indiquant que le registre de réception est plein, donc qu'une nouvelle donnée vient d'arriver.

Tous les indicateurs du SCI sont remis à 0 par deux opérations successives : une lecture du registre d'état suivie d'une lecture (RDRF) ou d'une écriture (TDRE) à l'adresse commune des deux registres de donnée. Cette technique de remise à 0 suit de très près le fonctionnement du module et est automatique dans un fonctionnement par scrutation. Elle diffère de la méthode utilisée dans les autres modules, où on remet un bit d'état à 0 en y écrivant un 1.

Exemple de sous programme d'entrée en scrutation :

```
************************LIOCT ************************
* Réception d'un octet sur la liaison série            *
* Attend l'arrivée d'un caractère et le récupère dans A *
* Aucun autre registre modifié sauf CCR, suivant A     *
*******************************************************
LIOCT  LDAA    #$20    ; Attente de la  montée
ATT    BITA    SCISR1  ; |de RDRF, bit 5 du
       BEQ     ATT     ; |registre d'état SCISR
       LDAA    SCIDRL  ; Lecture de la donnée
       RTS
```

Exemple de sous programme de sortie en scrutation :

```
************************ SOROCT ************************
* Emission d'un octet sur la liaison série             *
* Attend que la ligne soit libre et                    *
* émet l'octet contenu dans A                          *
* Aucun autre registre modifié sauf CCR, suivant A     *
*******************************************************
SOROCT TST     SCISR   ; Attente de la montée
       BPL     SOROCT  ; de TDRE, bit 7 de SCISR
       STAA    SCIDRL  ; Envoi de la donnée
       RTS
```