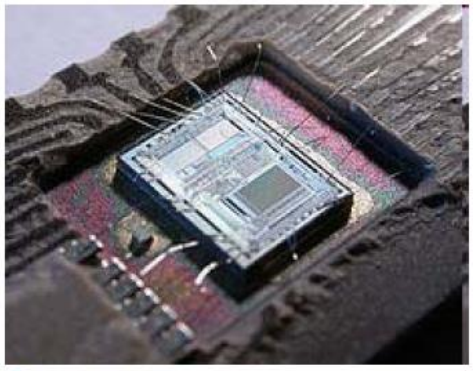
# CIRCUITS INTEGRES PROGRAMMABLES ET MICROPROCESSEURS.


FPGA de Xilinx (modèle Spartan XC3S400) avec 400 000 portes



***Présenté par :***

**Mr. Mazoughou GOEPOGUI**

***Tel: 655 34 42 38 / 669 35 43 10 / 624 05 56 40***

***g-mail: massaleidamagoe2014@gmail.com***

***yahoo/facebook : massaleidamagoe@yahoo.fr***

***site : www.massaleidamagoe2015.net***

# I. INTRODUCTION.

## I.1. Définition.

Un circuit programmable est un assemblage d'opérateurs logiques combinatoires et de bascules dans lequel la fonction réalisée n'est pas fixée lors de la fabrication. Il contient potentiellement la possibilité de réaliser toute une classe de fonctions, plus ou moins large suivant son architecture. La programmation du circuit consiste à définir une fonction parmi toutes celles qui sont potentiellement réalisables.

Comme dans toute réalisation en logique câblée, une fonction logique est définie par les interconnexions entre des opérateurs combinatoires et des bascules, et par les équations des opérateurs combinatoires. Ce qui est programmable dans un circuit concerne donc les interconnexions et les opérateurs combinatoires. Les bascules sont le plus souvent de simples bascules D, ou des bascules configurables en bascules D ou T.

## I.2. Généralité.

L'électronique moderne se tourne de plus en plus vers le numérique qui présente de nombreux avantages sur l'analogique : grande insensibilité aux parasites et aux dérives diverses, modularité et (re)configurabilité, facilité de stockage de l'information etc... Pour s'en convaincre, il n'y a qu'à regarder autour de nous l'explosion de la microinformatique, qui s'est même implantée dans les ménages. Un nombre de plus en plus grand de machines (télévision, voiture, machine à laver, etc.) utilisent de l'électronique numérique.

Jusqu'à présent, l'apprentissage de la logique se faisait à travers la découverte des fonctions logiques élémentaires contenues dans les circuits intégrés des familles TTL ou CMOS. Les expérimentations se limitaient aux fonctions proposées par les fabricants de ces circuits. La conception de fonctions logiques regroupant plusieurs de ces circuits nécessitait un câblage conséquent, et la réalisation d'un circuit imprimé de grande surface.

Ainsi les fabricants de circuits intégrés numériques s'attachent-ils à fournir des circuits présentant des densités d'intégration toujours plus élevée, pour des vitesses de fonctionnement de plus en plus grandes.

D'abord réalisées avec des circuits **SSI** (Small Scale Integration), les fonctions logiques intégrées se sont développées avec la mise au point du transistor MOS dont la facilité d'intégration a permis la réalisation de circuits **MSI** (Medium Scale Integration) puis **LSI** (Large Scale Integration) puis **VLSI** (Very Large Scale Integration). Ces deux dernières générations ont vu l'avènement des **microprocesseurs et microcontrôleurs**. Bien que ces derniers aient révolutionné l'électronique numérique par la possibilité de réaliser n'importe quelle fonction par programmation d'un composant générique, ils traitent l'information de manière séquentielle (du moins dans les versions classiques), ne répondant pas toujours aux exigences de rapidité.

Au début des années 70 sont apparus les premiers composants (en technologie bipolaire) entièrement configurable par programmation. La nouveauté résidait dans le fait qu'il était maintenant possible d'implanter physiquement par simple programmation, au sein du circuit, n'importe quelle fonction logique, et non plus de se contenter de faire réaliser une opération logique par un microprocesseur dont l'architecture est figée.

D'abord dédiés à des fonctions simples en combinatoire (décodage d'adresse par exemple), ces circuits laissent aujourd'hui au concepteur la possibilité d'implanter des composants aussi

divers qu'un inverseur et un microprocesseur au sein d'un même boîtier ; le circuit n'est plus limité à un mode de traitement séquentielle de l'information comme avec les microprocesseurs.

L'intégration des principales fonctions numériques d'une carte au sein d'un même boîtier permet de répondre à la fois aux critères de densité et de rapidité (les capacités parasites étant plus faibles, la vitesse de fonctionnement peut augmenter). La plupart de ces circuits sont maintenant programmés à partir d'un simple ordinateur type PC directement sur la carte où ils vont être utilisés. En cas d'erreur, ils sont reprogrammables électriquement sans avoir à extraire le composant de son environnement.

De nombreuses familles de circuits sont apparues depuis les années 70 avec des noms très divers suivant les constructeurs. Une certaine inertie dans l'évolution du vocabulaire a fait que certains circuits technologiquement différents ont le même nom. Le terme même de circuit programmable est ambigu, la programmation d'une FPGA ne faisant pas appel aux mêmes opérations que celle d'un microprocesseur. Il serait plus juste de parler pour les PLD, CPLD et FPGA de circuits à architecture programmable ou encore de circuits à réseaux logiques programmables.

Ce domaine de l'électronique est aussi celui qui certainement a vu la plus forte évolution technologique ces dernières années :

- en moins de 15 ans la densité d'intégration a été multipliée par 200 (2000 à 20 000 portes en 85 pour 72 000 à 4 000 000 en 2000).
- en moins de 10 ans la vitesse de fonctionnement a été multipliée par 6 (40MHz en 91 pour 240MHz en 2000).
- la taille d'un transistor est passée de 1,2µm en 91 à 0,18µm en 2000.
- la tension d'alimentation est passée de 5 V à 1,8 V diminuant ainsi la consommation.

Parallèlement à ces circuits, on trouve les **ASIC** (Application Specific Integrated Circuits) qui sont des composants où le concepteur intervient au niveau du dessin de la pastille de silicium en fournissant des masques à un fondeur. On ne peut plus franchement parler de circuits programmables.

Ces circuits sont surtout utilisés pour la faible consommation (quelques uA voire même nA) et pour les circuits analogiques/numériques (oscillateurs, sources de courants, comparateurs, etc...).

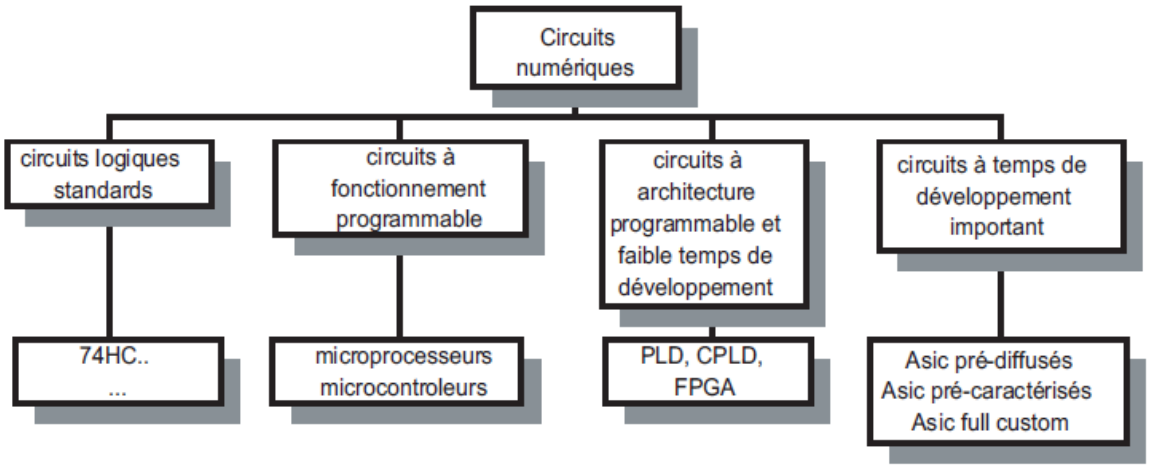
Les PLD, CPLD et FPGA sont parfois considérés comme des ASIC par certains auteurs. Le tableau ci-après tente une classification possible des circuits numérique.

Comme tout domaine spécialisé, le monde des circuits programmables comporte une terminologie, d'origine anglo-saxonne le plus souvent, difficile à éviter. Pour compliquer les choses, de nombreux termes sont à l'origine des noms propres, propriétés des sociétés qui sont à la source des produits concernés. Qui se souvient encore que frigidaire est un nom déposé par la société General Motors ?

Les sigles utilisés dans la suite semblent communément admis par la majorité des fabricants :

- **PLD** (*programmable logic device*) est un terme générique qui recouvre l'ensemble des circuits logiques programmables. Il est le plus souvent employé pour désigner les plus simples d'entre eux (équivalent de quelques centaines de portes logiques).

- **CPLD** (*complex programmable logic device*) désigne évidemment un circuit relativement complexe (jusqu'à une ou deux dizaines de milliers de portes), mais dont l'architecture dérive directement de celle des PLDs simples.
- **FPGA** (*field programmable gate array*) marque un saut dans l'architecture et la technologie, il désigne un circuit qui peut être très complexe (jusqu'à cent mille portes équivalentes) ; la complexité des FPGAs rejoint celle des ASICs (*application specific integrated circuits*).




## I.3. Utilisation des circuits numériques.

1. Les **fonctions standard** sont utilisées pour les applications réalisées en logique câblée. Les catalogues TTL et CMOS présentent plusieurs centaines de fonctions d'usage général, sous forme de circuits intégrés à grande échelle.
2. Les **microprocesseurs**, désormais d'usage courant, sont omniprésents dans les applications industrielles.
3. Dans des applications trop complexes pour être raisonnablement traitées en logique câblée traditionnelle, et trop rapides pour avoir une solution à base de microprocesseurs, on utilise des séquenceurs micro programmés (**PLD, CPLD, FPGA**).
4. Quand les volumes de production importants le justifient, les circuits intégrés spécifiques (**ASICS**) offrent une alternative aux cartes câblées classiques.

## I.4. Domaines d'application.

- Informatique.
- Médical.
- Electronique grand public.
- Militaire/aérospatial.
- Contrôle industriel.
- Automobile.
- Télécommunications.
- Stockage de données.
- Combinés multimédias.
- Boîtiers décodeurs pour télévision.

# II. MEMOIRES.

## II.1. Généralité.

Le développement des microprocesseurs stimule une évolution rapide des technologies de réalisation des mémoires à semi-conducteurs ; les circuits logiques programmables ont hérité directement des mémoires pour ce qui concerne les aspects technologiques. Leurs architectures internes sont, en revanche, très différentes. Il n'est donc pas surprenant que le premier fabriquant de circuits programmables ait été un fabriquant de mémoires (MMI, *monolithic memories inc.*).

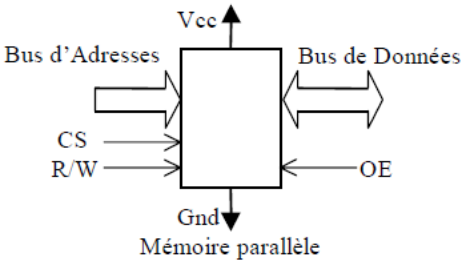
Notons, pour la petite histoire, que cette société a «fusionné» avec l'un des leaders des fabricants de processeurs rapides, processeurs micro programmés, processeurs spécialisés dans le traitement de signal, processeurs RISCs, AMD (*advanced micro devices*).

Indépendamment de sa structure interne et des détails de la technologie concernée, une mémoire est caractérisée par son mode de programmation et sa faculté de retenir l'information quand l'alimentation est interrompue. Les catégories de mémoires qui ont donné naissance aux circuits programmables sont :

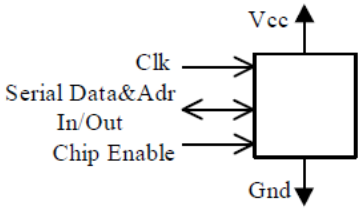
- Les mémoires de type **PROM** (*programmable memory*) sont programmables une seule fois au moyen d'un appareil spécial, le programmateur. Les données qui y sont inscrites ne sont pas modifiables. Elles conservent les informations quand l'alimentation est interrompue. Leur inconvénient majeur est l'impossibilité de modifier les informations qu'elles contiennent.
- Les mémoires de type **EPROM** (*erasable programmable memory*) sont programmables par l'utilisateur au moyen d'un programmateur, effaçables par une exposition aux rayons ultraviolets et reprogrammables après avoir été effacées. Elles aussi conservent les informations quand l'alimentation est interrompue. Leur boîtier doit être équipé d'une fenêtre transparente, ce qui en augmente le coût.
- Les mémoires de type **EEPROM** (*electricaly erasable programmable memory*), ou **FLASH**, sont effaçables et reprogrammables électriquement. Non alimentées, elles conservent les informations mémorisées. La diminution des tensions à appliquer pour programmer les mémoires FLASH permet même de s'affranchir du programmateur : il est intégré dans le circuit. On parle alors de mémoires programmables *in situ* (ISP, pour *in situ programming*), c'est à dire sans démonter la mémoire de la carte sur laquelle elle est implantée. Les technologies FLASH sont de loin les plus séduisantes pour les circuits programmables pas trop complexes.
- Les mémoires **RAM** (*random access memory*) statiques[3], ou SRAM, sont constituées de cellules accessibles, en mode normal, en lecture et en écriture. Elles sont utilisées dans certains circuits programmables complexes pour conserver la configuration (qui définit la fonction réalisée) du circuit. Ces mémoires perdent leur information quand l'alimentation est supprimée.

### II.1.1. Fonctionnement d'une mémoire électronique.

Les mémoires sont des circuits intégrés de forte densité d'intégration capables de stocker de l'information sous forme binaire. Elles sont réalisées en technologie bipolaire ou CMOS. Il y a, en ce qui concerne leur organisation, deux grandes familles : les mémoires à accès parallèle et les mémoires à accès sériel. Dans les mémoires sérielles il existe plusieurs protocoles de communication, les plus répandus étant $I^2C$-Bus, SPI-Bus, Microwire-Bus.



Mémoire parallèle

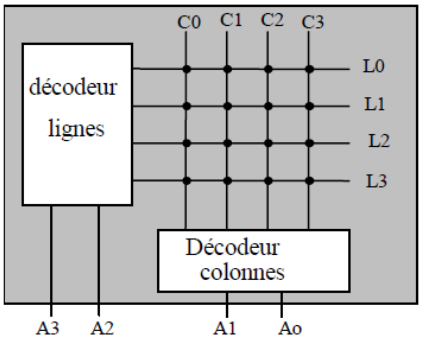


Mémoire sérielle ($I^2C$-Bus)

| | Mémoire parallèle | Mémoire sérielle |
|---|---|---|
| Avantages | Grande capacité.<br>Très rapide. | Encombrement très réduit ; nécessite peu de signaux de commande ; bonne immunité aux parasites |
| Inconvénients | Encombrement ; nécessite beaucoup de signaux de commande | Très lente |

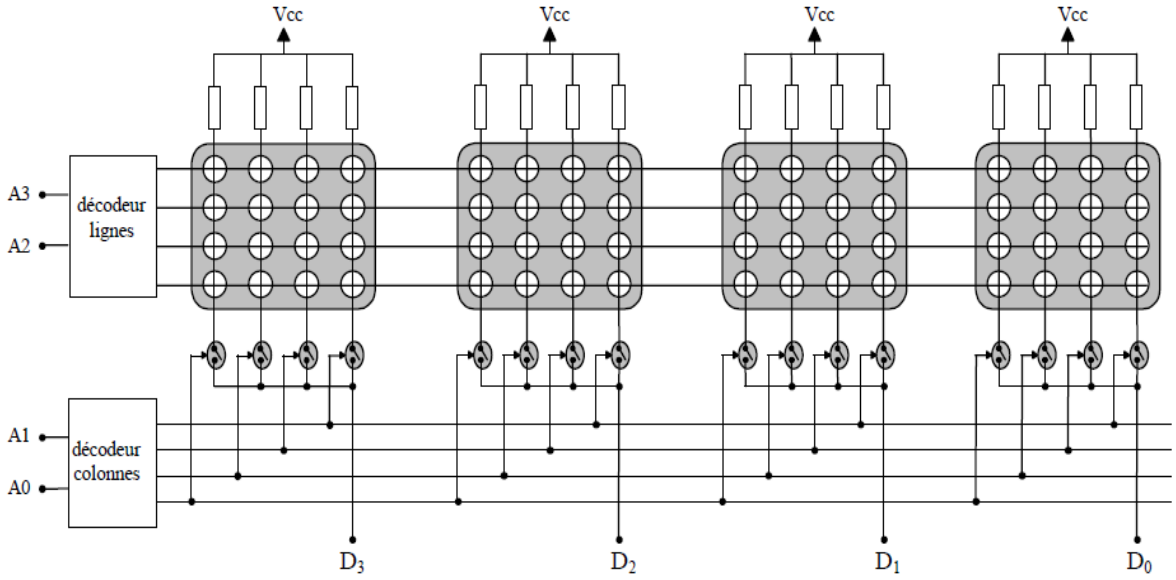
## II.1.2. Architecture des mémoires.

Quel que soit le type de mémoire, les cellules sont organisées en matrice XY. Une cellule est repérée par son numéro de ligne et son numéro de colonne qui constituent ce qu'on appelle **l'adresse** de la cellule. L'exemple de la figure ci-dessous illustre l'exemple d'une mémoire 16 bits, organisée en 4 lignes et 4 colonnes. En utilisant des décodeurs, on a besoin de deux bits d'adresse A1A0 Pour sélectionner une ligne, et de deux bits d'adresse A3A2 pour sélectionner une colonne, soit une adresse globale de 4 bits. Donc en général pour une mémoire de capacité N bits, il faut n bits d'adresses tels que $N=2^n$.



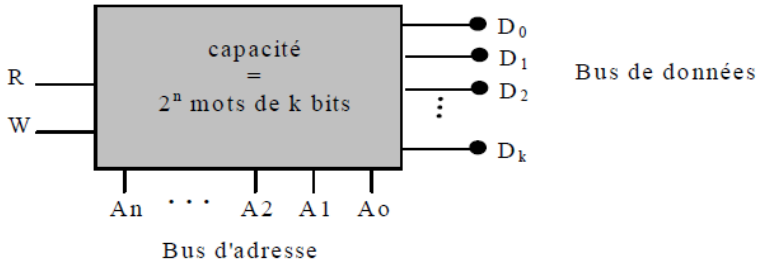

**Organisation par mot.**

Dans la structure donnée ci haut, on ne peut adresser qu'un bit à la fois. Dans la pratique, on désire souvent adresser des mots de plusieurs bits, comme des octets par exemple. Pour faciliter le dessin, la figure suivante montre une mémoire de 16 mots de 4 bits chacun. Elle est obtenue par association de 4 matrices de 16 bits. Toutes les matrices reçoivent la même adresse ligne et colonne. Quand on écrit un mot, chaque bit est stocké dans une matrice. Les circuits de lecture écriture ne sont pas représentés.




Pour éviter toute confusion lors de la détermination de la taille d'une mémoire, se rappeler que :

- Le nombre de bits du BUS DE DONNEES détermine la TAILLE DES MOTS que l'on peut mémoriser dans la mémoire.
- Le nombre de bits du BUS D'ADRESSE détermine la CAPACITE, c'est à dire le NOMBRE DE MOTS que la mémoire peut stocker.




| Nombre de bits d'adresse | Capacité | |
|---|---|---|
| 10 | $2^{10}$ | 1ko |
| 11 | $2^{11}$ | 2ko |
| 20 | $2^{20}$ | 1Mo |
| 21 | $2^{21}$ | 2Mo |
| 22 | $2^{22}$ | 4Mo |
| 23 | $2^{23}$ | 8Mo |
| 24 | $2^{24}$ | 16Mo |
| 25 | $2^{25}$ | 32Mo |
| 26 | $2^{26}$ | 64Mo |
| 27 | $2^{27}$ | 128Mo |
| 28 | $2^{28}$ | 256Mo |
| 29 | $2^{29}$ | 512Mo |
| 30 | $2^{30}$ | 1Go |
| 40 | $2^{40}$ | 1To |

***Tableau : Capacité d'une mémoire en fonction du nombre de bits du bus d'adresse.***

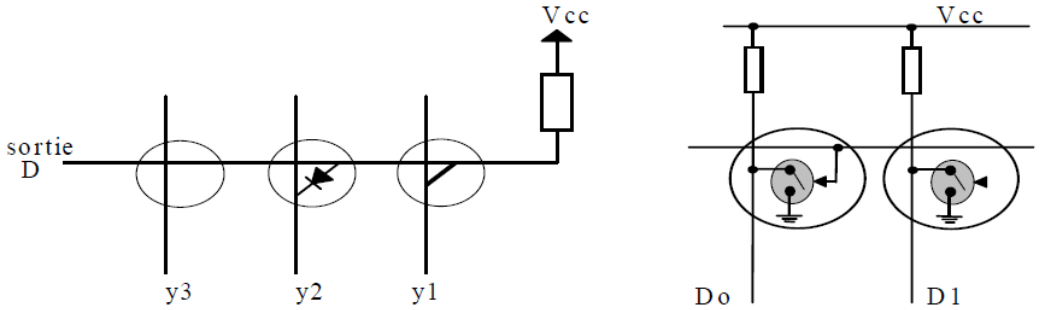
## II.2. Mémoire morte ou ROM.

Une mémoire ROM (Read Only Memory) est une mémoire dont le contenu a été défini et réalisé une bonne fois pour toutes au moment de la fabrication. Sa programmation est faite directement sur le wafer (galette de silicium) à l'aide des masques de programmation. On utilise ce genre de mémoire quand l'information qu'on y enregistre est une information figée qui n'est pas susceptible de subir un changement, comme par exemple les valeurs de la fonction sinus pour les angles compris entre 0 et 90°.

### II.2.1. Cellule d'une mémoire ROM.

Il s'agit essentiellement de présence ou d'absence d'une connexion entre une ligne et une colonne. Cette connexion peut être une métallisation (court-circuit), une diode ou un transistor MOS. Pour lire le contenu de la cellule (i,j), on met la colonne j à 0 et on lit la sortie D sur la ligne i.

- S'il y a présence de connexion ⇒ D = 0.
- S'il y a absence de connexion ⇒ D = 1.




En technologie MOS, le point de connexion est un transistor MOS avec ou sans grille selon que l'on désire mémoriser un 0 ou un 1. Pour lire le contenu de la cellule (i,j), on met la ligne i à 1 et on lit la sortie D sur la colonne j.

- Si c'est un MOS avec grille, il conduit ⇒ Dj = 0.
- Si c'est un MOS sans grille, il ne conduit pas ⇒ D = 1.

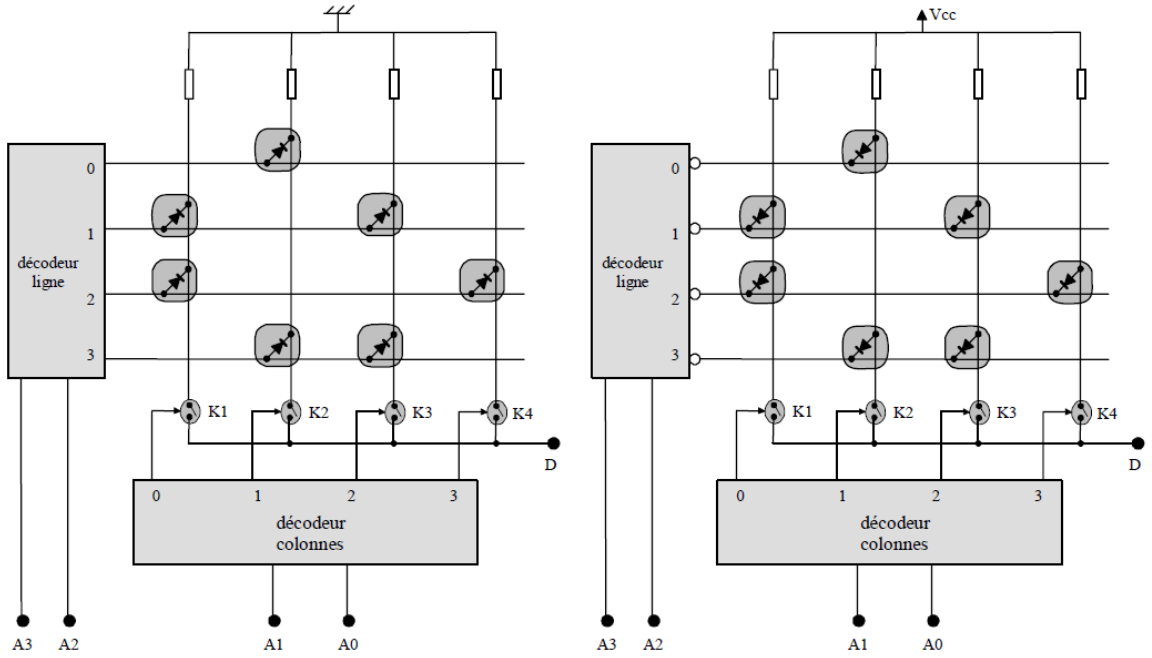



Fig. 4.12 : ROM 16 bits (décodeur ligne actif : (a) niveau haut, (b) niveau bas)

# II.5. Mémoire morte effaçable électriquement ou EEPROM.

Ces mémoire non volatiles présentent l'avantage d'être inscriptible électriquement et effaçable électriquement d'où leur nom EEPROM (Electrically erasable programmable Read Only Memory). Cela permet de gagner du temps car l'effacement électrique prend beaucoup moins de temps que l'effacement par ultraviolets.

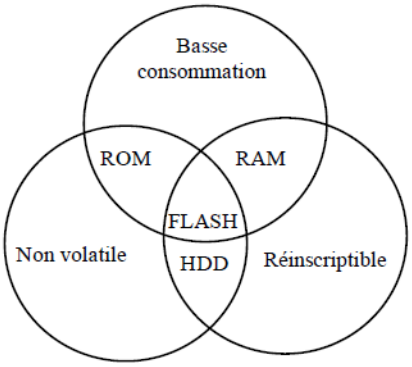
## II.5.1. Cellule d'une mémoire EEPROM.

Les EEPROM utilisent une technologie semblable au ROM avec la propriété d'être effaçable électriquement grâce à la présence d'un transistor MOS. En fait, on peut réécrire dans la mémoire avec une impulsion électrique sans être obligé de l'effacer. Ceci est rendu possible car la zone (tunnel) isolant la grille et le drain du MOSFET a une épaisseur très mince (50 à 200Å contre 1000 pour l'EPROM) ce qui rend possible le déplacement des électrons dans les deux sens.

Le développement des EEPROM a ouvert un champ d'utilisation très important car on a enfin des mémoires électroniques non volatiles. Elles ne sont pas aussi rapides que les RAM, mais en tout cas, bien plus rapides et surtout moins encombrantes que les mémoires magnétiques. Les plus rapides sont appelées mémoires flash. Elles remplacent très avantageusement les disquettes et les cartes magnétiques, mais il faut attendre encore un peu pour arriver à la capacité des disques durs.

# II.3. Mémoire flash.

Les mémoires flash sont des EEPROM à accès rapide. L'accès en lecture est comparable à celui des RAM (<= 100ns). L'accès en écriture est plus long (<= 10µs). Elles rassemblent les avantages des mémoires ROM, des RAM ainsi que des disques durs.




On distingue des variantes à accès parallèle et d'autres à accès série. Sur les ordinateurs, elles sont utilisées surtout pour le stockage du bios. Ailleurs, ces mémoires sont utilisées dans beaucoup d'applications et sont promues à un avenir très prometteur. Les cartes à puces en sont fournies et elles remplacent déjà les disques durs sur certains ordinateurs portables.

## II.4. Mémoire vive ou RAM.

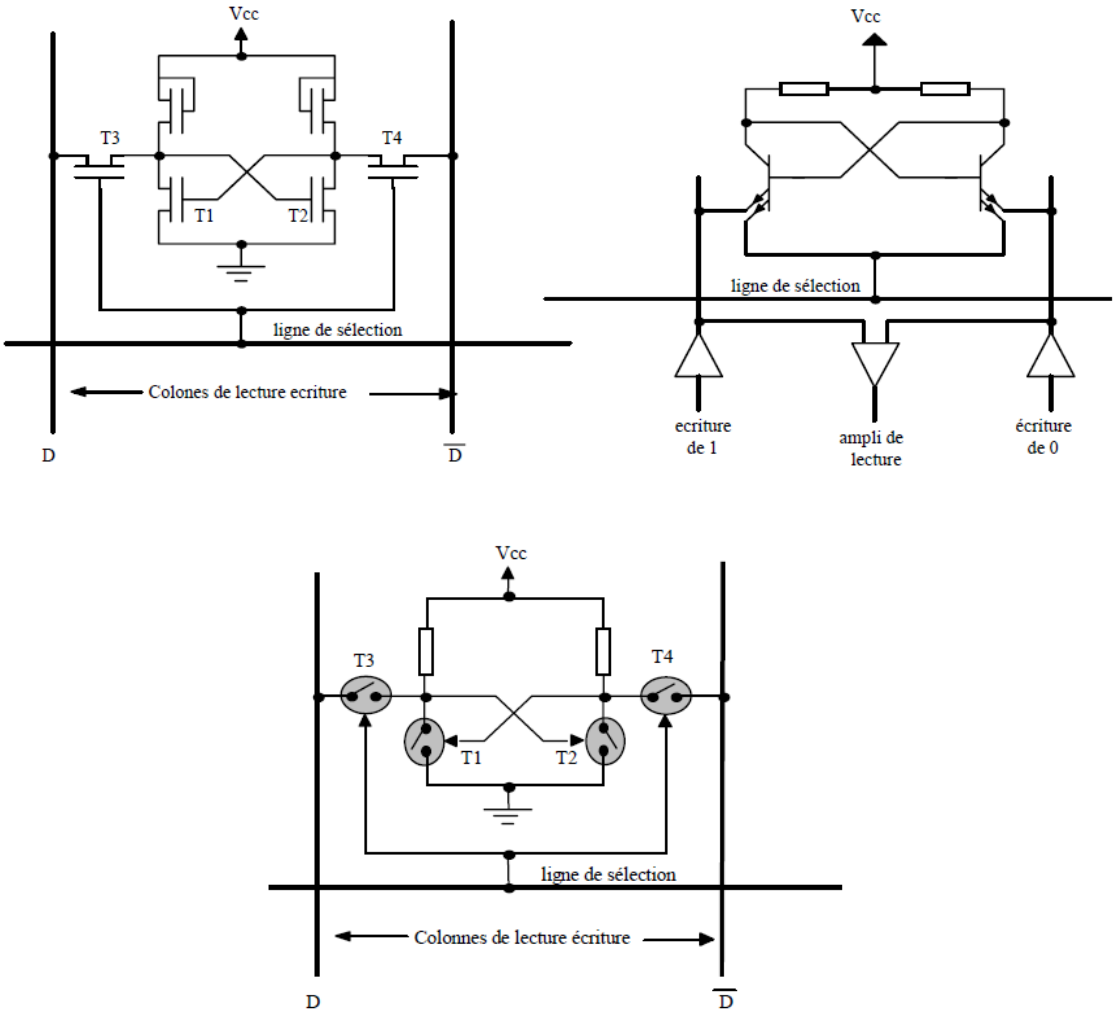
La mémoire vive est une mémoire volatile dans laquelle on peut écrire ou lire une information. En anglais on la désigne sous le sigle RAM (Random Access Memory), mémoire à accès aléatoire, cela signifie qu'après avoir lu ou écrit dans une position mémoire, on peut lire ou écrire dans une autre position quelconque. Ceci par opposition avec les mémoires à accès séquentiel (série), dans lesquels après avoir lu ou écrit dans une position mémoire, la prochaine opération de lecture/écriture ne peut porter que sur la position mémoire immédiatement voisine. Remarquons que la nomenclature RWM (read write memory) aurait été plus appropriée. Le contenu d'une mémoire vive s'efface quand la tension d'alimentation disparaît, d'où la qualification de mémoire **volatile**.

On distingue les RAM statiques et les RAM dynamiques :

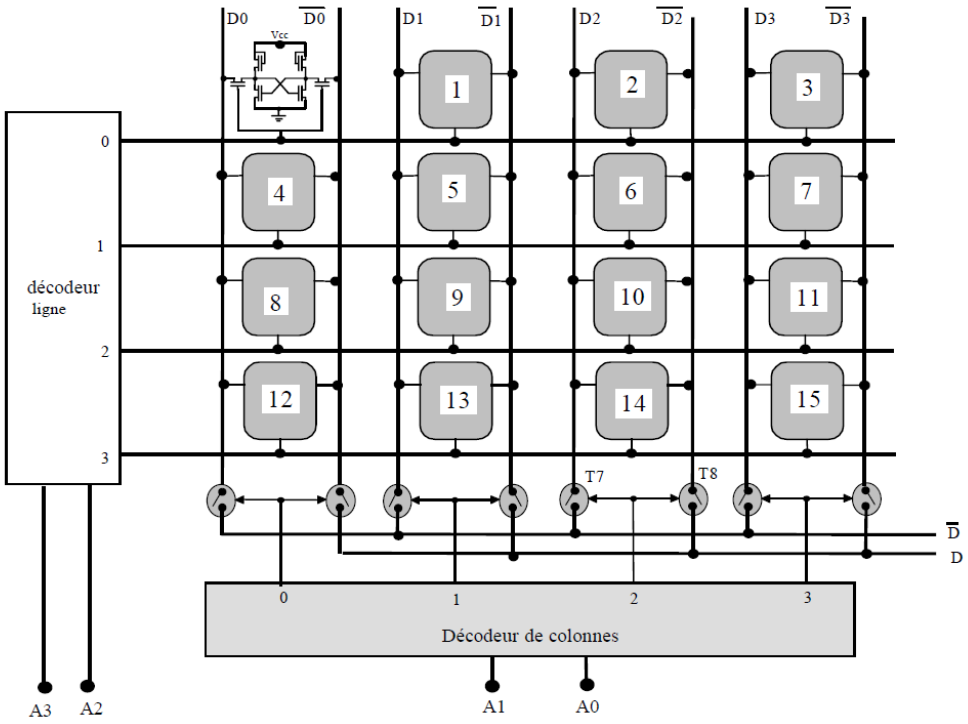
- Le taux d'intégration des RAM statique est assez faible et leur prix de revient (au Mbits) reste relativement élevé, par contre, leur temps d'accès est faible. Elles sont utilisées dans les mémoires caches (interne et externe).
- Le taux d'intégration des RAM dynamique est élevé et leur prix de revient (au Mbits) est plus faible mais leur temps d'accès est assez élevé. Elles sont utilisées dans la mémoire centrale.

### II.7.1. Cellule d'une RAM statique.

Dans les RAM statiques, l'information est stockée dans une bascule (D par exemple). Comme on le sait une fois la sortie de la bascule est dans un état, elle y restera tant qu'on ne vient pas la changer en mettant le bit à enregistrer sur l'entrée D et en envoyant un coup d'horloge sur son entrée horloge. Les mémoires ainsi construites sont appelées les RAM **Statiques (SRAM)**. Toutes les bascules (D, RS, JK) avec ou sans horloge peuvent servir de point de mémorisation, mais pour des raisons d'encombrement, on utilise des bascules bistables constituées de 6 transistors MOS ou de 2 transistors bipolaires. Les MOS sont plus utilisés du fait de leur facilité d'intégration et de leur faible consommation. Même cette solution reste trop encombrante ce qui fait qu'en général les RAM statiques n'ont pas une très grande capacité. La figure ci-dessous illustre la structure interne d'une cellule SRAM.

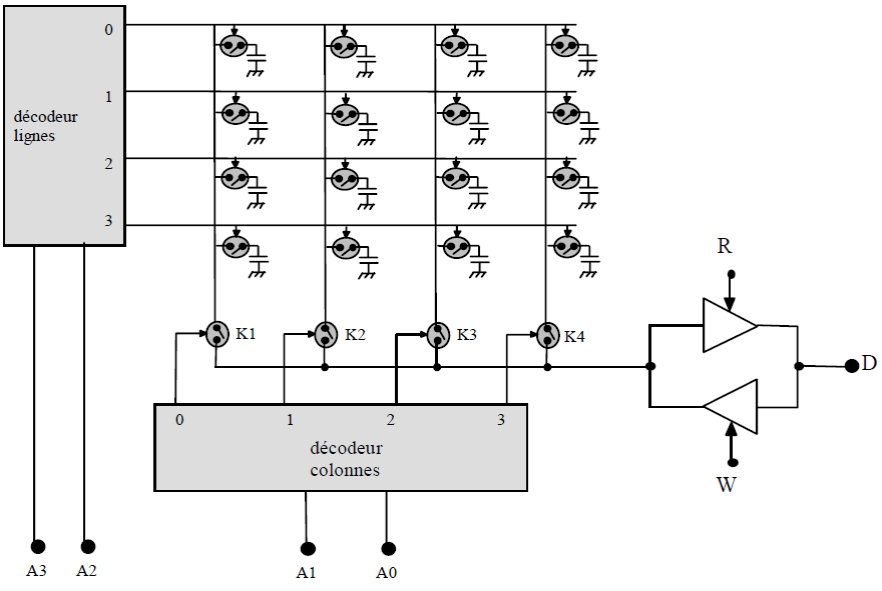



Le schéma de la figure suivante illustre l'exemple d'une RAM statique 16 bits organisée en matrice 4x4.




## II.7.2. Cellule d'une RAM dynamique.

L'information est stockée dans une capacité de structure qui en fait la capacité parasite grille-substrat d'un transistor MOS. Le "1" logique correspond à la capacité chargée et le "0" logique correspond à la capacité déchargée. La cellule de mémorisation de base peut alors être réalisée de façon simplifiée par rapport à celle des mémoires statiques comme cela est illustré sure la figure ci-dessous. Cependant, si cette structure occupe peu de place, elle n'a par contre pas d'état stable car la capacité à tendance à se décharger dans la résistance de fuite associée à la capacité. Il faut donc constamment rafraîchir la mémoire, pour cela on lit la cellule à intervalle régulier (quelques millisecondes) et on réinscrit son contenu. Pour cette raison, la mémoire est dite **dynamique**. Les mémoires dynamiques sont environ 4 fois plus denses que les mémoires statiques de même technologies mais plus délicates d'utilisation.



## II.5. Mémoire FIFO ou file.

Ce sont des mémoires réinscriptibles volatiles organisées de sorte que l'accès se fait d'une façon séquentielle dite premier entré, premier sorti (FIFO : First In First Out). Un mémoire FIFO fonctionne comme une file devant un guichet, le premier qui se place dans la file sera le premier qui arrive au guichet donc le premier servi. Au fure et à mesure que les clients de tête sont servis, les autres progressent dans la file. L'ordre chronologique d'entrée est respecté en sortie.

## II.6. Mémoire LIFO ou pile.

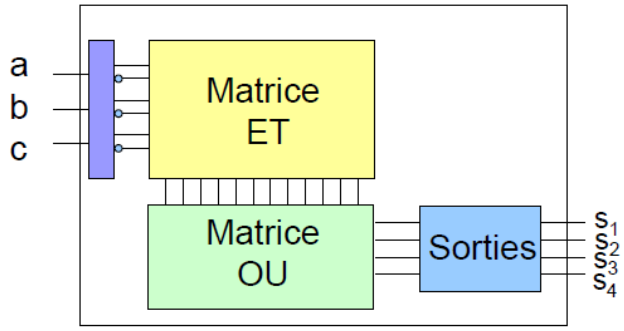
Ce sont des mémoires réinscriptibles volatiles organisées de sorte que l'accès se fait d'une façon séquentielle dite dernier entré, premier sorti (LIFO : Last In First Out). Un mémoire LIFO fonctionne comme une pile d'assiettes, la dernière assiette posée sur le dessus de la pile sera la première à en être retirée.

# III. CIRCUITS LOGIQUES PROGRAMMABLES.

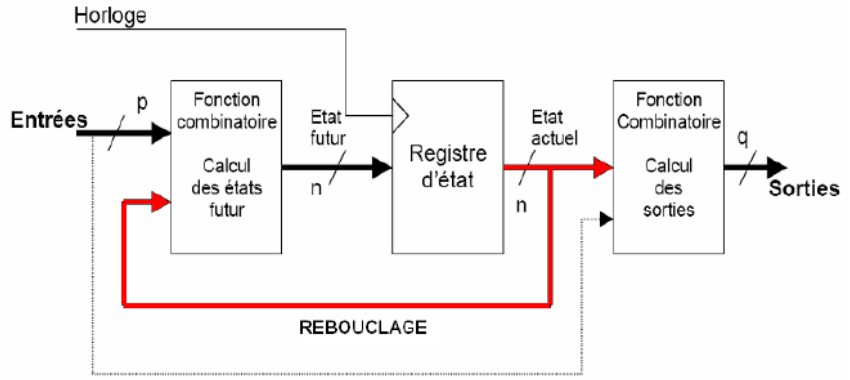
## III.1. PLD.

### III.1.1. Structure.

Un circuit logique programmable (en anglais Programmable Logic Device ou PLD) est un dispositif qui peut être configuré par l'utilisateur pour réaliser une fonction logique quelconque. La plupart des PLD sont constitués de deux réseaux de portes logiques, un réseau AND suivi d'un réseau OR. Les équations des fonctions programmées sont écrites sous la forme minterme ou somme de produits.

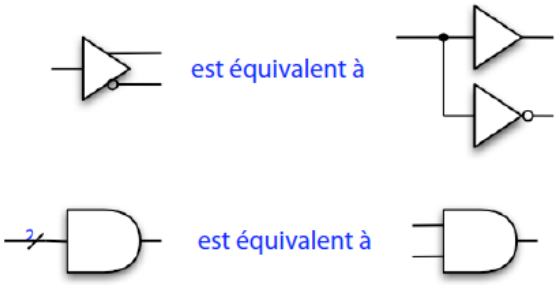


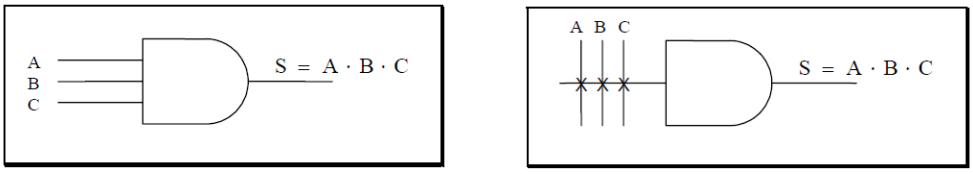

Cette structure interne nous permet pour l'instant de traiter que les équations logiques combinatoires. Pour les équations séquentielles, il faut insérer une bascule et un rebouclage de la sortie vers les entrées.



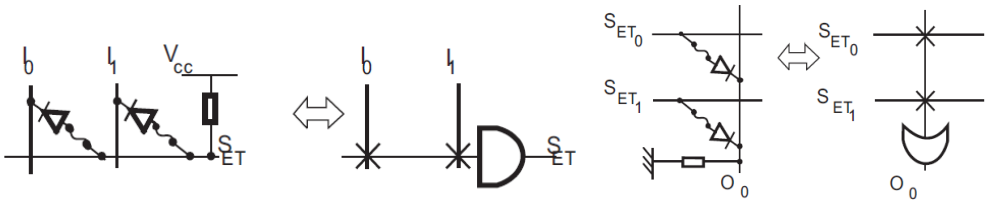

### III.1.2. Convention de notation.

La structure des PLD étant très différente de celle des portes TTL ordinaires, de nouvelles notations logiques ont été développées comme illustrées ci-dessous.

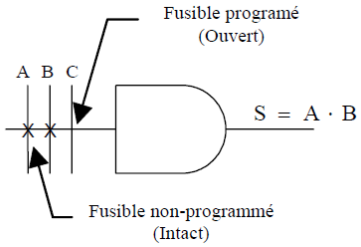




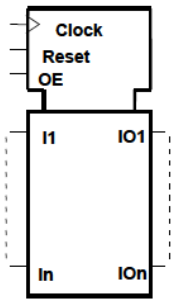

*Représentation standard* *Représentation pour PLD*




- Une porte AND est appelée une ligne de produits.
- Les lignes verticales sont les entrées du PLD.
- Les "X" représentent des fusibles.
- Les fusibles relient les entrées du PLD aux entrées de la porte AND.
- Lors de la programmation d'un PLD un fusible indésirable est ouvert et le "X" disparaît.




## III.1.3. Symbolisation normalisée.

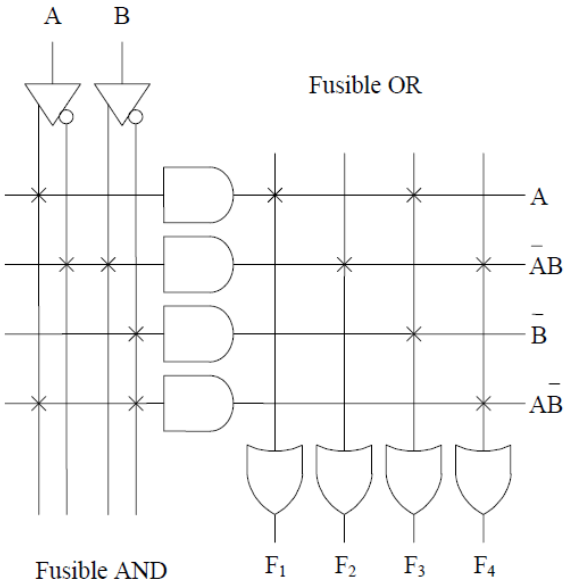



## III.1.4. Classification des PLD.

Nous pouvons citer deux types de base de PLD : les PAL, les PLA. Un GAL (Generic Array Logic) est un PAL effaçable électriquement.

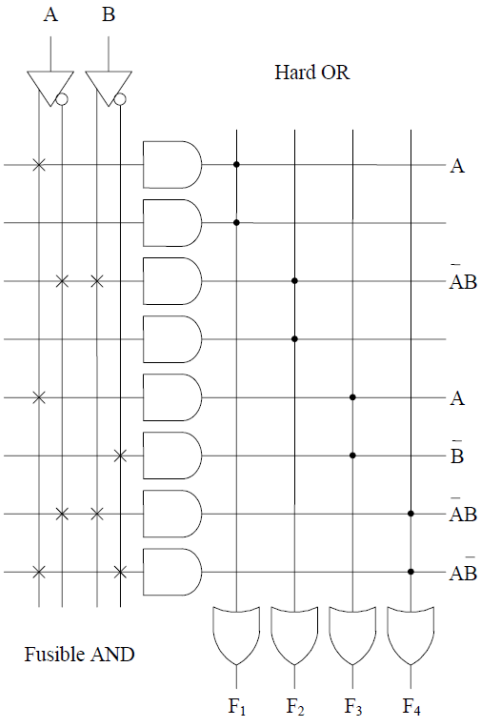
### III.1.4.1. PLA (Programmable Logic Array).

Les deux réseaux AND et OR sont programmable.



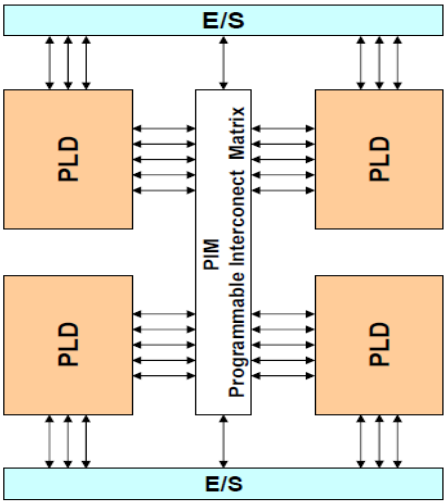

### III.1.4.2. PAL (Programmable Array Logic).

Le réseau AND est programmable et le réseau OR est fixe.



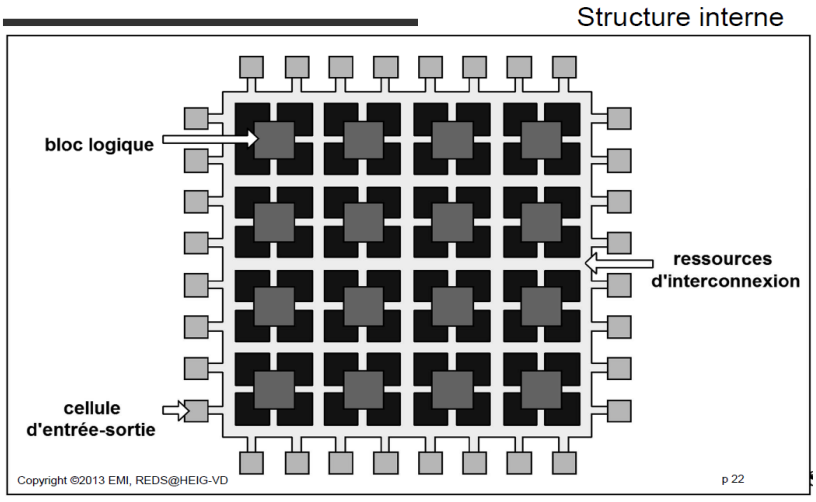

## III.2. CPLD.

Les circuits PLD étudiés jusqu'à maintenant sont connus comme SPLD (Simple Programmable Logic Devices), pour les différencier des circuits plus complexes: les CPLD (Complex Programmable Logic Devices). Les CPLD sont composés d'un certain nombre de SPLD qui partagent une matrice d'interconnexion programmable commune. En plus de la configuration des différents SPLD, il est donc également possible de configurer les interconnexions entre les blocs. Le nombre de portes peut varier entre 100 et 100 000 portes logiques et entre 16 et 1000 bascules voir plus.




## III.3. FPGA (Field Programmable Gate Array).

Ce circuit programmable est composé d'un réseau de petits blocs logiques, de cellules d'entrée-sortie et de ressources d'interconnexion totalement flexibles.




## III.4. ASIC (Application Specific Integrated Circuit).

Si les composants précédents pouvaient être développés avec un simple ordinateur, ceux que nous abordons maintenant nécessitent l'intervention d'un fondeur qui produira le circuit demandé à partir des masques fournis par son client. Ici encore, le terme programmable n'est pas des plus judicieux, les connexions entre les éléments étant dessinées sur les masques. Les temps et coûts de productions sont importants. On distingue trois types d'ASIC classés par ordre croissant de configurabilité.

1. ***Les ASIC prédiffusés (gate arrays).*** Ils contiennent une nébuleuse de transistors ou de portes à interconnecter avec les problèmes de routage et de délais que cela comporte.
2. ***Les ASIC précactérisés (standard cell).*** On utilise cette fois des bibliothèques de cellules standards à placer sur le semi-conducteur
3. ***Les AISC "fulls customs".*** Ils sont entièrement définissables par le client. Ces circuits conduisent à la réalisation de tous les composants VLSI comme les microprocesseurs.

# IV. MICROPROCESSEURS.

## IV.1. Définition.

Le microprocesseur, noté aussi M.P.U. (Microprocessor unit) ou encore C.P.U. (Central Processing Unit) est un circuit intégré complexe appartenant à la famille des VLSI (Very large scale intégration) capable d'effectuer séquentiellement et automatiquement des suites d'opérations élémentaires. Il remplit deux fonctions essentielles :

1. ***Le traitement des données.*** On parle d'unité de traitement. Cette fonction est dédiée à l'Unité Arithmétique Logique. Elle concerne la manipulation des données sous formes de transfert, d'opérations arithmétiques, d'opérations logiques....
2. ***le contrôle du système.*** Cette fonction se traduit par des opérations de décodage et d'exécution des ordres exprimés sous forme d'instruction.

## IV.2. Historique.

Le microprocesseur est l'aboutissement de progrès technologiques tant dans les domaines mécanique, informatique et électronique.

Quelques dates :

- **1690 :** Pascal invente la machine à calculer entièrement mécanique.
- **1800 :** Jacquart invente le métier à tisser avec cartes perforées.
- **1810 :** Invention de l'orgue de barbarie (succession de cartes perforées).
- **1940 :** Premier ordinateur à relais mécaniques (Navy).
- **1946 :** Premier ordinateur à tubes à vide (1800). (grande dissipation : 150kw).
- **1948 :** Progrès de la physique quantique avec découverte de l'effet transistor.
- **1958 :** Développement du premier circuit intégré (4 à 5 tr/puce).
- **1964 :** Ordinateur à transistors (à base de circuits TTL : 50 transistors dans une puce).
- **1970 :** Premiers circuits LSI. Naissance du premier microprocesseur 4 bits avec 1000 transistors sur une puce.
- **1975 :** Naissance du microprocesseur Motorola 6800 (8 bits).
- **1980 :** Apparition du microprocesseur 16 bits avec 50000 transistors sur la puce.
- **1984 :** Apparition du microprocesseur 32 bits avec un million de transistor sur la puce.
- **1994 :** Apparition du Pentium avec 3,5 millions de transistors.

C'est en 1971 que le premier microprocesseur est sorti des laboratoires d'Intel. Travaillant sur 4 bits et d'une puissance faible l'intérêt de ce nouveau composant électronique ne fut pas évident jusqu'à ce que l'idée de le transformer en calculatrice fut trouvée. Sept ans plus tard, l'arrivée du 8088 multiplie déjà cette puissance de calcul par 200. Cette date correspond à la naissance des véritables micro-ordinateurs. Arrivent ensuite les microprocesseurs 68000 et 80286 (16 bits) avec les Macintosh et PC que nous connaissons. Ils ont introduisent l'image et le son. Tout n'est plus qu'une question de course à la puissance de calcul. Chaque bond technologique apporte son innovation. Aujourd'hui, le multimédia puis le 3D et le temps réel. Demain, le monde virtuel !

## IV.3. Puissance d'un microprocesseur.

La notion de puissance est la capacité de traiter un grand nombre d'opérations par seconde sur de grands nombres et en grande quantité. Intrinsèquement la puissance se joue donc sur les trois critères suivants:

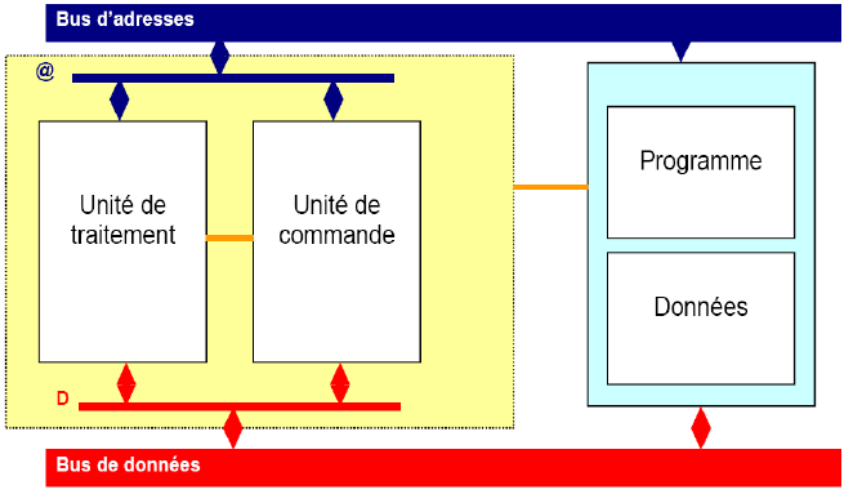
1. ***La longueur des mots :*** données et instructions (on parle de largeur du bus des données).
2. ***Le nombre d'octets*** que le microprocesseur peut adresser (on parle de largeur du bus des adresses).
3. ***La vitesse d'exécution des instructions*** liée à la fréquence de fonctionnement de l'horloge de synchronisation exprimée en MHz.

# IV.4. Architecture d'un microprocesseur.

## IV.4.1. Architecture interne.

Un microprocesseur est construit autour de deux éléments principaux :

1. Une unité de commande.
2. Une unité de traitement.




### IV.4.1.1. L'unité de commande.

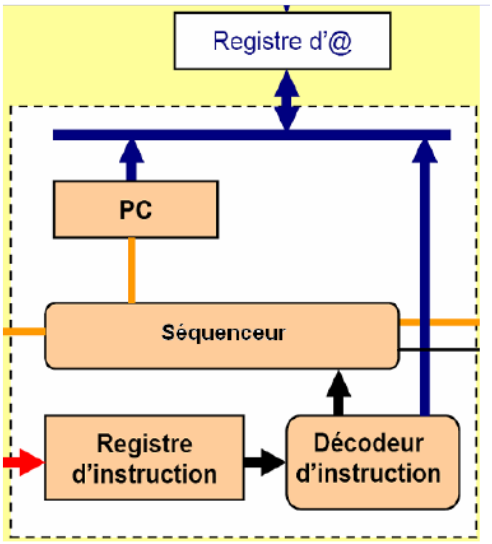
Elle permet de séquencer le déroulement des instructions. Elle effectue la recherche en mémoire de l'instruction, le décodage de l'instruction codée sous forme binaire. Enfin elle pilote l'exécution de l'instruction. Elle est constituée d'un compteur de programme, d'un registre d'instruction, d'un décodeur d'instruction et d'un bloc logique de commande.

**a.** ***Le compteur de programme (PC: Programme Counter ou compteur ordinal : CO).*** Il comporte un registre dont le contenu est initialisé avec l'adresse de la première instruction du programme. Ce registre contient toujours l'adresse de la prochaine instruction à exécuter.

b. ***Le registre d'instruction et le décodeur d'instruction.*** Chacune des instructions à exécuter est transférée depuis la mémoire dans le registre d'instruction puis est décodée par le décodeur d'instruction.

c. ***Le bloc logique de commande (ou séquenceur).*** Il organise l'exécution des instructions au rythme d'une horloge. Il élabore tous les signaux de synchronisation internes ou externes (bus de commande) du microprocesseur en fonction de l'instruction qu'il a à exécuter.

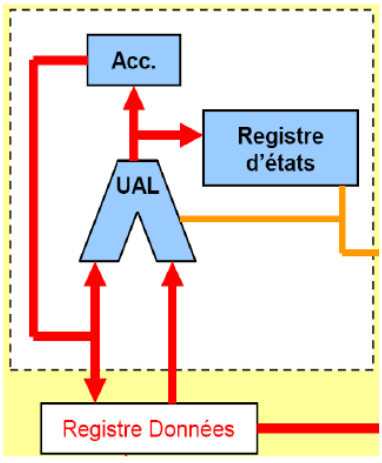
d. ***Le registre d'adresse*** sert d'interface entre le bus des données internes et le bus d'adresse. Son contenu pointe la zone mémoire utile au microprocesseur.

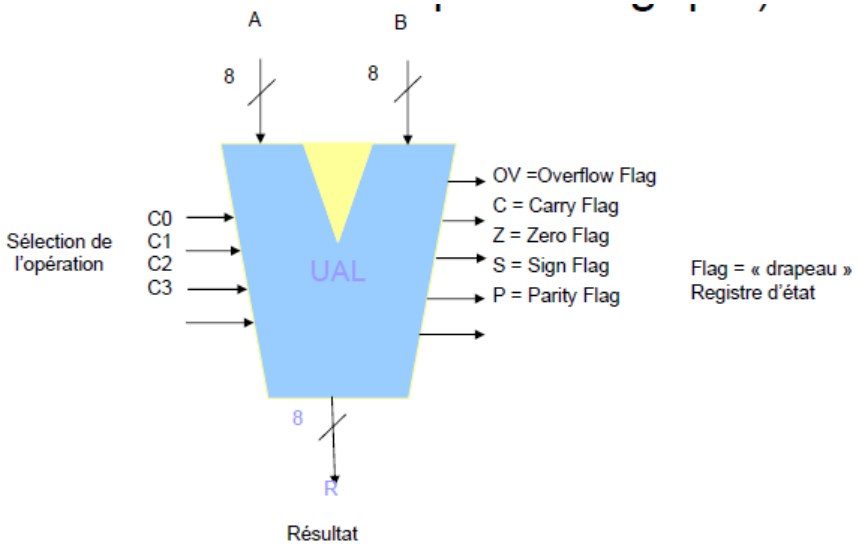



### IV.4.1.2. L'unité de traitement.

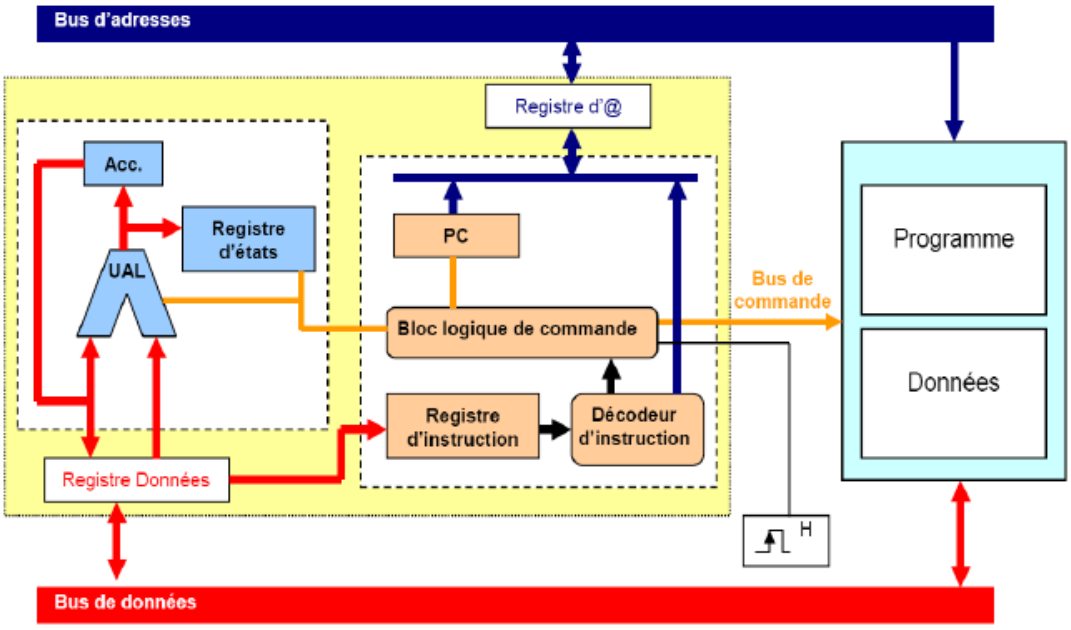
Elle regroupe les circuits qui assurent les traitements nécessaires à l'exécution des instructions (accumulateurs, unité arithmétique et logique, registre d'état).

a. ***Les accumulateurs*** sont des registres de travail qui servent à stocker un opérande au début d'une opération arithmétique et le résultat à la fin de l'opération.
b. ***L'Unité Arithmétique et Logique (UAL), ou ALU (Arithmetic and Logic Unit).*** **C'**est un circuit complexe qui assure les fonctions logiques (ET, OU, Comparaison, Décalage, etc…) ou arithmétique (Addition, soustraction…). Toute instruction qui modifie une donnée fait toujours appel à l'ALU.
c. ***Le registre d'état*** est un registre pour lequel chacun de ses bits est un indicateur dont l'état dépend du résultat de la dernière opération effectuée par l'UAL. On les appelle *indicateur d'état* ou *flag* ou *drapeaux*. Dans un programme, le résultat du test de leur état conditionne souvent le déroulement de la suite du programme. On peut citer par exemple les indicateurs de retenue *(carry : C)*, de débordement *(overflow : OV ou V)*, de zéro *(Z)*, ...





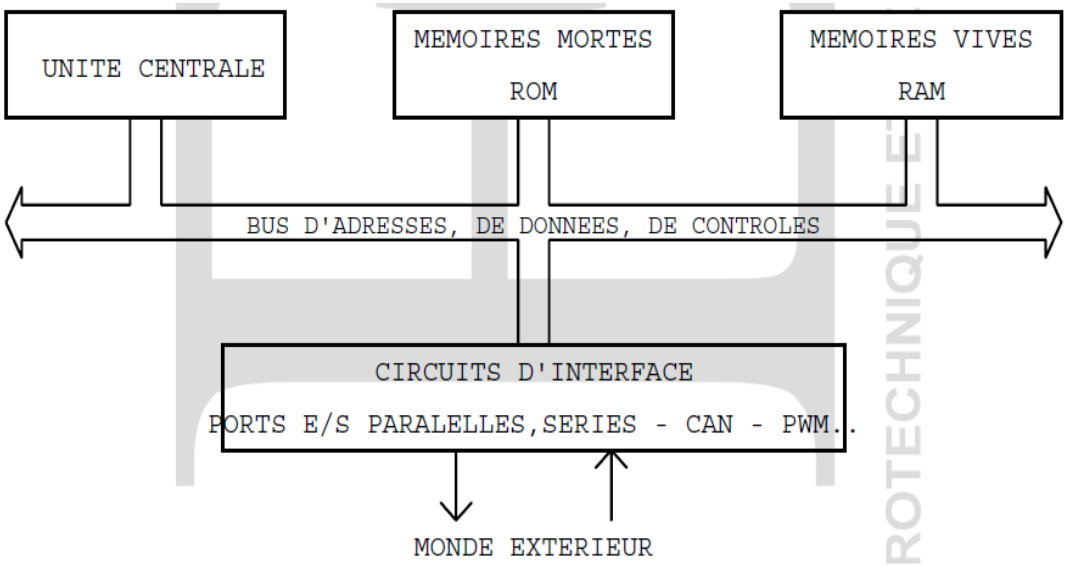

L'architecture interne d'un microprocesseur est ainsi donnée à la figure ci-dessous.




## IV.4.2. Architecture des systèmes à base de micro-processeur.

Le micro-processeur a besoin, pour fonctionner, d'un environnement minimum : celui-ci est au moins composé du processeur lui-même (exécutant les tâches à accomplir) des mémoires ROM (contenant le programme à exécuter) et RAM (stockant des données temporairement) ainsi que des interfaces avec l'extérieur (clavier, lecteur de carte, écran, capteur, etc…).

L'ensemble des boîtiers dialogue par l'intermédiaire des bus (ensemble de fils implantés en circuit imprimé), ce qui représente parfois un volume important et un coût élevé.




## IV.5. Introduction au jeu d'Instructions.

Le programme que doit exécuter un microprocesseur est une succession d'instructions ordonnées (chaque instruction pouvant prend plusieurs octets) qui se trouve rangé dans une zone mémoire, généralement à des adresses successives.

### IV.5.1. Durée d'une instruction.

L'exécution complète d'une instruction n'est pas instantanée. L'unité de mesure est la **période** de l'**horloge** encore appelé "Cycle Machine". La durée d'une instruction dépend de la complexité de l'instruction. Son expression est :

$$T_{instruction} = n.T_{cycle}.$$

Où n est le nombre de cycle que peut prendre l'instruction.

### IV.5.2. Mode de fonctionnement d'une instruction.

Exécuter une instruction c'est réaliser le cycle extraction-exécution.

- *L'extraction* consiste à lire les données en mémoire.
- *L'exécution* consiste à traduire et à interpréter les données une fois traduite

Les instructions utilisent toutes une méthode particulière d'accéder aux informations qu'elles manipulent. Ces méthodes sont appelées ***« modes d'adressage »*** et sous souvent fournies dans les fiches techniques des microprocesseurs. Le code correspondant aux modes d'adressage est appelé ***« le code mnémonique ».***

La ***« pile »*** est la zone mémoire RAM gérée par des pointeurs qui permettent de transférer rapidement des données dans des cases mémoires selon un protocole bien établi.

### IV.5.3. Architecture des instructions.

Actuellement l'architecture des microprocesseurs se composent de deux grandes familles :

1. ***L'architecture CISC (Complex Instruction Set Computer).*** Ce type de microprocesseur possède un nombre important d'instructions. Chacune d'elles s'exécute en plusieurs périodes d'horloges.

2. ***L'architecture RISC (Reduced Instruction Set Computer).*** Ces microprocesseurs possèdent un nombre réduit d'instructions. Chacune d'elles s'exécute en une période d'horloge.

| Architecture RISC | Architecture CISC |
|---|---|
| instructions simples ne prenant qu'un seul cycle | instructions complexes prenant plusieurs cycles |
| instructions au format fixe | instructions au format variable |
| décodeur simple (câblé) | décodeur complexe (microcode) |
| beaucoup de registres | peu de registres |
| peu de modes d'adressage | beaucoup de modes d'adressage |
| compilateur complexe | compilateur simple |

## IV.6. Les interruptions.

Une interruption est une rupture de séquence asynchrone, c'est-à-dire non synchronisée avec le déroulement normal du programme. Nous pouvons ainsi dire, sans se tromper de beaucoup, qu'une routine d'interruption est un sous-programme particulier, déclenché par l'apparition d'un événement spécifique.

Voici donc comment cela fonctionne :

- Le programme se déroule normalement.
- L'événement survient.
- Le programme achève l'instruction en cours de traitement.
- Le programme saute à l'adresse de traitement de l'interruption.
- Le programme traite l'interruption.
- Le programme saute à l'instruction qui suit la dernière exécutée dans le programme principal.

Il va bien sûr de soi que n'importe quel événement ne peut pas déclencher une interruption. Il faut que 2 conditions principales soient remplies :

1. L'événement en question doit figurer dans la liste des événements susceptibles de provoquer une interruption pour le processeur sur lequel on travaille.
2. L'utilisateur doit avoir autorisé l'interruption, c'est à dire doit avoir signalé que l'événement en question devait générer une interruption.

# V. MICROCONTROLLEUR. Exemple : PIC16F877A.

## V.1. Généralité.

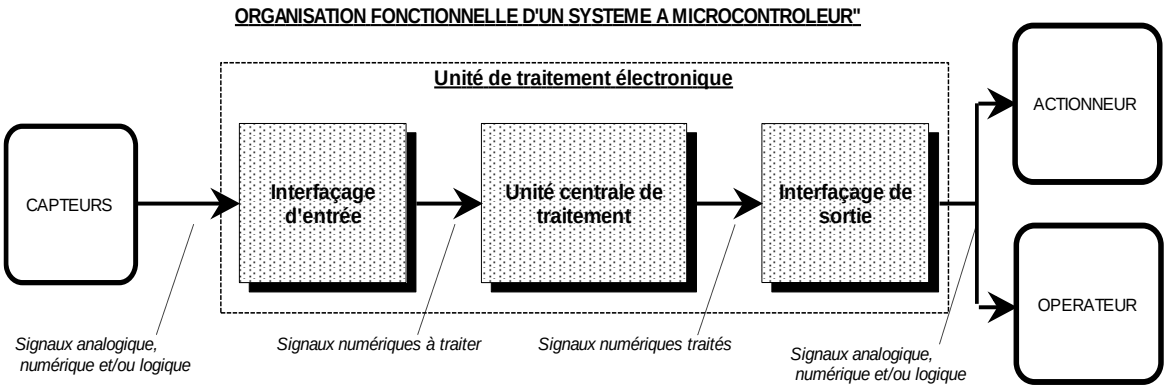
Le microcontrôleur est né lorsque les technologies d'intégration ont suffisamment progressé pour permettre sa fabrication et aussi parce que très souvent, dans des applications domestiques ou industrielles, on a besoin d'un système "intelligent" ,complet, puissant, facile à mettre en œuvre et d'une grande souplesse d'emploi ; le tout dans un seul boîtier. Il permet d'avoir, dans un seul boîtier, tous les éléments nécessaire à la mise en œuvre d'un automatisme industriel. Il ne reste qu'à l'interfacer avec les différents capteurs et pré-actionneurs du processus à commander. Grâce à l'arrivée des microcontrôleurs, des cartes qui contenaient des dizaines de circuits intégrés logiques se sont vues réduites à un seul boîtier.

On définit ainsi un microcontrôleur comme étant une unité de traitement de l'information de type microprocesseur à laquelle on a ajouté des périphériques internes permettant de réaliser des montages sans nécessiter l'ajout de composants externes.

### V.1.1. Architecture externe.

Les fabricants de microcontrôleurs intègrent autant que possible dans un seul boîtier l'ensemble des fonctions de la structure d'un système à microprocesseur. Cette intégration poussée est généralement réalisée en technologie HCMOS qui permet une diminution de la consommation énergétique, une simplification du tracé du circuit imprimé (nombre réduit de boîtiers) ainsi qu'une réduction globale du coût.

Ainsi la structure générale d'un système à microcontrôleur est donnée à la figure ci-dessous.



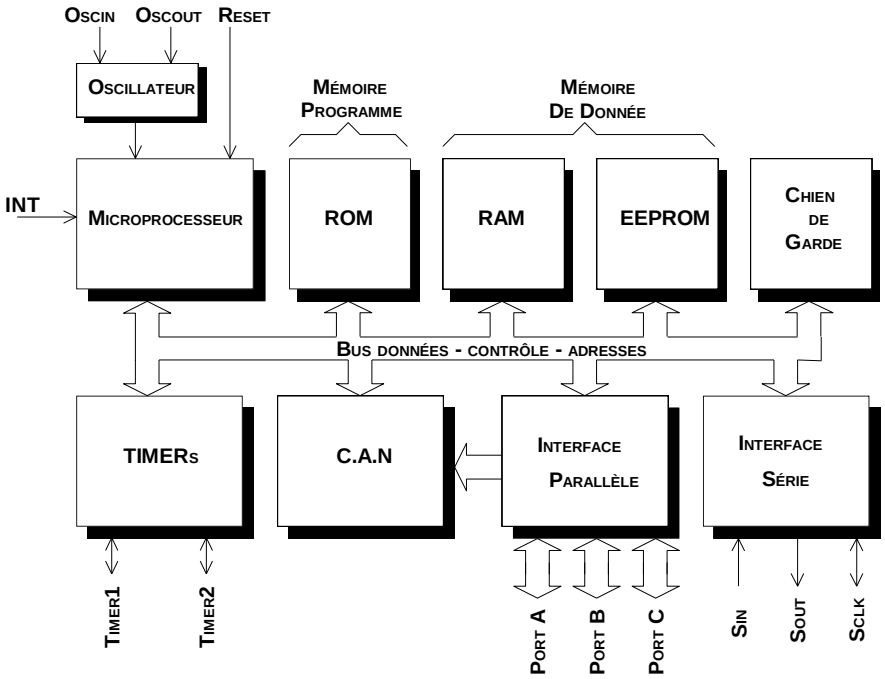

### V.1.2. Architecture interne.

La structure interne d'un microcontrôleur comporte typiquement :

- Une unité de calcul et de commande (CPU).
- De la mémoire de donnée (RAM, EEPROM).
- De la mémoire programme (ROM, PROM, EPROM, EEPROM).
- Un compteur/temporisateur (timer) pour générer ou mesurer des signaux avec une grande précision temporelle.
- Des interfaces parallèles pour la connexion des entrées/sorties.
- Des interfaces série (synchrones et asynchrones) pour le dialogue avec d'autres unités.

Il peut aussi posséder :

- Un chien de garde (Watchdog) qui assure la surveillance du programme.
- Une sortie PWM (Pulse Width Modulation) pour la modulation d'impulsion.
- Un convertisseur analogique-numérique et numérique-analogique (CAN/CNA) pour le traitement des signaux analogiques.




## V.1.3. Généralité sur les microcontrôleurs de type PIC.

Un PIC n'est rien d'autre qu'un microcontrôleur. La dénomination PIC est sous copyright de Microchip. Ses caractéristiques principales sont :

- Séparation des mémoires de programme et de données (architecture Harvard) : On obtient ainsi une meilleure bande passante et des instructions et des données pas forcément codées sur le même nombre de bits.
- Communication avec l'extérieur seulement par des ports : il ne possède pas de bus d'adresses, de bus de données et de bus de contrôle comme la plupart des microprocesseurs.
- Utilisation d'un jeu d'instructions réduit, d'où le nom de son architecture : RISC (Reduced Instructions Set Construction). Les instructions sont ainsi codées sur un nombre réduit de bits, ce qui accélère l'exécution (1 cycle machine par instruction sauf pour les sauts qui requièrent 2 cycles). En revanche, leur nombre limité oblige à se restreindre à des instructions basiques, contrairement aux systèmes d'architecture CISC (Complex Instructions Set Construction) qui proposent plus d'instructions donc codées sur plus de bits mais réalisant des traitements plus complexes.

### a) Les différentes familles des PIC.

Il existe trois familles de PIC :

1. La famille ***Base-Line*** pour laquelle les instructions sont codées sur 12 bits.
2. La famille ***Mid-Range*** pour laquelle les instructions sont codées sur 14 bits.
3. La famille ***High-End*** pour laquelle les instructions sont codées sur 16 bits.

### b) Identification d'un PIC.

Un PIC est identifié par un numéro de la forme suivant : **xx(L)XXyy –zz.**

- **xx** : Famille du composant (**12** pour **Base-Line**, **16** pour **Mid-Range**, **18** pour **High-End**).
- **L** : Tolérance plus importante de la plage de tension.
- **XX** : Type de mémoire de programme (C : EPROM ou EEPROM ; CR: PROM ; F : flash).
- **yy** : Identification.
- **zz** : Vitesse maximum du quartz.

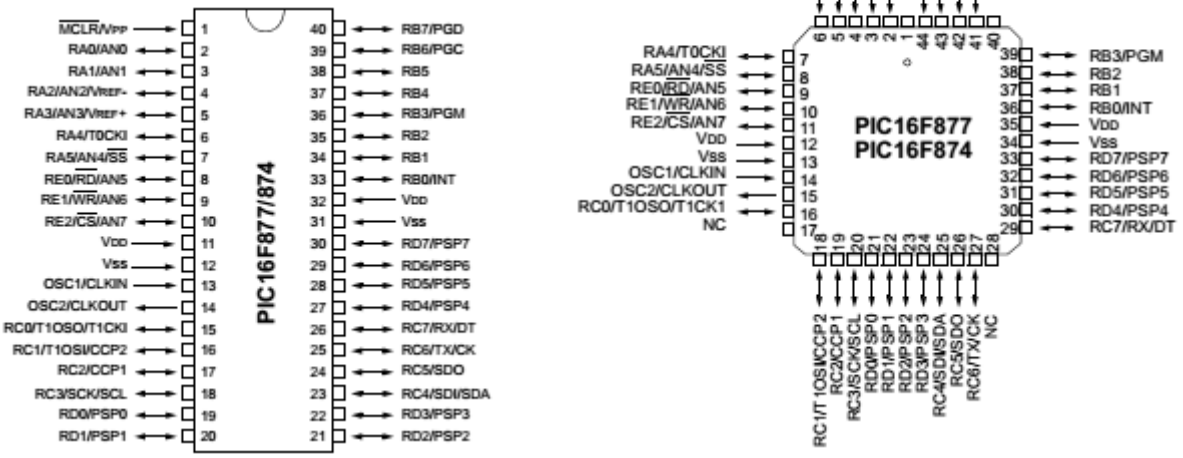
## V.2. Description générale du PIC16F877A.

Le PIC16F877A est un microcontrôleur de la famille mid-range qui se présente sous la forme d'un circuit intégré disponible en boîtier DIL de 40 broches. Il est réalisé en technologie HCMOS FLASH, et est cadencé par une horloge interne ou externe pouvant avoir une fréquence de 0 à 20MHz.

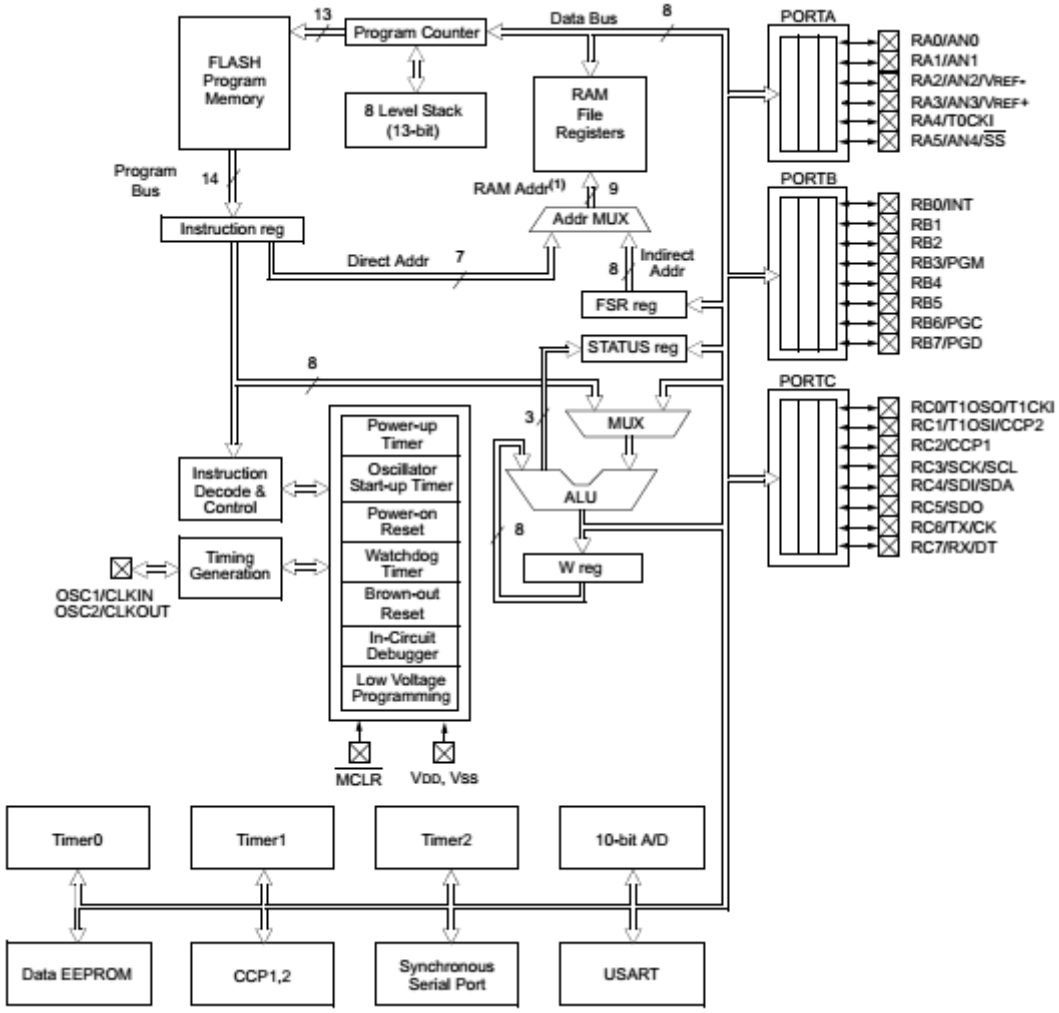
Les principales caractéristiques des séries 16F87X sont données dans le tableau ci-dessous.

| Key Features PICmicro™ Mid-Range Reference Manual (DS33023) | PIC16F873 | PIC16F874 | PIC16F876 | PIC16F877 |
|---|---|---|---|---|
| Operating Frequency | DC - 20 MHz | DC - 20 MHz | DC - 20 MHz | DC - 20 MHz |
| RESETS (and Delays) | POR, BOR (PWRT, OST) | POR, BOR (PWRT, OST) | POR, BOR (PWRT, OST) | POR, BOR (PWRT, OST) |
| FLASH Program Memory (14-bit words) | 4K | 4K | 8K | 8K |
| Data Memory (bytes) | 192 | 192 | 368 | 368 |
| EEPROM Data Memory | 128 | 128 | 256 | 256 |
| Interrupts | 13 | 14 | 13 | 14 |
| I/O Ports | Ports A,B,C | Ports A,B,C,D,E | Ports A,B,C | Ports A,B,C,D,E |
| Timers | 3 | 3 | 3 | 3 |
| Capture/Compare/PWM Modules | 2 | 2 | 2 | 2 |
| Serial Communications | MSSP, USART | MSSP, USART | MSSP, USART | MSSP, USART |
| Parallel Communications | — | PSP | — | PSP |
| 10-bit Analog-to-Digital Module | 5 input channels | 8 input channels | 5 input channels | 8 input channels |
| Instruction Set | 35 instructions | 35 instructions | 35 instructions | 35 instructions |

Les différents circuits de brochage sont donnés ci-dessous.




Le schéma bloc est donné à la figure ci-dessous.



# V.3. Paramètres spéciaux de configuration.

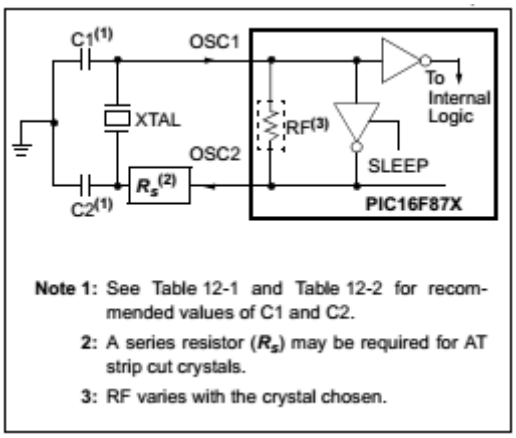
## V.3.1. L'oscillateur.

Le PIC16F877A peut fonctionner suivant quatre modes d'oscillateurs différents :

1. Le mode LP (Low Power Crystal) : Oscillateur à quartz faible consommation.
2. Le mode XT (Crystal/Resonator) : Oscillateur à quartz.
3. Le mode HS (High Speed Crystal/Resonator): Oscillateur à quartz de haute fréquence.
4. Le mode RC (Resistor/Capacitor): Oscillateur RC.

Avec l'oscillateur à Quartz, on peut avoir des fréquences allant jusqu'à 20 MHz selon le type de µC. Le filtre passe bas (Rs, C1, C2) limite les harmoniques dus à l'écrêtage et réduit l'amplitude de l'oscillation, il n'est pas obligatoire.

Le schéma correspondant au mode oscillateur à quartz est donné à la figure suivante.



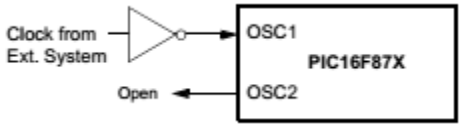

**Note 1:** See Table 12-1 and Table 12-2 for recommended values of C1 and C2.
**2:** A series resistor ($R_s$) may be required for AT strip cut crystals.
**3:** RF varies with the crystal chosen.

Les valeurs remarquables des composants conseillées par le fabricant pour la mise en œuvre de l'oscillateur sont données ci-dessous.

| Ranges Tested: | | | |
|---|---|---|---|
| **Mode** | **Freq.** | **OSC1** | **OSC2** |
| XT | 455 kHz<br>2.0 MHz<br>4.0 MHz | 68 - 100 pF<br>15 - 68 pF<br>15 - 68 pF | 68 - 100 pF<br>15 - 68 pF<br>15 - 68 pF |
| HS | 8.0 MHz<br>16.0 MHz | 10 - 68 pF<br>10 - 22 pF | 10 - 68 pF<br>10 - 22 pF |
| **These values are for design guidance only.** See notes following Table 12-2. | | | |
| **Resonators Used:** | | | |
| 455 kHz | Panasonic EFO-A455K04B | ± 0.3% | |
| 2.0 MHz | Murata Erie CSA2.00MG | ± 0.5% | |
| 4.0 MHz | Murata Erie CSA4.00MG | ± 0.5% | |
| 8.0 MHz | Murata Erie CSA8.00MT | ± 0.5% | |
| 16.0 MHz | Murata Erie CSA16.00MX | ± 0.5% | |
| All resonators used did not have built-in capacitors. | | | |

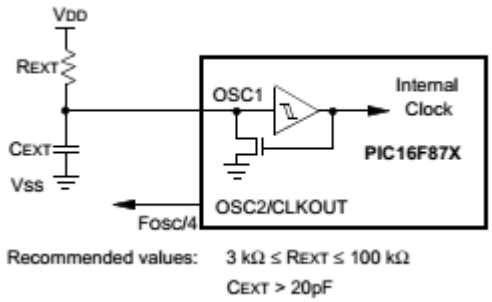
| Osc Type | Crystal Freq. | Cap. Range C1 | Cap. Range C2 |
|---|---|---|---|
| LP | 32 kHz | 33 pF | 33 pF |
| | 200 kHz | 15 pF | 15 pF |
| XT | 200 kHz | 47-68 pF | 47-68 pF |
| | 1 MHz | 15 pF | 15 pF |
| | 4 MHz | 15 pF | 15 pF |
| HS | 4 MHz | 15 pF | 15 pF |
| | 8 MHz | 15-33 pF | 15-33 pF |
| | 20 MHz | 15-33 pF | 15-33 pF |
| **These values are for design guidance only.** See notes following this table. | | | |
| **Crystals Used** | | | |
| 32 kHz | Epson C-001R32.768K-A | ± 20 PPM | |
| 200 kHz | STD XTL 200.000KHz | ± 20 PPM | |
| 1 MHz | ECS ECS-10-13-1 | ± 50 PPM | |
| 4 MHz | ECS ECS-40-20-1 | ± 50 PPM | |
| 8 MHz | EPSON CA-301 8.000M-C | ± 30 PPM | |
| 20 MHz | EPSON CA-301 20.000M-C | ± 30 PPM | |

Il est aussi possible de faire tourner le microcontrôleur avec un oscillateur externe, comme indiqué à la figure suivante.




Avec un oscillateur RC, la fréquence de l'oscillation est fixée par Vdd, Rext et Cext. Elle peut varier légèrement d'un circuit à l'autre.

Le schéma correspondant au mode oscillateur RC est donné à la figure suivante. Dans ce mode, la précision est faible en plus la fréquence n'est pas stable.




Quelque soit l'oscillateur utilisé, l'horloge système dite aussi horloge instruction est obtenue en divisant la fréquence par 4. Dans la suite de ce document on utilisera le terme Fosc/4 pour désigner l'horloge système.

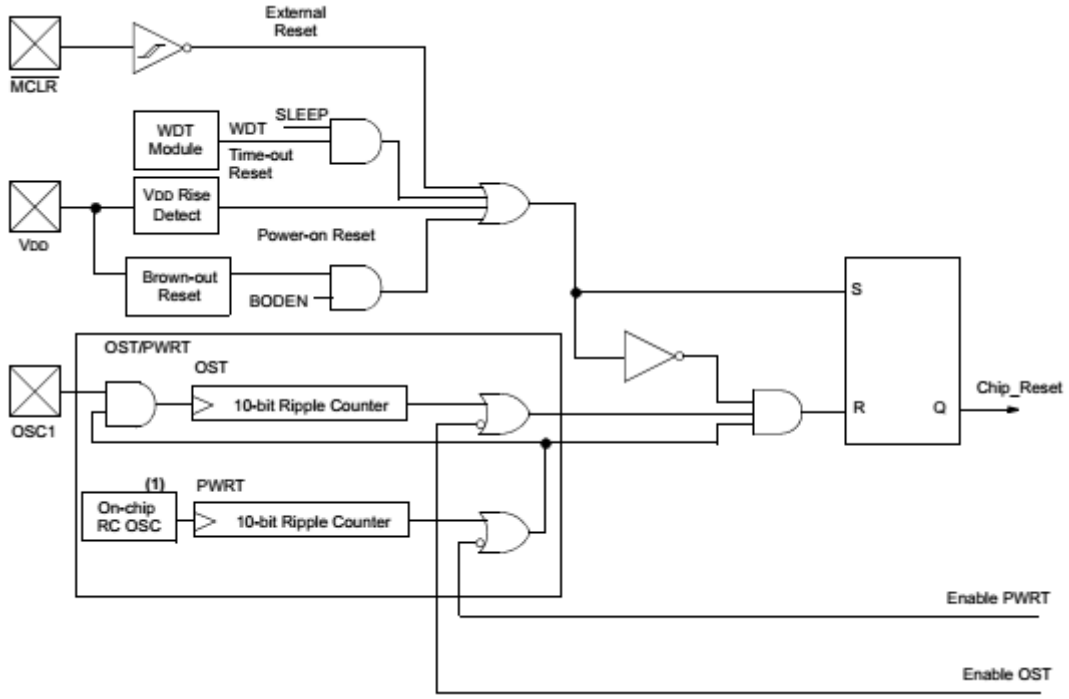
Avec un quartz de 4 MHz, on obtient une horloge instruction de 1 MHz, soit 1µs le temps pour exécuter une instruction.

## V.3.2. Les différents types de Reset (réinitialisation).

Le PIC16F877A peut être réinitialisé selon six sources de reset.

1. Power On Reset (POR). Réinitialise à la mise sous tension.
2. $\overline{MCLR}$ (Master clear) reset during normal operation. Réinitialisation principale qui peut survenir lorsque le composant est en fonctionnement normal.
3. $\overline{MCLR}$ (Master clear) reset during sleep mode. Réinitialisation principale qui peut survenir lorsque le composant est en veille.
4. WDT (Watch Dog Timer) reset during normal mode. Réinitialisation provoquée par le chien de garde en mode normal.
5. WDT (Watch Dog Timer) wake-up during sleep mode. Réinitialisation provoquant la sortie de veille.
6. Brown out Reset (BOR). Réinitialisation lorsque la tension tombe sous une valeur critique pendant le fonctionnement.

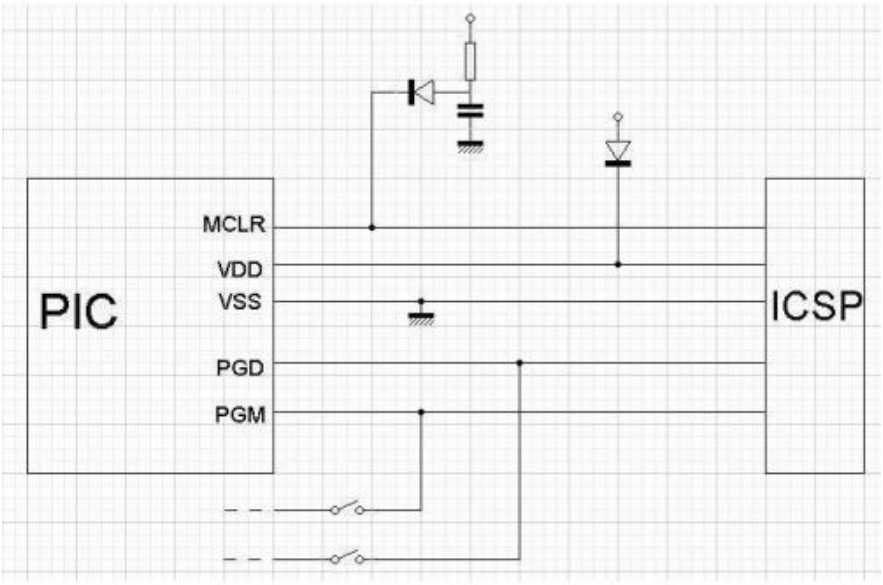
Le schéma descriptif de ces différents types de reset est donné ci-dessous.




## V.3.3. Programmation sur circuit.

Le PIC16F877A peut être programmé directement sur le circuit d'application finale, sans avoir besoin de programmateur spécial. Cela est possible à l'aide :

- ✓ De la ligne d'horloge PGC.
- ✓ De la ligne de donnée PGD.
- ✓ De la ligne de masse GND
- ✓ De la ligne d'alimentation Vdd
- ✓ De la ligne de programmation Vpp. Vpp doit être de l'ordre 13V.



### V.3.4. Programmation basse tension sur circuit.

La programmation basse tension est configurée à l'aide du bit LVP (Low Voltage Programming). Cette mode permet au microcontrôleur d'être programmé à partir d'une faible tension (environ Vdd). Dans cette configuration, la broche PGM est utilisée pour recevoir le signl d'horloge. Pendant la programmation, la tension Vdd est appliquée sur la broche $\overline{MCLR}$.

### V.3.5. Le watchdog.

Le watchdog, ou chien de garde est un mécanisme de protection de votre programme. Il sert à surveiller si celui-ci s'exécute toujours dans l'espace et dans le temps que vous lui avez attribués.

La mise en service ou l'arrêt du watchdog se décide au moment de la programmation de votre PIC. Si « _WDT_OFF » est précisé, le watchdog ne sera pas en service. Si au contraire vous précisez « _WDT_ON », le watchdog sera actif.

Il n'est donc pas possible de mettre en ou hors service le watchdog durant l'execution de votre programme.

Le fonctionnement du watchdog est lié à un timer interne spécifique, qui n'est pas synchronisé au programme, ni à un événement extérieur.

La durée spécifique de débordement de ce timer est approximativement de 18ms. Cette valeur est à prendre avec précaution, car elle varie en fonction de différents paramètres comme la tension d'alimentation ou la température. La valeur minimale de 7ms est celle que vous devrez utiliser dans la pratique.

En effet, Microchip vous garanti qu'aucun PIC ne provoquera un reset avant ces 7ms. Il vous indique que le temps moyen de reset de ses PICs sera de 18ms, mais il ne vous garantit pas ce temps, c'est juste un temps « généralement constaté ».

Chaque fois que l'instruction clrwdt est envoyé au PIC, le timer du watchdog est remis à 0, ainsi que la valeur contenue dans son prédiviseur. Si par accident cette instruction n'est pas reçue dans le délai prévu, le PIC est redémarré à l'adresse 0x00 et le bit TO du registre STATUS est mis à 0.

En lisant ce bit au démarrage, vous avez donc la possibilité de détecter si le PIC vient d'être mis sous tension, ou si ce démarrage est dû à un « plantage » de votre programme.

### V.3.6. Le mode Sleep.

Voici un mode très particulier des PIC, qui leur permet de se mettre en sommeil afin de limiter leur consommation.

Le mode « sleep » ou « power down » est un mode particulier dans lequel vous pouvez placer votre PIC grâce à l'instruction « sleep ». Une fois dans ce mode, le PIC est placé en sommeil et cesse d'exécuter son programme. Dès réception de cette instruction, la séquence suivante est exécutée :

- ✓ Le watchdog est remis à 0, exactement comme le ferait une instruction « clrwdt ».
- ✓ Le bit TO du registre STATUS est mis à 1.
- ✓ Le bit PD du registre STATUS est mis à 0.
- ✓ L'oscillateur est mis à l'arrêt, le PIC n'exécute plus aucune instruction.

Une fois dans cet état, le PIC est à l'arrêt. La consommation du circuit est réduite au minimum. Si le TMR0 est synchronisé à l'horloge interne, il est également mis dans l'incapacité de compter.

Par contre, il est très important de se rappeler que le timer du watchdog possède son propre circuit d'horloge. Ce dernier continue de compter comme si de rien n'était.

Pour profiter au maximum de la chute de la consommation (montage sur piles par exemple), Microchip recommande de veiller à ce que les pins d'entrées/sorties et l'électronique connectée soient à des niveaux 0 ou 1 tels qu'il n'y ait aucun passage de courant qui résulte du choix de ces niveaux.

## V.4. Les ports E/S.

Le PIC16F877A dispose de cinq ports bidirectionnels d'E/S (port A à port E). Certaines broches de ces ports sont multiplexées avec d'autres fonctions de périphériques internes (comparateur et référence de tension par exemple). Chaque borne du port a donc plusieurs rôles qui doivent être définis par des registres de configuration associés.

### V.4.1. Le port A.

Le tableau ci-dessous décrit les différentes fonctions multiplexées sur le port A.

| Name | Bit# | Buffer | Function |
|---|---|---|---|
| RA0/AN0 | bit0 | TTL | Input/output or analog input. |
| RA1/AN1 | bit1 | TTL | Input/output or analog input. |
| RA2/AN2 | bit2 | TTL | Input/output or analog input. |
| RA3/AN3/VREF | bit3 | TTL | Input/output or analog input or VREF. |
| RA4/T0CKI | bit4 | ST | Input/output or external clock input for Timer0. Output is open drain type. |
| RA5/$\overline{SS}$/AN4 | bit5 | TTL | Input/output or slave select input for synchronous serial port or analog input. |

Legend: TTL = TTL input, ST = Schmitt Trigger input

- ✓ RA0 à RA5 : Entrée / Sortie numérique.
- ✓ AN0 à AN3 : Entrées analogiques.
- ✓ Vref : Tension de référence, on la fixe par programmation.
- ✓ T0CKI : Timer Clock In ; entrée d'horloge du TMR0.
- ✓ $\overline{SS}$ : Entrée de sélection esclave pour la communication série synchrone.

Les registres associés à la gestion du port A sont données dans le tableau ci-dessous.

| Address | Name | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Value on: POR, BOR | Value on all other RESETS |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 05h | PORTA | — | — | RA5 | RA4 | RA3 | RA2 | RA1 | RA0 | --0x 0000 | --0u 0000 |
| 85h | TRISA | — | — | PORTA Data Direction Register | | | | | | --11 1111 | --11 1111 |
| 9Fh | ADCON1 | ADFM | — | — | — | PCFG3 | PCFG2 | PCFG1 | PCFG0 | --0- 0000 | --0- 0000 |

Legend: x = unknown, u = unchanged, - = unimplemented locations read as '0'.
Shaded cells are not used by PORTA.

*b) Registre TRISA.*

Ce registre permet de définir si la patte considérée fonctionne en entrée ou en sortie.

- Un «1» dans un bit du registre TRISA configure la broche correspondante entrée.
- Un «0» dans un bit de ce registre configure la broche correspondante entrée sortie.

## V.4.2. Le port B.

Le tableau ci-dessous décrit les différentes fonctions multiplexées sur le port B.

| Name | Bit# | Buffer | Function |
|---|---|---|---|
| RB0/INT | bit0 | TTL/ST(1) | Input/output pin or external interrupt input. Internal software programmable weak pull-up. |
| RB1 | bit1 | TTL | Input/output pin. Internal software programmable weak pull-up. |
| RB2 | bit2 | TTL | Input/output pin. Internal software programmable weak pull-up. |
| RB3/PGM(3) | bit3 | TTL | Input/output pin or programming pin in LVP mode. Internal software programmable weak pull-up. |
| RB4 | bit4 | TTL | Input/output pin (with interrupt-on-change). Internal software programmable weak pull-up. |
| RB5 | bit5 | TTL | Input/output pin (with interrupt-on-change). Internal software programmable weak pull-up. |
| RB6/PGC | bit6 | TTL/ST(2) | Input/output pin (with interrupt-on-change) or In-Circuit Debugger pin. Internal software programmable weak pull-up. Serial programming clock. |
| RB7/PGD | bit7 | TTL/ST(2) | Input/output pin (with interrupt-on-change) or In-Circuit Debugger pin. Internal software programmable weak pull-up. Serial programming data. |

Legend: TTL = TTL input, ST = Schmitt Trigger input
**Note 1:** This buffer is a Schmitt Trigger input when configured as the external interrupt.
**2:** This buffer is a Schmitt Trigger input when used in Serial Programming mode.
**3:** Low Voltage ICSP Programming (LVP) is enabled by default, which disables the RB3 I/O function. LVP must be disabled to enable RB3 as an I/O pin and allow maximum compatibility to the other 28-pin and 40-pin mid-range devices.

- ✓ RB0 à RB7 : Entrée / Sortie numérique.
- ✓ INT : Entrée d'interruption externe.
- ✓ PGM : Broche de programmation en mode LVP.
- ✓ PGC : Entrée d'horloge en mode programmation.
- ✓ PGD : Entrée de donnée en mode programmation.

Les registres associés à la gestion du port B sont données dans le tableau ci-dessous.

| Address | Name | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Value on: POR, BOR | Value on all other RESETS |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 06h, 106h | PORTB | RB7 | RB6 | RB5 | RB4 | RB3 | RB2 | RB1 | RB0 | xxxx xxxx | uuuu uuuu |
| 86h, 186h | TRISB | PORTB Data Direction Register | | | | | | | | 1111 1111 | 1111 1111 |
| 81h, 181h | OPTION_REG | RBPU | INTEDG | T0CS | T0SE | PSA | PS2 | PS1 | PS0 | 1111 1111 | 1111 1111 |

Legend: x = unknown, u = unchanged. Shaded cells are not used by PORTB.

## V.4.3. Le port C.

Le tableau ci-dessous décrit les différentes fonctions multiplexées sur le port C.

| Name | Bit# | Buffer Type | Function |
|---|---|---|---|
| RC0/T1OSO/T1CKI | bit0 | ST | Input/output port pin or Timer1 oscillator output/Timer1 clock input. |
| RC1/T1OSI/CCP2 | bit1 | ST | Input/output port pin or Timer1 oscillator input or Capture2 input/ Compare2 output/PWM2 output. |
| RC2/CCP1 | bit2 | ST | Input/output port pin or Capture1 input/Compare1 output/ PWM1 output. |
| RC3/SCK/SCL | bit3 | ST | RC3 can also be the synchronous serial clock for both SPI and $I^2C$ modes. |
| RC4/SDI/SDA | bit4 | ST | RC4 can also be the SPI Data In (SPI mode) or data I/O ($I^2C$ mode). |
| RC5/SDO | bit5 | ST | Input/output port pin or Synchronous Serial Port data output. |
| RC6/TX/CK | bit6 | ST | Input/output port pin or USART Asynchronous Transmit or Synchronous Clock. |
| RC7/RX/DT | bit7 | ST | Input/output port pin or USART Asynchronous Receive or Synchronous Data. |

Legend: ST = Schmitt Trigger input

- ✓ RC0 à RC7 : Entrée / Sortie numérique.
- ✓ T1OSO : Timer 1 Oscillateur Out ; sortie de l'oscillateur du TMR1.
- ✓ T1OSI : Timer 1 Oscillateur In ; entrée de l'oscillateur du TMR1.
- ✓ T1CKI : Timer 1 Clock Int ; entrée d'horloge du Timer 1.
- ✓ CCP2 : Capture 2 input / Capture 2 output / PWM 2 output.
- ✓ SCK/SCL: Horloge en mode SPI/$I^2C$.
- ✓ SDI/SDA : Entrée ou sortie de donnée en mode SPI/ $I^2C$.
- ✓ SDO : Sortie de donnée en mode SSP (Synchronous Serial Port).
- ✓ TX/RX : Transmission/Réception en mode USART asynchrone.
- ✓ CK/DT : Horloge/Entrée-Sortie de donnée en mode USART synchrone.

Les registres associés à la gestion du port C sont données dans le tableau ci-dessous.

| Address | Name | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Value on: POR, BOR | Value on all other RESETS |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 07h | PORTC | RC7 | RC6 | RC5 | RC4 | RC3 | RC2 | RC1 | RC0 | xxxx xxxx | uuuu uuuu |
| 87h | TRISC | PORTC Data Direction Register | | | | | | | | 1111 1111 | 1111 1111 |

Legend: x = unknown, u = unchanged

## V.4.4. Le port D.

Le tableau ci-dessous décrit les différentes fonctions multiplexées sur le port D.

| Name | Bit# | Buffer Type | Function |
|---|---|---|---|
| RD0/PSP0 | bit0 | ST/TTL(1) | Input/output port pin or parallel slave port bit0. |
| RD1/PSP1 | bit1 | ST/TTL(1) | Input/output port pin or parallel slave port bit1. |
| RD2/PSP2 | bit2 | ST/TTL(1) | Input/output port pin or parallel slave port bit2. |
| RD3/PSP3 | bit3 | ST/TTL(1) | Input/output port pin or parallel slave port bit3. |
| RD4/PSP4 | bit4 | ST/TTL(1) | Input/output port pin or parallel slave port bit4. |
| RD5/PSP5 | bit5 | ST/TTL(1) | Input/output port pin or parallel slave port bit5. |
| RD6/PSP6 | bit6 | ST/TTL(1) | Input/output port pin or parallel slave port bit6. |
| RD7/PSP7 | bit7 | ST/TTL(1) | Input/output port pin or parallel slave port bit7. |

Legend: ST = Schmitt Trigger input, TTL = TTL input
**Note 1:** Input buffers are Schmitt Triggers when in I/O mode and TTL buffers when in Parallel Slave Port mode.

- ✓ PSP0-PSP7 : Port esclave parallèle.

Les registres associés à la gestion du port D sont données dans le tableau ci-dessous.

| Address | Name | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Value on: POR, BOR | Value on all other RESETS |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 08h | PORTD | RD7 | RD6 | RD5 | RD4 | RD3 | RD2 | RD1 | RD0 | xxxx xxxx | uuuu uuuu |
| 88h | TRISD | PORTD Data Direction Register | | | | | | | | 1111 1111 | 1111 1111 |
| 89h | TRISE | IBF | OBF | IBOV | PSPMODE | — | PORTE Data Direction Bits | | | 0000 -111 | 0000 -111 |

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used by PORTD.

## V.4.5. Le port E.

Le tableau ci-dessous décrit les différentes fonctions multiplexées sur le port E.

| Name | Bit# | Buffer Type | Function |
|---|---|---|---|
| RE0/$\overline{RD}$/AN5 | bit0 | ST/TTL(1) | I/O port pin or read control input in Parallel Slave Port mode or analog input:<br>$\overline{RD}$<br>1 = Idle<br>0 = Read operation. Contents of PORTD register are output to PORTD I/O pins (if chip selected) |
| RE1/$\overline{WR}$/AN6 | bit1 | ST/TTL(1) | I/O port pin or write control input in Parallel Slave Port mode or analog input:<br>$\overline{WR}$<br>1 = Idle<br>0 = Write operation. Value of PORTD I/O pins is latched into PORTD register (if chip selected) |
| RE2/$\overline{CS}$/AN7 | bit2 | ST/TTL(1) | I/O port pin or chip select control input in Parallel Slave Port mode or analog input:<br>$\overline{CS}$<br>1 = Device is not selected<br>0 = Device is selected |

Legend: ST = Schmitt Trigger input, TTL = TTL input
**Note 1:** Input buffers are Schmitt Triggers when in I/O mode and TTL buffers when in Parallel Slave Port mode.

- ✓ $\overline{RD}$ : Configuration du port D en lecture. A 1 Les contenus du registre PORTD sont disponibles sur les broches du port D.
- ✓ $\overline{WR}$ : Configuration du port D en écriture. A 1 Les valeurs des broches du port D sont stockées dans le registre PORTD.
- ✓ $\overline{CS}$ : Contrôle de la sélection du composant en mode PSP. A 0 le composant est sélectionné.

Les registres associés à la gestion du port D sont données dans le tableau ci-dessous.

| Address | Name | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Value on: POR, BOR | Value on all other RESETS |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 09h | PORTE | — | — | — | — | — | RE2 | RE1 | RE0 | ---- -xxx | ---- -uuu |
| 89h | TRISE | IBF | OBF | IBOV | PSPMODE | — | PORTE Data Direction Bits | | | 0000 -111 | 0000 -111 |
| 9Fh | ADCON1 | ADFM | — | — | — | PCFG3 | PCFG2 | PCFG1 | PCFG0 | --0- 0000 | --0- 0000 |

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used by PORTE.

***Description du registre TRISE.***

| R-0 | R-0 | R/W-0 | R/W-0 | U-0 | R/W-1 | R/W-1 | R/W-1 |
|---|---|---|---|---|---|---|---|
| IBF | OBF | IBOV | PSPMODE | — | Bit2 | Bit1 | Bit0 |
| bit 7 | | | | | | | bit 0 |

**Parallel Slave Port Status/Control Bits:**

bit 7 **IBF:** Input Buffer Full Status bit
1 = A word has been received and is waiting to be read by the CPU
0 = No word has been received

bit 6 **OBF**: Output Buffer Full Status bit
1 = The output buffer still holds a previously written word
0 = The output buffer has been read

bit 5 **IBOV**: Input Buffer Overflow Detect bit (in Microprocessor mode)
1 = A write occurred when a previously input word has not been read (must be cleared in software)
0 = No overflow occurred

bit 4 **PSPMODE**: Parallel Slave Port Mode Select bit
1 = PORTD functions in Parallel Slave Port mode
0 = PORTD functions in general purpose I/O mode

bit 3 **Unimplemented**: Read as '0'

**PORTE Data Direction Bits:**

bit 2 **Bit2**: Direction Control bit for pin RE2/$\overline{CS}$/AN7
1 = Input
0 = Output

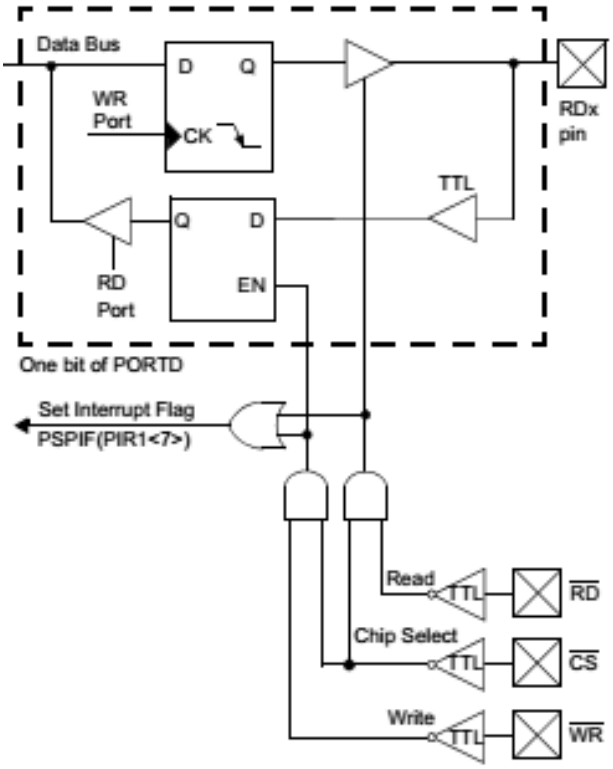
bit 1 **Bit1**: Direction Control bit for pin RE1/$\overline{WR}$/AN6
1 = Input
0 = Output

bit 0 **Bit0**: Direction Control bit for pin RE0/$\overline{RD}$/AN5
1 = Input
0 = Output

| Legend: | | | |
|---|---|---|---|
| R = Readable bit | W = Writable bit | U = Unimplemented bit, read as '0' | |
| - n = Value at POR | '1' = Bit is set | '0' = Bit is cleared | x = Bit is unknown |

## V.4.6. Port D en PSP.

En mode PSP, le port D peut être lu ou écrit de façon asynchrone par un périphérique (microcontrôleur par exemple) externe à partir des broches $\overline{RD}$ et $\overline{WR}$. Le schéma correspondant est donné à figure ci-dessous.




**Note 1:** I/O pins have protection diodes to VDD and VSS.

### a) A quoi sert le mode PSP ?

Le mode PSP (pour Parallel Slave Port) permet à un microprocesseur, ou à tout autre système extérieur de prendre le contrôle du PORTD de votre PIC®. Ce système pourra écrire de lui-même des valeurs sur le PORTD, et y lire des valeurs que votre programme PIC® y aura préparées à son intention. Le PORTD devra donc passer alternativement en entrée et en sortie, et sous la seule décision du système extérieur.

C'est donc celui-ci qui décide à quel moment une lecture ou une écriture s'opère. Les transferts sont donc réalisés de façon asynchrone avec le programme de votre PIC®.

### a) Comment passer en MODE PSP ?

La première chose à faire est de forcer le bit PSPMODE du registre TRISE à « 1 ». Ensuite, configurer les pins RE0, RE1 et RE2 en entrée via le TRISE. Mais puisque c'est la valeur imposée par une mise sous tension, il est donc inutile de vous en occuper. Il faut noter également que le PORTD passe sous contrôle de ces pins, le registre TRISD n'est donc plus d'aucune utilité : inutile de le configurer.

Notez que dans la plupart des cas, on utilise les interruptions (interruption PSP) pour gérer ce mode de fonctionnement.

En effet, les échanges d'information se font sur décision du microprocesseur maître, et donc de façon asynchrone avec le programme du PIC®. Les interruptions restent la meilleure façon de traiter rapidement les événements asynchrones.

### c) Connexion des PORTD et PORTE en mode PSP.

La pin « CS » est la pin « Chip Select », elle permet de valider la sélection du circuit concerné, en l'occurrence notre PIC®. Sur la carte à microprocesseur, un détecteur d'adresse, placé sur le bus d'adresse permet de sélectionner l'adresse à laquelle va répondre notre PIC®. Comme dans la plupart des systèmes, ce « CS » sera actif à l'état bas.

Ensuite, nous trouvons les signaux « RD » pour « ReaD » et « WR » pour « WRite ». De nouveau ces signaux sont actifs à l'état bas.

Maintenant, concernant le PORD, il sera connecté au bus de data de la carte à microprocesseur. Notez, une fois de plus, que sur certains microprocesseurs, les bus de data et d'adresse peuvent être multiplexés, ou d'une largeur différente de 8 bits. Il vous incombe une fois de plus de rendre les signaux et les messages compatibles électroniquement.

### d) Le fonctionnement logiciel.

Maintenant que tout est configuré et que le PIC® est interfacé, voyons comment fonctionne l'ensemble au niveau software.

En fait, la procédure est simple, il existe 2 cas : soit le microprocesseur écrit dans le PIC®, soit il lit.

***Configuration d'une écriture***.

- ✓ Le microprocesseur exécute une écriture de DATA à l'adresse du PIC®.
- ✓ La DATA est mémorisée dans le tampon (latch) de lecture du PIC®.
- ✓ Le flag d'interruption PSPIF est positionné, ainsi que le bit IBF du registre TRISE.
- ✓ Si l'interruption est autorisée, une interruption est générée.
- ✓ Le programme lit le latch pour savoir quelle valeur a été écrite par le microprocesseur, ce qui provoque également le reset du bit IBF (Input Buffer Full = buffer d'entrée plein).

Si on regarde ce qui se passe au niveau du PIC®, on aura tout simplement :

- ✓ Le flag PSPIF est positionné, et une interruption a éventuellement lieu.
- ✓ Le programme agit pour récupérer la donnée écrite par le microprocesseur.
- ✓ Le test de IBF et OBF permet de savoir si on a eu affaire à une lecture ou à une écriture.

Notez que lorsqu'on lit le PORTD, on obtient non pas le niveau présent sur les pins RD0 à RD7, mais la valeur que le microprocesseur a écrite directement dans le tampon d'entrée.

Il se peut bien entendu que le microprocesseur écrive une seconde donnée dans le PORTD du PIC®, avant que celui-ci n'ait eu le temps de lire la donnée précédente. Dans ce cas, cette précédente donnée est perdue, et le bit IBOV est positionné pour signaler à l'utilisateur qu'un écrasement a eu lieu et qu'au moins une des données transmises est perdue.

***Configuration d'une lecture de donnée depuis un microprocesseur*** *:*

- ✓ Le PIC® écrit sa donnée dans le latch d'écriture, qui est également le PORTD.
- ✓ Ceci provoque le positionnement du bit OBF (Ouput Buffer Full = buffer de sortie plein).
- ✓ Quand le microprocesseur le décide, il effectue une lecture du PORTD, et récupère alors la valeur qui y est écrite.
- ✓ Le flag OBF passe à « 0 » dès que le cycle de lecture est commencé.
- ✓ Le flag PSPIF est positionné une fois la fin du cycle de lecture terminé.
- ✓ Une interruption est alors éventuellement générée si elle a été autorisée.

Si on regarde au niveau du PIC®, on aura tout simplement :

- ✓ On écrit la valeur à envoyer au processeur dans PORTD.
- ✓ On est prévenu que la valeur a été lue par le positionnement du flag PSPIF, et, éventuellement, par l'apparition de l'interruption correspondante.
- ✓ Le test de IBF et OBF permet de savoir si on a eu affaire à une lecture ou à une écriture.

Notez que le tampon de lecture possède la même adresse que le tampon d'écriture. Cette adresse est tout simplement le registre PORTD. Attention, ces tampons sont physiquement différents, donc en écrivant dans PORTD vous ne détruisez pas ce que le microprocesseur a écrit. Autrement dit, vous ne lisez pas ce que vous avez écrit, mais ce que le microprocesseur a écrit. Pour faire simple, quand vous écrivez dans PORTD, vous écrivez dans un registre différent de celui que vous accédez en lisant PORTD. Vous avez donc « 2 PORTD ». Remarquez donc que dans ce mode, l'écriture d'une valeur dans PORTD ne se traduit par l'apparition de cette valeur sur les pins RDx qu'au moment où le microprocesseur effectue une requête de lecture du PORTD. Le PORTD est donc sous le contrôle total du microprocesseur extérieur.

### e) Le fonctionnement matériel.

Nous nous plaçons du point de vue du maître, c'est-à-dire du microprocesseur. Il s'agit donc d'une écriture dans le PIC®, donc d'une lecture au niveau du programme du PIC®. Souvenez-vous également qu'un cycle d'instruction est composé de 4 clocks d'horloge, nommés Q1 à Q4. La fréquence de fonctionnement du PIC® est en effet égale à la fréquence de l'horloge divisée par 4.

Voici donc les événements hardware et leurs implications software du cycle d'écriture :

- ✓ La donnée est placée sur le PORTD par le microprocesseur (en connexion avec le bus de données) et doit rester stable jusqu'à la fin de l'ordre d'écriture.
- ✓ Au minimum 20 à 25 ns plus tard, les lignes de contrôle CS et WR sont placées à l'état bas (peu importe l'ordre).
- ✓ La ligne WR ou CS ou les deux repasse(nt) à 1, la data est capturée dans le latch d'entrée du PIC®.
- ✓ Le microprocesseur doit maintenir la donnée stable encore durant 20 à 35ns.
- ✓ Au temps Q4 qui suit, IBF et PSPIF passent simultanément à 1.

Quant au cycle de lecture (on suppose que la donnée se trouve déjà dans le latch de sortie du PIC®).

- ✓ Les lignes de contrôle RD et CS sont placées à l'état bas (peu importe l'ordre).
- ✓ A ce moment, le bit OBF passe à 0.
- ✓ Le PIC® place la donnée sur les lignes RD0 à RD7.
- ✓ Au minimum 80 à 90ns plus tard, le microprocesseur effectue la lecture de la donnée.
- ✓ Il remonte ensuite la ligne RD ou CS, ou les 2.
- ✓ La donnée disparaît des lignes RD0 à RD7 entre 10 et 30ns plus tard.
- ✓ Au cycle Q4 suivant du PIC®, le bit PSPIF est positionné, ce qui provoquera éventuellement une interruption.

Les chronogrammes correspondants sont donnés ci-dessous (extrait du datasheet).

**FIGURE 3-10: PARALLEL SLAVE PORT WRITE WAVEFORMS**

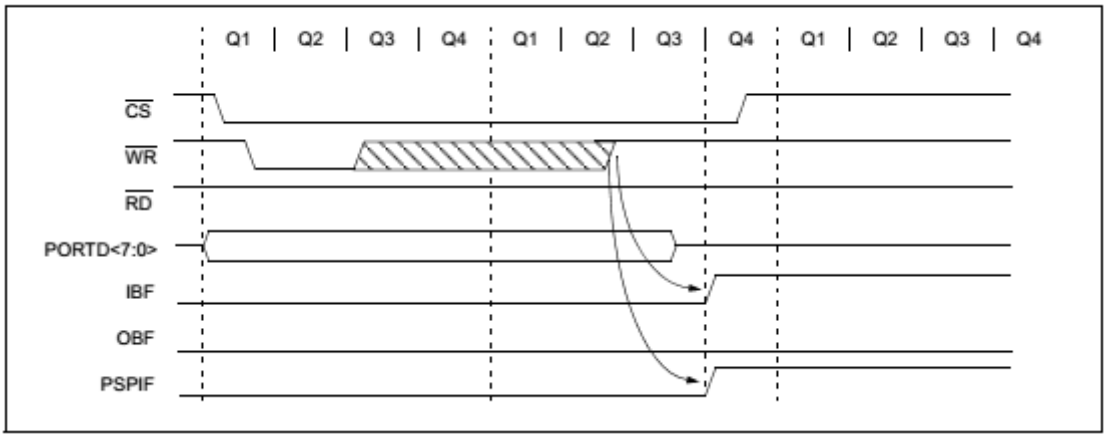
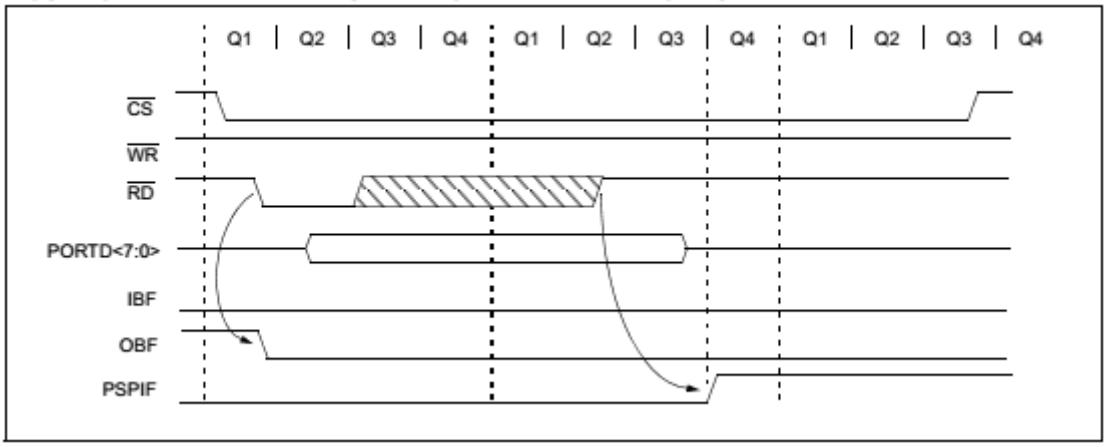



**FIGURE 3-11: PARALLEL SLAVE PORT READ WAVEFORMS**




Les registres associés avec le PSP sont donnés dans le tableau ci-dessous.

| Address | Name | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Value on: POR, BOR | Value on all other RESETS |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 08h | PORTD | Port Data Latch when written: Port pins when read | | | | | | | | xxxx xxxx | uuuu uuuu |
| 09h | PORTE | — | — | — | — | — | RE2 | RE1 | RE0 | ---- -xxx | ---- -uuu |
| 89h | TRISE | IBF | OBF | IBOV | PSPMODE | — | PORTE Data Direction Bits | | | 0000 -111 | 0000 -111 |
| 0Ch | PIR1 | PSPIF(1) | ADIF | RCIF | TXIF | SSPIF | CCP1IF | TMR2IF | TMR1IF | 0000 0000 | 0000 0000 |
| 8Ch | PIE1 | PSPIE(1) | ADIE | RCIE | TXIE | SSPIE | CCP1IE | TMR2IE | TMR1IE | 0000 0000 | 0000 0000 |
| 9Fh | ADCON1 | ADFM | — | — | — | PCFG3 | PCFG2 | PCFG1 | PCFG0 | --0- 0000 | --0- 0000 |

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used by the Parallel Slave Port.
**Note 1:** Bits PSPIE and PSPIF are reserved on the PIC16F873/876; always maintain these bits clear.

## V.5. Le module TMR0.

Les registres associés à la gestion du module TMR0 sont donnés dans le tableau ci-dessous.

| Address | Name | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Value on: POR, BOR | Value on all other RESETS |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 01h,101h | TMR0 | Timer0 Module's Register | | | | | | | | xxxx xxxx | uuuu uuuu |
| 0Bh,8Bh, 10Bh,18Bh | INTCON | GIE | PEIE | T0IE | INTE | RBIE | T0IF | INTF | RBIF | 0000 000x | 0000 000u |
| 81h,181h | OPTION_REG | RBPU | INTEDG | T0CS | T0SE | PSA | PS2 | PS1 | PS0 | 1111 1111 | 1111 1111 |

Legend: x = unknown, u = unchanged, - = unimplemented locations read as '0'.
Shaded cells are not used by Timer0.

### V.5.1. Les différents modes de fonctionnement.

Le timer 0 est en fait un compteur. Il y a deux possibilités :

- ✓ En premier lieu, il peut compter les impulsions reçues sur la pin RA4/TOKI. Nous dirons dans ce cas que nous sommes en mode compteur.
- ✓ Il peut aussi décider de compter les cycles d'horloge du PIC lui-même. Dans ce cas, comme l'horloge est fixe, nous compterons donc en réalité du temps. Donc, nous serons en mode « timer ».

La sélection de l'un ou l'autre de ces deux modes de fonctionnement s'effectue par le bit 5 du registre OPTION_REG : T0CS pour Tmr0 Clock Source select bit.

- ✓ T0CS = 1 : Fonctionnement en mode compteur.
- ✓ T0CS = 0 : Fonctionnement en mode timer.

Dans le cas où on décide de travailler en mode compteur, on doit aussi préciser lors de quelle transition de niveau le comptage est effectué. Ceci est précisé grâce au bit 4 du registre OPTION_REG : T0SE pour Timer0 Source Edge select bit.

- ✓ T0SE = 0 : comptage si l'entrée RA4/TOKI passe de 0 à 1.
- ✓ T0SE = 1 : comptage si l'entrée RA4/TOKI passe de 1 à 0.

### V.5.2. Le registre TMR0.

Ce registre, qui se localise à l'adresse 0x01 en banque0, contient tout simplement la valeur actuelle du timer0. Il est possible d'écrire ou de lire TMR0. Si par exemple tmr0 a été configuré en compteur, la lecture du registre TMR0 donnera le nombre d'événements survenus sur la pin RA4/TOKI.

## V.5.3. Les méthodes d'utilisation du timer 0.

### a) Le mode de lecture simple.

La première méthode qui vient à l'esprit est la suivante : Nous lisons le registre TMR0 pour voir ce qu'il contient. La valeur lue est le reflet du nombre d'événements survenus, en prenant garde au fait que le TMR0 ne peut compter que jusqu'à 255. En cas de dépassement, le TMR0 recommence à 0.

### b) Le mode de scrutation du flag.

Nous devons savoir à ce niveau, que tout débordement du timer 0 (passage de 0xFF à 0x00) entraîne le positionnement du flag T0IF du registre INTCON. On peut donc utiliser ce flag pour déterminer s'il y a eu débordement du timer 0, ou, en d'autres termes, si le temps programmé est écoulé.

### c) Le mode d'interruption.

C'est évidemment le mode principal d'utilisation du timer 0. En effet, lorsque T0IE est positionné dans le registre INTCON, chaque fois que le flag T0IF passe à 1, une interruption est générée.

### d) Les méthodes combinées.

Supposons qu'on veut, par exemple, mesurer un temps entre 2 impulsions sur la broche RB0. Supposons également que ce temps soit tel que plusieurs débordements du TMR0 puissent avoir lieu. Une méthode simple de mesure du temps serait la suivante :

- ✓ A la première impulsion sur RB0, on lance le timer 0 en mode interruptions.
- ✓ A chaque interruption de tmr0, on incrémente une variable.
- ✓ A la seconde interruption de RB0, on lit TMR0 et on arrête les interruptions.
- ✓ Le temps total sera donc (256*variable)+TMR0.

On a donc utilisé les interruptions pour les multiples de 256, et la lecture directe de TMR0 pour les « unités ».

## V.5.4. Le prédiviseur.

Supposons que nous travaillions avec un quartz de 4MHz. Nous avons donc dans ce cas (4000000/4) = 1.000.000 de cycles par seconde. Chaque cycle d'horloge dure donc 1/1000000ème de seconde, soit 1µs.

Si nous décidons d'utiliser le timer 0 dans sa fonction timer et en mode interruptions. Nous aurons donc une interruption toutes les 256µs, soit à peut près toutes les quarts de millième de seconde.

Si nous désirons réaliser une LED clignotante à une fréquence de 1Hz, nous aurons besoin d'une temporisation de 500ms, soit 2000 fois plus. Ce n'est donc pas pratique.

Nous disposons pour améliorer ceci d'un PREDIVISEUR. Qu'est-ce donc ? Et bien, tout simplement un diviseur d'événements situé AVANT l'entrée de comptage du timer 0. Nous pourrons donc décider d'avoir incrémentation de TMR0 tous les 2 événements par exemple, ou encore tous les 64 événements.

Le tableau des bits PS0 à PS2 du registre OPTION_REG déterminent la valeur du prédiviseur.

Ces valeurs varient, pour le timer 0, entre 2 et 256. Le bit PSA, quant à lui, détermine si le prédiviseur est affecté au timer 0 ou au watchdog.

Ci-dessous la description du registre OPTION_REG.

| R/W-1 | R/W-1 | R/W-1 | R/W-1 | R/W-1 | R/W-1 | R/W-1 | R/W-1 |
|---|---|---|---|---|---|---|---|
| $\overline{\text{RBPU}}$ | INTEDG | T0CS | T0SE | PSA | PS2 | PS1 | PS0 |
| bit 7 | | | | | | | bit 0 |

bit 7 **$\overline{\text{RBPU}}$**: PORTB Pull-up Enable bit
1 = PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled by individual port latch values

bit 6 **INTEDG**: Interrupt Edge Select bit
1 = Interrupt on rising edge of RB0/INT pin
0 = Interrupt on falling edge of RB0/INT pin

bit 5 **T0CS**: TMR0 Clock Source Select bit
1 = Transition on RA4/T0CKI/CMP2 pin
0 = Internal instruction cycle clock (CLKOUT)

bit 4 **T0SE**: TMR0 Source Edge Select bit
1 = Increment on high-to-low transition on RA4/T0CKI/CMP2 pin
0 = Increment on low-to-high transition on RA4/T0CKI/CMP2 pin

bit 3 **PSA**: Prescaler Assignment bit
1 = Prescaler is assigned to the WDT
0 = Prescaler is assigned to the Timer0 module

bit 2-0 **PS<2:0>**: Prescaler Rate Select bits

| Bit Value | TMR0 Rate | WDT Rate |
|---|---|---|
| 000 | 1 : 2 | 1 : 1 |
| 001 | 1 : 4 | 1 : 2 |
| 010 | 1 : 8 | 1 : 4 |
| 011 | 1 : 16 | 1 : 8 |
| 100 | 1 : 32 | 1 : 16 |
| 101 | 1 : 64 | 1 : 32 |
| 110 | 1 : 128 | 1 : 64 |
| 111 | 1 : 256 | 1 : 128 |

- ✓ $\overline{RBPU}$ : PORTB pull-up enable bit. Ce bit permet d'activer (0) ou non (1) l'utilisation d'une résistance de tirage.
- ✓ INTEDG : Interrupt Edge Select bit. Ce bit permet de sélectionner le front du signal qui peut provoquer une interruption sur la broche RB0. A 1 le front est montant, à 0 le front est descendant.

# V.6. Les sources d'interruption.

## V.6.1. Qu'est-ce qu'une interruption ?

Imaginez une conversation normale :

- ✓ Chaque interlocuteur prend la parole quand vient son tour de parler.
- ✓ Survient alors un événement extérieur dont le traitement est urgent. Par exemple, un piano tombe du 3ème étage de l'immeuble au pied duquel vous discutez.
- ✓ Vous imaginez bien que votre interlocuteur ne va pas attendre la fin de votre phrase pour vous signaler le danger. Il va donc vous INTERROMPRE durant le cours normal de votre conversation., afin de pouvoir TRAITER IMMEDIATEMENT l'EVENEMENT extérieur.
- ✓ Les interlocuteurs reprendront leur conversation où elle en était arrivée, sitôt le danger écarté (s'ils ont évité le piano, bien sûr).

Et bien, pour les programmes, c'est exactement le même principe :

- ✓ Votre programme se déroule normalement.
- ✓ Survient un événement spécifique.

- ✓ Le programme principal est interrompu (donc, subit une INTERRUPTION), et va traiter l'événement, avant de reprendre le programme principal à l'endroit où il avait été interrompu.

L'interruption est donc une RUPTURE DE SEQUENCE ASYNCHRONE, c'est-à-dire non synchronisée avec le déroulement normal du programme. Vous voyez ici l'opposition avec les ruptures de séquences synchrones, provoquées par le programme lui-même (goto, break…).

### V.6.2. Mécanisme général d'une interruption.

Nous pouvons dire, sans nous tromper de beaucoup, qu'une routine d'interruption est un sous-programme particulier, déclenché par l'apparition d'un événement spécifique.

Voici donc comment cela fonctionne :

- ✓ Le programme se déroule normalement.
- ✓ L'événement survient.
- ✓ Le programme achève l'instruction en cours de traitement.
- ✓ Le programme saute à l'adresse de traitement de l'interruption.
- ✓ Le programme traite l'interruption.
- ✓ Le programme saute à l'instruction qui suit la dernière exécutée dans le programme principal.

Il va bien sûr de soi que n'importe quel événement ne peut pas déclencher une interruption. Il faut que 2 conditions principales soient remplies :

- ✓ L'événement en question doit figurer dans la liste des événements susceptibles de provoquer une interruption pour le processeur sur lequel on travaille.
- ✓ L'utilisateur doit avoir autorisé l'interruption, c'est à dire doit avoir signalé que l'événement en question devait générer une interruption.

### V.6.3. Mécanisme d'interruption sur les PIC.

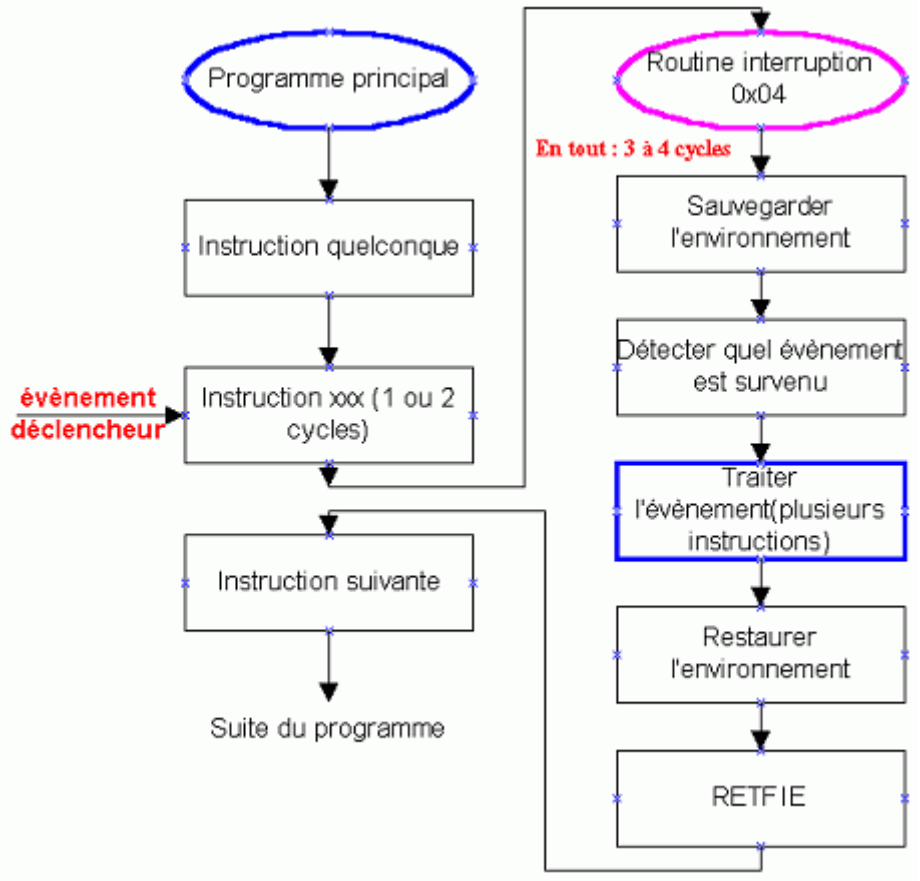
Bien entendu, les PIC répondent au fonctionnement général ci-dessus, mais ils ont également leurs particularités. Voyons maintenant le principe des interruptions sur les PIC.

- ✓ Tout d'abord, l'adresse de début de toute interruption est fixe. Il s'agit toujours de l'adresse 0x04.
- ✓ Toute interruption provoquera le saut du programme vers cette adresse. Toutes les sources d'interruption arrivant à cette adresse, si le programmeur utilise plusieurs sources d'interruptions, il lui faudra déterminer lui-même laquelle il est en train de traiter.
- ✓ Les PIC, en se connectant à cette adresse, ne sauvent rien automatiquement, hormis le contenu du PC, qui servira à connaître l'adresse du retour de l'interruption. C'est donc à l'utilisateur de se charger des sauvegardes.

***Remarque.***

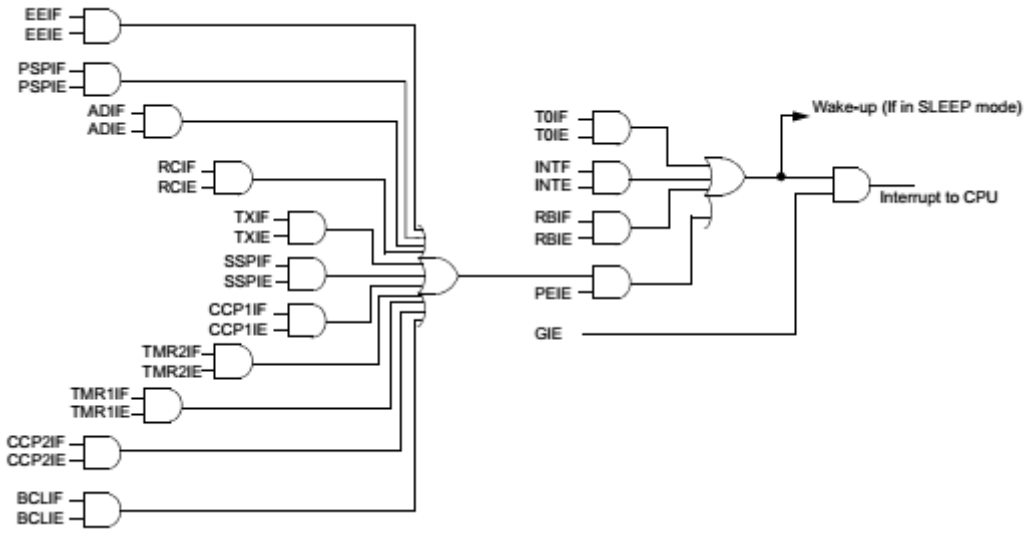
- ✓ Une interruption ne peut pas être interrompue par une autre interruption. Les interruptions sont donc invalidées automatiquement lors du saut à l'adresse 0x04 par l'effacement du bit GIE.
- ✓ Les interruptions sont remises en service automatiquement lors du retour de l'interruption.

***Algorigramme des interruptions sur les PIC.***




## V.6.4. Les sources d'interruptions.

Le PIC16F877A comporte 14 sources d'interruption comme indiquées ci-dessous.




Voici un tableau récapitulatif des 13 interruptions disponibles sur le 16F876. Notez que le 16F877/4 dispose d'un port parallèle supplémentaire qui lui autorise une source d'interruption en plus. Sur ces composants, nous aurons donc 14 sources d'interruption.

| Déclencheur | Flag | Registre | Adr | PEIE | Enable | Registre | Adr |
|---|---|---|---|---|---|---|---|
| Timer 0 | T0IF | INTCON | 0x0B | NON | T0IE | INTCON | 0x0B |
| Pin RB0 / INT | INTF | INTCON | 0x0B | NON | INTE | INTCON | 0x0B |
| Ch. RB4/RB7 | RBIF | INTCON | 0x0B | NON | RBIE | INTCON | 0x0B |
| Convert. A/D | ADIF | PIR1 | 0x0C | OUI | ADIE | PIE1 | 0x8C |
| Rx USART | RCIF | PIR1 | 0x0C | OUI | RCIE | PIE1 | 0x8C |
| Tx USART | TXIF | PIR1 | 0x0C | OUI | TXIE | PIE1 | 0x8C |
| Port série SSP | SSPIF | PIR1 | 0x0C | OUI | SSPIE | PIE1 | 0x8C |
| Module CCP1 | CCP1IF | PIR1 | 0x0C | OUI | CCP1IE | PIE1 | 0x8C |
| Module CCP2 | CCP2IF | PIR2 | 0x0D | OUI | CCP2IE | PIE2 | 0x8D |
| Timer 1 | TMR1IF | PIR1 | 0x0C | OUI | TMR1IE | PIE1 | 0x8C |
| Timer 2 | TMR2IF | PIR1 | 0x0C | OUI | TMR2IE | PIE1 | 0x8C |
| EEPROM | EEIF | PIR2 | 0x0D | OUI | EEIE | PIE2 | 0x8D |
| SSP mode I2C | BCLIF | PIR2 | 0x0D | OUI | BCLIE | PIE2 | 0x8D |
| Port parallèle | PSPIF | PIR1 | 0x0C | OUI | PSPIE | PIE1 | 0x8C |

***Explication des différentes colonnes, de gauche à droite :***

- ✓ *Déclencheur* : Evénement ou fonction qui est la source de l'interruption.
- ✓ *Flag* : Bit qui se trouve positionné lorsque l'événement survient.
- ✓ *Registre* : Registre auquel le flag appartient.
- ✓ *Adr* : Adresse de ce registre (tous en banque 0, avec INTCON présent dans les 4 banques).
- ✓ *PEIE* : Indique si le positionnement de PEIE du registre INTCON joue un rôle.
- ✓ *Enable* : nom du bit qui permet d'autoriser ou non l'interruption.
- ✓ *Registre* : Registre qui contient ce bit.
- ✓ *Adr* : adresse de ce registre (tous en banque1, avec INTCON dans les 4 banques).

## V.6.5. Les registres de contrôle des interruptions.

### a) Le registre INTCON.

**REGISTER 4-3: INTCON – INTERRUPT CONTROL REGISTER (ADDRESS: 0Bh, 8Bh, 10Bh, 18Bh)**

| R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-x |
|---|---|---|---|---|---|---|---|
| GIE | PEIE | T0IE | INTE | RBIE | T0IF | INTF | RBIF |
| bit 7 | | | | | | | bit 0 |

bit 7 **GIE:** Global Interrupt Enable bit
1 = Enables all un-masked interrupts
0 = Disables all interrupts

bit 6 **PEIE**: Peripheral Interrupt Enable bit
1 = Enables all un-masked peripheral interrupts
0 = Disables all peripheral interrupts

bit 5 **T0IE**: TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 interrupt
0 = Disables the TMR0 interrupt

bit 4 **INTE**: RB0/INT External Interrupt Enable bit
1 = Enables the RB0/INT external interrupt
0 = Disables the RB0/INT external interrupt

bit 3 **RBIE**: RB Port Change Interrupt Enable bit
1 = Enables the RB port change interrupt
0 = Disables the RB port change interrupt

bit 2 **T0IF**: TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

bit 1 **INTF**: RB0/INT External Interrupt Flag bit
1 = The RB0/INT external interrupt occurred (must be cleared in software)
0 = The RB0/INT external interrupt did not occur

bit 0 **RBIF**: RB Port Change Interrupt Flag bit
1 = When at least one of the RB<7:4> pins changes state (must be cleared in software)
0 = None of the RB<7:4> pins have changed state

1. ***b7 : GIE.*** Il permet de valider ou d'invalider toutes les interruptions d'une seule fois.
2. ***b6 : PEIE.*** Il permet de valider ou d'invalider les interruptions périphériques.
3. ***b5 : T0IE***. Valide l'interruption générée par le débordement du timer0.
4. ***b4 : INTE***. Valide l'interruption dans le cas d'une modification de niveau de la pin RB0. Rappelons le bit 6 du registre OPTION_REG, qui détermine quel est le sens de transition qui provoque l'interruption. On pourra donc choisir si c'est une transition 0→1 ou 1→0 qui provoque l'interruption, mais pas les deux ensemble.
5. ***b3 : RBIE***. Valide les interruptions si il y a changement de niveau sur une des entrées RB4 à RB7.
6. ***b2 : T0IF***. C'est un Flag, donc il signale. Ici c'est le débordement du timer0.
7. ***b1 : INTF***. Signale une transition sur la pin RB0 dans le sens déterminé par INTEDG du registre OPTION (b6).
8. ***b0 : RBIF***. Signale qu'une des entrées RB4 à RB7 a été modifiée. Rappelons que les flags ne se remettent pas à 0 tout seul. C'est le programme qui doit s'en charger, sous peine de rester indéfiniment bloqué dans une routine d'interruption. Nous dirons que ces flags sont des FLAGS REMANENTS.

**Remarquez**.

1. Tous les bits dont le nom se termine par E (Enable) sont en fait des commutateurs de validation.
2. Les bits dont le nom se termine par F sont des Flags (indicateurs).

### b) Les registres PIE1, PIR1, PIE2, PIR2.

Voici le nom de chaque bit de ces registres.

| Registre | Adresse | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 |
|---|---|---|---|---|---|---|---|---|---|
| PIE1 | 0x8C | PSPIE | ADIE | RCIE | TXIE | SSPIE | CCP1IE | TMR2IE | TMR1IE |
| PIR1 | 0x0C | PSPIF | ADIF | RCIF | TXIF | SSPIF | CCP1IF | TMR2IF | TMR1IF |
| PIE2 | 0x8D | N.I. | Réservé | N.I. | EEIE | BCLIE | N.I. | N.I. | CCP2IE |
| PIR2 | 0x0D | N.I. | Réservé | N.I. | EEIF | BCLIF | N.I. | N.I. | CCP2IF |

Remarquez qu'il y a quelques bits non implémentés, qui donneront toujours « 0 » s'ils sont lus, mais aussi 2 bits réservés qu'il est impératif de maintenir à « 0 » afin de conserver toute compatibilité future.

Les bits des registres PIEx permettent d'autoriser les interruptions correspondantes, mais rappelez-vous que ces bits ne sont opérationnels que si le bit PEIE du registre INTCON est mis à 1. Dans le cas contraire, toutes ces interruptions sont invalidées.

Pour qu'une des interruptions décrites ici soit opérationnelle, je rappelle qu'il faut donc la triple condition suivante :

- ✓ Le bit GIE du registre INTCON doit être mis à 1.
- ✓ Le bit PEIE du registre INTCON doit être mis à 1.
- ✓ Le bit correspondant à l'interruption concernée (registre PIE1 ou PIE2) doit être mis à 1.

Voici maintenant le rôle de chacun des bits d'autorisation d'interruption, sachant qu'un bit à «1» autorise l'interruption correspondante :

**PIE1**

1. ***PSPIE***. Lecture/écriture sur le port PSP.
2. ***ADIE***. Conversion analogique/digitale.
3. ***RCIE***. Réception d'un caractère sur le port série USART.
4. ***TXIE***. Emission d'un caractère sur le port série USART.
5. ***SSPIE***. Communication sur le port série synchrone SSP.
6. ***CCP1IE***. Evénement sur compare/capture registre 1.
7. ***TMR2IE***. Correspondance de valeurs pour le timer TMR2.
8. ***TMR1IE***. Débordement du timer TMR1.

**PIE2**.

1. ***EEIE***. Ecriture dans l'EEPROM.
2. ***BCLIE***. Collision de bus pour le port série synchrone $I^2C$.
3. ***CCP2IE***. Evénement sur compare/capture registre 2.

Et maintenant, les flags correspondants.

**PIR1.**

1. ***PSPIF***. Lecture ou écriture terminée sur le port parallèle.
2. ***ADIF***. Fin de la conversion analogique/digitale.
3. ***RCIF***. Le buffer de réception de l'USART est plein (lecture seule).
4. ***TXIF***. Le buffer d'émission de l'USART est vide (lecture seule).
5. ***SSPIF***. Fin de l'événement dépendant du mode de fonctionnement comme suit :
   - ✓ *Mode SPI*. Un caractère a été envoyé ou reçu.
   - ✓ *Mode $I^2C$ esclave*. Un caractère a été envoyé ou reçu.
   - ✓ *Mode $I^2C$ maître*. Un caractère a été envoyé ou reçu
     ou fin de la séquence « start »
     ou fin de la séquence « stop »
     ou fin de la séquence « restart »
     ou fin de la séquence « acknowledge »
     ou « start » détecté durant IDLE (multimaître)
     ou « stop » détecté durant IDLE (multimaître)
6. ***CCP1IF***. Evénement compare/capture 1 détecté suivant :
   - ✓ *Mode capture* : capture de la valeur TMR1 réalisée.
   - ✓ *Mode compare* : La valeur de TMR1 atteint la valeur programmée.
7. ***TMR2IF***. La valeur de TMR2 atteint la valeur programmée.
8. ***TMR1IF***. Débordement du timer TMR1.

**PIR2**.

1. ***EEIF***. Fin d'écriture de la valeur en EEPROM.
2. ***BCLIF***. Collision de bus, quand le SSP est configuré en maître $I^2C$.
3. ***CCP2IF***. Evénement compare/capture 2 détecté suivant le mode :
   a. *Mode capture* : capture du TMR1 réalisée.
   b. *Mode compare* : La valeur de TMR1 atteint la valeur programmée.

## V.6.6. Mise en service des interruptions primaires.

Des constatations précédentes, nous pouvons dire que nous disposons de 3 interruptions primaires. Pour mettre en service une de ces interruptions, il faut donc :

1. Valider le bit concernant cette interruption.
2. Valider le bit GIE (General interrupt enable) qui met toutes les interruptions choisies en service.

## V.6.7. Mise en service des interruptions périphériques.

Les interruptions périphériques sont tributaires du bit de validation générale des interruptions périphériques PEIE. Voici donc les 3 étapes nécessaires à la mise en service d'une telle interruption :

1. Validation du bit concernant l'interruption dans le registre concerné (PIE1 ou PIE2).
2. Validation du bit PEIE (PEripheral Interrupt Enable bit) du registre INTCON.
3. Validation du bit GIE qui met toutes les interruptions en service.

Notez donc que la mise en service de PEIE ne vous dispense pas l'initialisation du bit GIE, qui reste de toute façon prioritaire. Cette architecture vous permet donc de couper toutes les interruptions d'un coup (effacement de GIE) ou de couper toutes les interruptions périphériques en une seule opération (effacement de PEIE) tout en conservant les interruptions primaires. Ceci vous donne d'avantage de souplesse dans le traitement des interruptions, mais complique un petit peu le traitement.

# V.7. Convertisseur analogique numérique (CAN).

## V.7.1. Généralité.

### a) Principe.

Qu'est-ce qu'un Convertisseur Analogique/Digital (CAD), de préférence nommé Convertisseur Analogique/Numérique (CAN), ou Analogic to Digital Converter (ADC) ?

En fait, nous avons vu jusqu'à présent que nous pouvions entrer un signal sur les pins du PIC®, qui déterminait, en fonction du niveau de tension présente, si ce signal était considéré comme un « 1 » ou un « 0 » logique. Ceci est suffisant pour tout signal binaire, c'est-à-dire ne présentant que 2 valeurs possibles.

Supposons que vous désiriez, avec votre PIC®, mesurer une valeur analogique, par exemple si vous voulez mesurer la tension de votre batterie. Comme l'électronique interne du PIC® ne comprend que les valeurs binaires, il vous faudra donc transformer cette valeur analogique en une représentation numérique. Ce procédé s'appelle numérisation, et, pour l'effectuer, vous avez besoin d'un convertisseur analogique/numérique.

Vous effectuez couramment, et sans le savoir, des conversions analogiques/numériques. En effet, lorsque vous décidez d'arrondir des valeurs, vous transformez une valeur analogique (qui peut donc prendre une infinité de valeur) en une valeur numérique (qui contient un nombre fini d'éléments).

En effet, prenons le nombre 128,135132. Si vous décidez d'arrondir ce nombre en conservant 4 digits, vous direz que ce nombre « numérisé » devient 128,1. Vous voyez donc qu'en numérisant vous perdez de la précision, puisque vous éliminez de fait les valeurs intermédiaires.

Avec cet exemple, vous pouvez représenter les valeurs 128,1 et 128,2, mais toute autre valeur intermédiaire sera transformée en un de ces deux nombres discrets. La précision obtenue dépend donc du nombre de digits que vous souhaitez conserver pour le résultat, lequel sera entaché d'une erreur.

Que vaut cette erreur, notée E ? En fait, vous voyez que 128,13 sera converti en 128,1, tandis que 128,16 sera converti en 128,2. L'erreur maximale obtenue est donc de la moitié du plus faible digit (appelé quantum ou pas de quantification, noté q).

$$E = \frac{q}{2}$$

Comme notre digit vaut 0,1, l'erreur finale maximale sera donc dans notre exemple de 0,05. Si vous convertissez maintenant dans l'autre sens, vous pouvez dire que votre nombre numérisé de 128,1 représente en réalité une grandeur analogique réelle comprise entre 128,05 et 128,15. La valeur moyenne étant de 128,1, ce qui est logique.

La constatation est qu'à une seule valeur numérique correspond une infinité de valeurs analogiques dans un intervalle bien défini.

Supposons que nous voulions numériser une valeur analogique comprise entre 0 et 90 en une valeur numérique codée sur 1 digit décimal. Nous voyons tout de suite que la valeur analogique 0 sera traduite en D'0', la valeur analogique 10 sera traduite en D'1', etc. jusque la valeur 90 qui sera convertie en D'9'.

Si maintenant notre valeur analogique varie entre 10 et 100, la valeur analogique 10 sera convertie en D'0', la valeur analogique 20 sera convertie en D'1', etc. jusque la valeur analogique 100 qui sera convertie en D'9'.

Donc, puisque nous ne disposons ici que d'un digit, qui peut prendre 10 valeurs, de 0 à 9 pour traduire une valeur qui peut varier d'un minimum à un maximum, on peut tirer des petites formules. On peut dire que chaque digit numérique (le pas p) représente la plage de valeurs de la grandeur analogique (Amax-Amin) divisée par la différence entre la plus grande (Nmax) et la plus petite (Nmin) valeur possible représentée par notre nombre numérisé.

$$p = \frac{A_{max} - A_{min}}{N_{max} - N_{min}}$$

Donc, pour prendre notre premier exemple, 1 digit numérisé représente la valeur maximale analogique moins la valeur minimale analogique divisé par la valeur maximale numérique. Donc : (90-00)/9 = 10. Vous voyez que « 1 » en numérique représente la valeur analogique 10.

Notre grandeur numérique mesure donc les dizaines. Le pas de notre conversion est de 10. Si nous prenons notre second exemple, nous aurons : (100-10)/9 = 10 également. J'ai choisi ces exemples car ils étaient simples.

Si maintenant, nous désirons effectuer l'opération inverse, c'est-à-dire savoir à quelle valeur analogique correspond une valeur numérique, nous pourrons dire que : La valeur analogique typique est égale à la valeur analogique minimale représentée à laquelle on ajoute le pas multiplié par la valeur numérique correspondante.

$$A = A_{min} + p \times N$$

C'est très simple à comprendre. Supposons que nous ayons la valeur numérique 5. Que représente-t-elle comme grandeur analogique réelle dans notre premier exemple ? Et bien, tout simplement 0 + (5*10) = 50. Ca semble logique, non ?

Quand à notre second exemple, cette même valeur représente une grandeur analogique de : 10 + (5*10) = 60. C'est tout aussi logique.

Voici un graphique correspondant à notre premier exemple, pour vous permettre d'y voir clair. N'oubliez pas que nous arrondissons (convertissons) toujours au niveau du demi digit. En rouge vous avez toutes les valeurs analogiques possibles de 0 à 90, tandis qu'en bleu, vous avez les seules 10 valeurs numériques correspondantes possibles.


Vous constatez que la courbe analogique peut prendre n'importe quelle valeur, tandis que la courbe numérique ne peut prendre qu'une des 10 valeurs qui lui sont permises.

Par exemple, la valeur 16 existe sur la courbe analogique, mais la valeur 1,6 n'existe pas sur la courbe numérique. Chaque palier de cette courbe représente 1 digit (échelle bleue) et un pas de 10 sur l'échelle rouge analogique.

Nous voyons également que pour certaines valeurs analogiques (par exemple 15), nous avons 2 valeurs numériques possibles. La même situation se retrouve dans la vie réelle. Si vous donnez 10.000GNF à un commerçant pour payer un montant de 7.565GNF, libre à lui de considérer que vous lui devez 7.600GNF plutôt que 7.500GNF. Je pense qu'un juriste aura grand mal à vous départager, et d'ailleurs je doute que vous portiez l'affaire en justice.

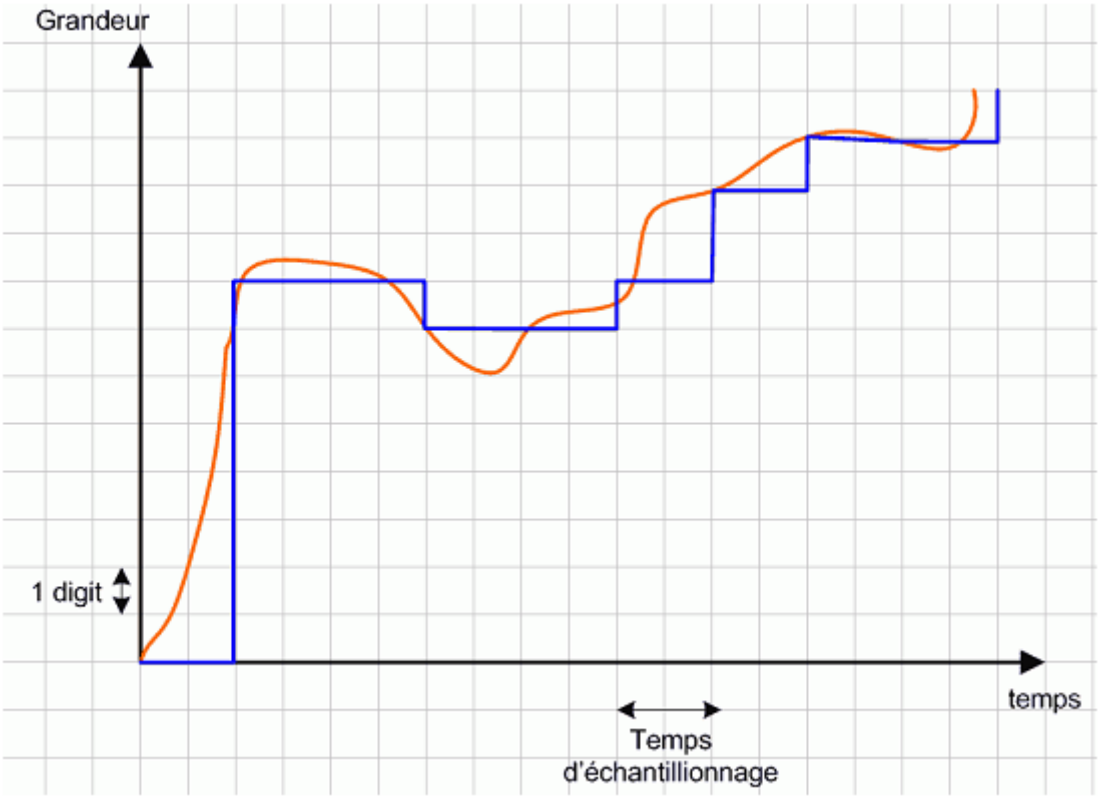
Nous venons de voir que nous avons effectué une conversion d'une grandeur analogique en grandeur numérique et réciproquement pour une grandeur que nous avons supposée fixe dans le temps.

Mais que se passerait-il pour une grandeur qui varie, comme par exemple un signal audio ? En fait, de nouveau, ce signal va varier dans le temps de façon continue. Le PIC®, lui (ou tout autre convertisseur existant), va effectuer à intervalles réguliers des mesures du signal pour le convertir en valeurs numériques successives. Donc, de nouveau, notre échelle de temps ne sera plus continue, mais constituée de bonds.

C'est comme si, par exemple, vous preniez une série de photos successives avec un appareil rapide. Lorsque vous visualisez l'animation, vous aurez une succession d'images fixes qui restitueront l'animation.

Plus les photos sont rapprochées dans le temps, plus vous vous rapprochez de la réalité, et moins vous perdez d'événements. Vous passez progressivement de l'appareil photo à la caméra, mais cette dernière travaille également sur le même principe.

**Corollaire :** Plus les événements à filmer sont rapides, et plus vos photos devront être rapprochées pour ne pas perdre des événements. C'est exactement le même principe pour la numérisation de signaux variables. En fait, vous réalisez une double numérisation. La première consiste, comme nous l'avons vu plus haut, à découper la valeur en une succession de tranches, la seconde consiste à découper le temps en une autre succession de tranches. Voici ce que ça donne pour un signal quelconque.




Vous voyez que vous perdez 2 fois en précision. D'une part votre valeur est arrondie en fonction du nombre de digits utilisés pour la conversion, et d'autre part, tous les événements survenus entre 2 conversions (échantillonnages) sont perdus. Vous pouvez en déduire que :

1. Plus vous désirez de précision, plus vous devez augmenter le nombre de digits utilisés pour le résultat.
2. Plus votre signal évolue rapidement, plus vous devez diminuer le temps séparant 2 échantillonnages, autrement dit, augmenter votre vitesse d'échantillonnage.

Je vous passerai les théories de Fourrier et autres, pour simplement vous dire que pour un signal sinusoïdal, on admet que la fréquence d'échantillonnage doit être supérieure au double de la fréquence du signal à mesurer.

$$F_{ech} \geq 2f_{max}$$

Pour vous donner un exemple, lorsqu'on convertit un signal audio pour en faire un signal numérique destiné à être placé sur un disque CD, les caractéristiques de la conversion sont les suivantes :

1. Echantillonnage sur 16 bits, soit 65536 valeurs numériques différentes : on se rapproche donc énormément du signal analogique original.
2. Fréquence d'échantillonnage de 44,100 kHz, ce qui nous donne, suite au théorème précédent, une fréquence maximale sinusoïdale digitalisée 22 KHz s'il n'y avait pas de filtrage qui réduit un peu cette fréquence (l'oreille humaine parvient en général à capter des fréquences maximales de 16KHz, 20Khz pour certaines personnes).

### b) La conversion avec les PIC16F877.

Jusqu'à présent, nous venons de raisonner en décimal. Les PIC®, eux travaillent en binaire. Mais, rassurez-vous, ceci reste strictement identique. Souvenez-vous que lorsqu'on change de base de numérotation, les formules restent toutes d'application.

Notre 16F877 travaille avec un convertisseur analogique / numérique qui permet un échantillonnage sur 10 bits. Le signal numérique peut donc prendre 1024 valeurs possibles. Vous avez vu que pour pouvoir convertir une grandeur, nous devons connaître la valeur minimale qu'elle peut prendre, ainsi que sa valeur maximale. Les PIC® considèrent par défaut que la valeur minimale correspond à leur Vss d'alimentation, tandis que la valeur maximale correspond à la tension positive d'alimentation Vdd. Nous verrons cependant qu'il est possible d'utiliser d'autres valeurs.

Nous n'avons toujours pas parlé des méthodes utilisées pour convertir physiquement la grandeur analogique en grandeur numérique au cœur du PIC®. Il est inutile d'entrer ici dans un cours d'électronique appliquée, mais il est bon de connaître le principe utilisé, car cela va vous aider à comprendre la suite. La séquence est la suivante :

1. Le PIC® connecte la pin sur laquelle se trouve la tension à numériser à un condensateur interne, qui va se charger via une résistance interne jusque la tension appliquée.
2. La pin est déconnectée du condensateur, et ce dernier est connecté sur le convertisseur analogique/numérique interne.
3. Le PIC® procède à la conversion.

Plusieurs remarques et questions sont soulevées par cette procédure. En tout premier lieu, le condensateur va mettre un certain temps à se charger, il nous faut donc connaître ce temps. Ensuite, il nous faut comprendre comment fonctionne la conversion, pour évaluer le temps mis pour cette conversion. Ceci nous donnera le temps total nécessaire, afin de savoir quelle est la fréquence maximale d'échantillonnage pour notre PIC®.

Remarquez que si le signal varie après le temps de charge du condensateur interne, cette variation ne sera pas prise en compte, puisque la pin sera déconnectée du dit condensateur.

### c) Le temps d'acquisition.

C'est le temps qu'il faut pour que le condensateur interne atteigne une tension proche de la tension à convertir. Cette charge s'effectue à travers une résistance interne et la résistance de la source connectée à la pin.

Ce temps est incrémenté du temps de réaction des circuits internes, et d'un temps qui dépend de la température (coefficient de température). Il faut savoir en effet que les résistances augmentent avec la température, donc les temps de réaction des circuits également.

Donc, si on pose :

- ✓ Tacq = temps d'acquisition total ;
- ✓ Tamp = temps de réaction des circuits ;
- ✓ Tc = temps de charge du condensateur ;
- ✓ Tcoff = temps qui dépend du coefficient de température.

La formule est donc :

$$Tacq = Tamp + Tc + Tcoff.$$

Le temps de réaction Tamp est typiquement de 2µs, pas donc de problème à ce niveau. Pour le coefficient de température, il n'est nécessaire que pour les températures supérieures à 25°C. Dans les autres cas, il n'entre pas en compte. Ce coefficient est typiquement de 0,05 µs par °C qui est supérieur à 25°C. Il s'agit bien entendu de la t° du PIC®, et non de la température ambiante. Donc, ce temps Tcoff sera au minimum de 0 (à moins de 25°C) et au maximum de (50 - 25)*0.05, soit 1,25 µs.

La température du PIC® ne pouvant pas, en effet, excéder 50°C. $0 \leq Tcoff \leq 1{,}25\mu s$.

Première constatation, si vous voulez bénéficier d'une fréquence maximale, vous devez maintenir le PIC® sous 25°C.

Reste le temps de charge. Ce temps de charge dépend de la résistance placée en série avec le condensateur. En fait, il y a 2 résistances, celle de votre source de signal, et celle à l'intérieur du PIC®. Il est recommandé que la résistance de votre source reste inférieure à 10KOhms. Celle interne au PIC® est directement liée à la tension d'alimentation.

Plus la tension baisse, plus la résistance est élevée, donc plus le temps de chargement est long. Donc, de nouveau, pour obtenir de hautes vitesses, il vous faudra alimenter le PIC® avec la tension maximale supportée, soit 6V à l'heure actuelle pour le 16F877. La résistance interne totale (composée de 2 résistances internes) varie de 6Kohms à 6V pour arriver à 12Kohms sous 3V, en passant par 8Kohms sous 5V.

De plus, comme la charge du condensateur dépend également de la résistance de la source du signal, pour augmenter votre vitesse, vous devez également utiliser une source de signal présentant la plus faible impédance (résistance) possible. Sachant que le condensateur interne a une valeur de 120pF pour les versions actuelles de PIC® (16F876), la formule du temps de charge du condensateur est :

$$T_c = -C(R_{int} + R_{ext}) \times \ln\frac{1}{2047} = 0{,}914895 \times 10^{-9}$$

Le 2047 provient de ce que pour numériser avec une précision de ½ bit, la numérisation utilisant une valeur maximale de 1023, la charge du condensateur doit être au minimum de 2046/2047ème de la tension à mesurer.

Si on se place dans le cas le plus défavorable (tension de 3V, et résistance source = 10Kohms), notre temps de chargement est de :

Tcmaximal = 20,12 µs.

Maintenant le cas le plus favorable (tension de 6V, et résistance source négligeable) :

Tc minimal : 5,48 µs.

Si, maintenant, nous prenons un cas typique, à savoir une tension d'alimentation de 5V et une résistance de source de 10 Kohms, nous aurons : Tc typique = 16,46 µs.

Nous allons maintenant calculez les temps minimum, maximum, et typique du temps total d'acquisition Tacq.

- ✓ Le cas le plus défavorable est : une température de 50°C et un Tc maximal, ce qui nous donne : Tacq = Tamp + Tc + Tcoff ; Tacq maximum = 2µs + 20,12µs + 1,25µs = 23,37 µs.
- ✓ Le cas le plus favorable, une température inférieure ou égale à 25°C et un Tc minimal, nous donne : Tacq minimum = 2µs + 5,48µs = 7,48 µs.
- ✓ Maintenant, pour nos utilisations classiques, sous 5V, nous aurons dans le pire des cas : Tacq sous 5V = 2µs + 16,46µs + 1,25µs = 19,71µs.

Donc, nous prendrons un Tacq de 20µs pour notre PIC® alimentée sous 5V. Mais vous devez vous souvenir que si vous travaillez sous une tension différente, il vous faudra adapter ces valeurs. De même, si vous avez besoin de la plus grande vitesse possible dans votre cas particulier, vous possédez maintenant la méthode vous permettant de calculer votre propre Tacq. Pour ma part, dans la suite de ce cours, je travaillerai avec la valeur standard de 20 µs. Remarquez que ces valeurs sont données telles quelles dans les datasheets.

Je vous ai démontré mathématiquement d'où provenaient ces valeurs. Ceci vous permettra de connaître les temps nécessaires pour votre application particulière, et ainsi, vous autorisera la plus grande vitesse possible.

### d) La conversion.

Arrivé à ce stade, après le temps Tacq, on peut considérer que le condensateur est chargé et prêt à être connecté sur l'entrée du convertisseur analogique/digital. Cette connexion prend de l'ordre de 100ns. Une fois le condensateur connecté, et donc, la tension à numériser présente sur l'entrée du convertisseur, ce dernier va devoir procéder à la conversion. Je ne vais pas entrer ici dans les détails électroniques de cette conversion, mais sachez que le principe utilisé est celui de l'approximation successive.

Le temps nécessaire à la conversion (Tc) est égal au temps nécessaire à la conversion d'un bit (Tc1) multiplié par le nombre de bits (N) désirés pour le résultat.

$$Tc = Tc1.N$$

Concernant notre PIC®, il faut savoir qu'il nécessite, pour la conversion d'un bit, un temps qu'on va nommer Tad. Ce temps est dérivé par division de l'horloge principale. Le diviseur peut prendre une valeur de 2, 8 ou 32.

Attention, on divise ici l'horloge principale, et non le compteur d'instructions. Donc, une division par 2 signifie un temps 2 fois plus court que celui nécessaire pour exécuter une instruction, puisque ce temps d'exécution est de Tosc/4.

Il est également possible d'utiliser une horloge constituée d'un oscillateur interne de type RC. Cet oscillateur donne un temps de conversion compris entre 2 et 6µs, avec une valeur typique de 4µs. Pour les versions LC du 16F877, ce temps passe entre 3 et 9µs. Si la fréquence du PIC® est supérieure à 1Mhz, vous ne pourrez cependant l'utiliser qu'en cas de mise en sommeil du PIC® dans l'attente du temps de conversion.

En effet, durant le mode « sleep », l'horloge principale est stoppée, donc seul cet oscillateur permettra de poursuivre la conversion. Cependant, je le rappelle encore une fois, pour votre PIC® tournant à plus de 1Mhz, vous êtes contraint en utilisant cette horloge de placer votre PIC® en mode sleep jusqu'à la fin de la conversion. Dans le cas contraire, le résultat serait erroné.

Je vous conseille donc de n'utiliser cette méthode que si vous désirez placer votre PIC® en mode sleep durant la conversion, ou si vous utilisez une fréquence de PIC® très basse, et de toute façon sous les 1MHz.

Le temps de conversion Tad ne peut descendre, pour des raisons électroniques, en dessous de 1,6µs pour les versions classiques de 16F877, et en dessous de 6µs pour les versions LC. Donc, en fonction des fréquences utilisées pour le quartz du PIC®, il vous faudra choisir le diviseur le plus approprié.

Voici un tableau qui reprend les valeurs de diviseur à utiliser pour quelques fréquences courantes du quartz et pour les PIC® de type classique.

| Diviseur | 20Mhz | 5Mhz | 4Mhz | 2Mhz | 1,25Mhz | 333,3Khz |
|---|---|---|---|---|---|---|
| 2 | 100ns | 400ns | 500ns | 1µs | 1,6µs | 6µs |
| 8 | 400ns | 1,6µs | 2µs | 4µs | 6,4µs | 24µs |
| 32 | 1,6µs | 6,4µs | 8µs | 16µs | 25,6µs | 96µs |
| Osc RC | 2-6µs | 2-6µs | 2-6µs | 2-6µs | 2-6µs | 2-6µs |

La formule d'obtention des temps Tad est simple, puisqu'il s'agit tout simplement, du temps d'instruction (Tosc) divisé par le diviseur donné.

Exemple, à 20Mz, le temps d'instruction est de 1/20.000.000, soit 50ns. Donc, avec un diviseur de 2, on aura 100ns.

- ✓ Les valeurs en vert sont celles qui correspondent au meilleur diviseur en fonction de la fréquence choisie.
- ✓ Les valeurs en bleu sont inutilisables, car le temps Tad serait inférieur à 1,6µs.
- ✓ Quant aux valeurs en jaune, je rappelle que pour l'utilisation de l'oscillateur interne RC à ces fréquences, la mise en sommeil du PIC® est impératif durant le temps de conversion.

Vous remarquez que, du à ces diviseurs, il est possible, par exemple de numériser plus vite avec un PIC® tournant à 1,25 Mhz qu'avec le même PIC® muni d'un quartz à 4 Mhz.

Il faut à présent préciser que le PIC® nécessite un temps Tad avant le démarrage effectif de la conversion, et un temps supplémentaire Tad à la fin de la conversion. Donc, le temps total de conversion est de :

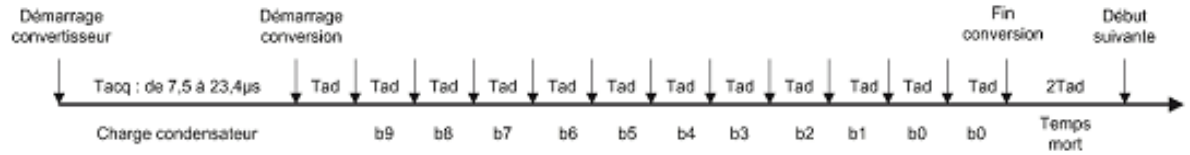
- ✓ Tad : avant le début de conversion (le temps de connexion du condensateur est inclus).
- ✓ 10*Tad pour la conversion des 10 bits du résultat.
- ✓ Tad supplémentaire pour la fin de la conversion.

Soit, au total, un temps de 12 Tad, soit dans le meilleur des cas, un temps de 12 * 1,6µs = 19,2 µs.

Notez qu'un temps équivalent à 2*Tad est nécessaire avant de pouvoir effectuer une nouvelle conversion.

Résumons donc le temps nécessaire pour effectuer l'ensemble des opérations :

- ✓ On charge le condensateur interne (nécessite le temps Tacq).
- ✓ On effectue la conversion (nécessite le temps 12*Tad).
- ✓ On doit attendre 2*Tad avant de pouvoir recommencer une autre conversion.




Nous voici donc arrivé à la question principale. A quelle fréquence maximale pouvons-nous échantillonner notre signal ? En fait, nous avons vu que cela dépendait des conditions, puisque Tacq dépend de la tension d'alimentation et d'autres paramètres.

Calculons le cas le plus défavorable sur une tension d'alimentation de 5V avec un PIC® tournant à une vitesse permettant un Tad de 1,6 µs :

T entre 2 échantillonnages = Tacq + 12 Tad + 2 Tad = Tacq + 14Tad.

Donc : T = 19,71µs + 14 * 1,6µs = 42,11 µs.

Ceci correspond donc à une fréquence de : F = 1/T = 1 / 42,11 * 10-6 = 23,747 kHz.

Cette fréquence vous permet donc d'échantillonner des signaux sinusoïdaux d'une fréquence maximale de 11.874 kHz (la moitié de la fréquence d'échantillonnage).

Si vous vous placez dans le meilleur des cas, vous ramenez Tacq à 7,5µs, ce qui vous donne :

T = 7,5µs + 14 * 1,6µs = 29,9µs.

Soit une fréquence de F = 33.445 kHz. Soit la possibilité de numériser des signaux sinusoïdaux d'une fréquence maximale de 16.722 kHz (si ce n'était la résolution sur 10 bits au lieu de 16, on serait dans le domaine de la hi-fi).

### e) Les valeurs remarquables.

Nous avons vu toutes les formules concernant les temps de numérisation. Ne restent plus que les formules qui nous donnent les relations entre valeurs analogiques et représentations numériques. Nous en avons déjà parlé au moment de l'évocation des techniques d'arrondissement. Si nous définissons :

- ✓ $V_{REF-}$ : Tension minimale analogique (référence négative).
- ✓ $V_{REF+}$ : Tension maximale analogique (référence positive).
- ✓ $V_{IN}$ : Tension d'entrée à numériser.
- ✓ Val : valeur numérique obtenue sur 10 bits.

Nous pouvons dire que pour une numérisation sur 10 bits, on obtiendra la valeur numérique :

$$V_{al} = \frac{V_{IN} - V_{REF-}}{V_{REF+} - V_{REF-}} \times 1023$$

Et réciproquement, la valeur typique qui a été numérisée correspond à une tension de :

$$V_{IN} = \frac{V_{al}}{1023} \times (V_{REF+} - V_{REF-}) + V_{REF-}$$

Si nous utilisons une tension de référence négative de 0V, c'est-à-dire que la référence de tension négative est en réalité Vss, nous obtenons 2 formules simplifiées :

$$V_{al} = \frac{V_{IN}}{V_{REF+}} \times 1023$$

$$V_{IN} = \frac{V_{al}}{1023} \times V_{REF+}$$

En donnant un exemple concret, si nous décidons que la tension de référence positive est de 5V, et que la tension de référence négative est de 0V, nous avons :

$$V_{al} = \frac{V_{IN}}{5} \times 1023$$

$$V_{IN} = \frac{V_{al}}{1023} \times 5$$

**Dernière remarque :** La tension d'entrée ne peut être supérieure à la tension d'alimentation Vdd du PIC®, ni inférieure à sa tension Vss. Si vous voulez mesurer une tension supérieure, par exemple une tension de 15V maximum, il vous faudra réaliser un diviseur de tension à partir de 2 résistances pour que la tension appliquée reste dans les limites prévues.

**Attention.** Maintenant, certains vont trouver curieux qu'on utilise la valeur de 1023 et non 1024. En fait, il serait possible également de raisonner différemment. Si vous prenez nos formules précédentes, vous constatez bien que si vous utilisez 1024 vous allez obtenir une valeur maximale numérique qui peut atteindre 1024 (impossible). Ou, réciproquement, vous ne pourriez jamais trouver par calcul une tension analogique d'entrée égale à Vref+ - Vref-, puisque cette valeur divisée par 1024 et multipliée par la plus grande valeur possible du convertisseur (1023) ne donnerait jamais une valeur maximale. En fait, tout ceci est du à la façon dont on utilise les plages de valeurs. La tension numérique prend des valeurs discrètes, alors que la valeur analogique prend des valeurs continues. Il en résulte que pour une valeur numérique donnée correspond une plage de valeurs analogiques correspondantes.

Les formules ci-dessus donnent les correspondances entre la valeur numérique produite et la valeur analogique des limites Je vais prendre un exemple plus simple pour vous expliquer. Soit un convertisseur analogique/numérique à deux bits. Il ne peut donc fournir que 4 valeurs (0 à 3). Imaginons notre Vref+ - Vref- valant 5V. Nos formules précédentes donneront comme résultat :

- ✓ Si convertisseur = 0, tension = 0V.
- ✓ Si convertisseur = 1, tension = 1,67V.
- ✓ Si convertisseur = 2, tension = 3.33 V.
- ✓ Si convertisseur = 3, tension = 5V.

Tout ça semble logique et produit des résultats cohérents. Mais la valeur produite par un convertisseur de ce type est toujours soumis à une tolérance et correspond de plus à une plage de tension en entrée, et non à une tension unique. Partant de là, il faut évidemment interpréter la valeur lue afin qu'elle corresponde à une réalité physique analogique. Bref, nous pourrions interpréter par exemple maintenant nos valeurs comme suit :

- ✓ Si je lis 0, la tension présente était <= 0,83V (milieu de l'intervalle).
- ✓ Si je lis 1, la tension était comprise entre 0,83V et 2,5V.
- ✓ Si je lis 2, la tension était comprise entre 2,5V et 4,17V.
- ✓ Si je lis 3, la tension était >= 4,17V.

Vous voyez qu'une fois que vous transformez votre valeur analogique unique en plage de valeurs, la réalité devient plus nuancée. Partant de là, puisqu'on parle d'intervalle, on pourrait raisonner de façon légèrement différente.

Reprenons notre exemple, et raisonnons sur les intervalles. Notre convertisseur produit 4 valeurs différentes, nous avons donc 4 intervalles. Une intervalle vaut donc : 5V/4 = 1,25V. Nous obtenons donc 4 valeurs charnière, 0V, 1,25V, 2.5V, 3,75V, 5V. Or, ça fait 5 valeurs et non plus 4, et notre convertisseur, lui, ne peut produire que 4 valeurs.

Qu'à cela ne tienne, il suffit alors de raisonner en considérant des valeurs comme étant nos nouvelles limites plutôt que de considérer comme précédemment que ce sont nos valeurs typiques. Nous obtenons alors :

- ✓ Si je lis 0, la tension présente était <= 1,25V.
- ✓ Si je lis 1, la tension présente était comprise entre 1,25 et 2,5V.
- ✓ Si je lis 2, la tension présente était comprise entre 2,5 et 3,75V.
- ✓ Si je lis 3, la tension présente était >= 3,75V.

Et donc, nous avons remplacé notre « 3 » par « 4 » et donc le « 1023 » de toutes nos formules précédentes par « 1024 » plus intuitif (puissance de 2).

### f) Conclusions pour la partie théorique.

Pour résumer, vous disposez des prescriptions suivantes pour utiliser en pratique le convertisseur A/D avec un PIC® standard :

- ✓ Si une fréquence d'échantillonnage de l'ordre de 23KHz vous suffit, vous utilisez un temps Tacq de 20µs et vous digitalisez sur 10 bits. Aucun besoin de calculs.
- ✓ Si vous avez besoin d'une fréquence comprise entre 23 et 33Khz avec une résolution de 10 bits, vous pouvez optimiser votre montage en réduisant l'impédance de la source (par exemple en utilisant un amplificateur opérationnel), et en utilisant la tension maximale supportée par le PIC®.
- ✓ Si vous avez besoin d'une fréquence supérieure, vous devrez réduire le nombre de bits de numérisation.
- ✓ Si aucune de ces solutions ne vous convient, vous devrez renoncer à numériser avec votre PIC®, et utiliser un convertisseur externe.

Pour plus de facilité, je vous rassemble ici toutes les formules :

$$V_{al\ numérisée} = \frac{V_{IN} - V_{REF-}}{V_{REF+} - V_{REF-}} \times 1023$$

$$V_{IN\ analogique} = \frac{V_{al}}{1023} \times (V_{REF+} - V_{REF-}) + V_{REF-}$$

Temps de conversion sur N bits = Tad + N*Tad + (11-N) (2Tosc).

Tad = Tosc*diviseur ≥ 1,6 µs.

Temps de conversion sur 10 bits = 12 Tad.

Tacq = 2µs + Tc = 0,914895 * 10-9 * (Rinterne + Rsource) + 0,05(T° - 25°C) avec T° ≥ 25°C.

Sachant que les valeurs que vous pouvez utiliser dans la majorité des cas sont :

- ✓ Tacq courant : 19,7µs.
- ✓ Temps de conversion courant : 19,2µs.
- ✓ Temps entre 2 numérisation successives : 3,2µs.

## V.7.2. La théorie appliquée aux PIC®.

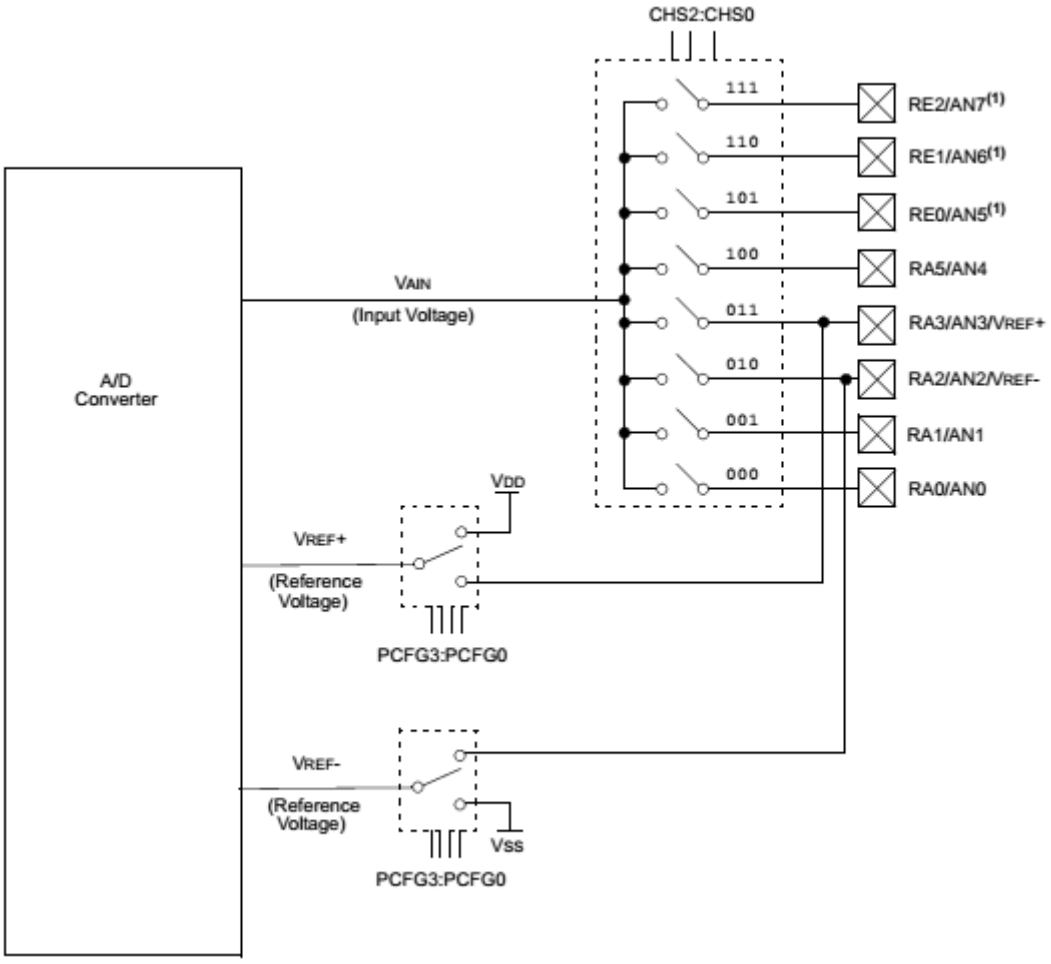
### a) Pins et canaux utilisés.

Maintenant vous savez ce qu'est une conversion analogique/numérique, et comment calculer les différentes valeurs utiles, à moins que vous n'ayez sauté les paragraphes précédents. Reste à savoir comment connecter notre signal analogique sur notre PIC®.

La première chose à comprendre, c'est que notre PIC® ne contient qu'un seul convertisseur, mais plusieurs pins sur lesquelles connecter nos signaux analogiques. Un circuit de commutation sélectionnera donc laquelle des pins sera reliée au condensateur de maintien interne durant le temps Tacq. Ces différentes entrées seront donc des canaux différents d'un seul et même convertisseur.

***Corollaire :*** si vous avez plusieurs canaux à échantillonner, vous devrez les échantillonner à tour de rôle, et donc le temps total nécessaire sera la somme des temps de chaque conversion. Donc, plus vous avez de signaux à échantillonner, moins la fréquence d'échantillonnage pour chaque canal pourra être élevée.

Le 16F876 dispose de 8 canaux d'entrée analogique. Vous pouvez donc échantillonner successivement jusque 8 signaux différents avec ce composant. Les pins utilisées sont les pins AN0 à AN7. Les pins AN0 à AN4 sont les dénominations analogiques des pins RA0 à RA3 + RA5, tandis que les pins AN5 à AN7 sont les dénominations analogiques des pins RE0 à RE2.



### b) Les tensions de référence.

Nous avons vu dans l'étude théorique générale de la conversion analogique/digitale, que cette conversion nécessitait une tension de référence minimale (Vref-) et une tension de référence maximale (Vref+).

Au niveau de notre PIC®, nous avons 3 modes de fonctionnement possibles :

- ✓ Utilisation de Vss (masse du PIC®) comme tension Vref- et de Vdd (alimentation positive du PIC®) comme tension Vref+. Dans ce mode, les tensions de références sont tirées en interne de la tension d'alimentation. Il n'y a donc pas besoin de les fournir.
- ✓ Utilisation de la pin Vref+ pour fixer la tension de référence maximale Vref+, et utilisation de Vss comme tension de référence Vref-. Dans ce cas, la tension Vref+ doit donc être fournie au PIC® via la pin RA3.
- ✓ Utilisation de la pin Vref+ pour fixer la tension de référence maximale Vref+, et utilisation de la pin Vref- pour fixer la tension de référence minimale Vref-. Dans ce cas, les 2 tensions de références devront être fournies au PIC® via RA3 et RA2.

Notez que la broche Vref+ est une dénomination alternative de la broche RA3/AN3, tandis que la broche Vref- est une dénomination alternative de la broche RA2/AN2. Donc, l'utilisation d'une pins comme entrée analogique interdit son utilisation comme entrée numérique (pin entrée/sortie « normale »). De même, l'utilisation des références Vref+ et Vref- interdit leur utilisation comme pin entrée/sortie ou comme pin d'entrée analogique. Notez également que les pins ANx sont des pins d'entrée. Il n'est donc pas question d'espérer leur faire sortir une tension analogique. Ceci nécessiterait un module numérique/analogique complet dont n'est pas pourvu notre PIC®.

*ATTENTION :* lors de l'utilisation d'une tension de référence Vref+ ou Vref-, vous ne pouvez pas utiliser n'importe quelle tension. Vous devez, pour un 16F877 respecter les contraintes suivantes :

- ✓ Vref+ doit être au minimum de Vdd-2.5V et au maximum de Vdd+0.3V.
- ✓ Vref- doit être au minimum de Vss –0.3V et au maxium égal à (Vref+) – 2V.

On voit très bien sur ce schéma que les pins AN2 et AN3 servent selon la position du sélecteur d'entrée analogique ou de tension de référence. Le sélecteur de canal permet de sélectionner lequel des 8 canaux va être appliqué au convertisseur analogique/digital.

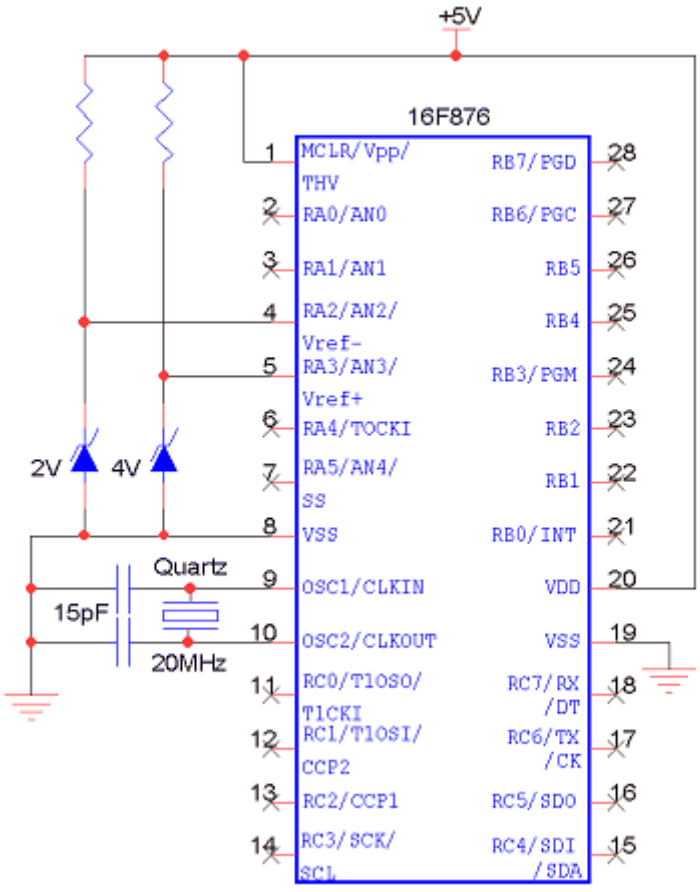
Remarquez que la sélection de la source des tensions de référence dépend de bits du registre ADCON1, tandis que le canal sélectionné pour être numérisé dépend de ADCON0. Nous allons en parler. Le convertisseur en lui-même, en toute bonne logique, n'a besoin que de la tension d'entrée (la pin ANx sélectionnée), et des 2 tensions de référence. Il sort un nombre numérique de 10 bits, dont nous verrons la destination.

Donc, notre procédure de numérisation, pour le cas où on utilise plusieurs canaux, devient la suivante (après paramétrage) :

1. On choisit le canal à numériser, et on met en route le convertisseur.
2. On attend Tacq.
3. On lance la numérisation.
4. On attend la fin de la numérisation.
5. On attend 2.Tad.
6. On recommence avec le canal suivant.

Je vais à présent vous donner un exemple de schéma mettant en œuvre ces tensions de référence. Imaginons que vous vouliez échantillonner une tension qui varie de 2V à 4V en conservant une précision maximale. Vous avez 2 solutions :

1. La première qui vient à l'esprit est d'utiliser une entrée analogique sans tension de référence externe, comme pour l'exercice précédent. Dans ce cas votre valeur numérique ne pourra varier, en appliquant la formule Val = (VIN / VREF+ ) * 1023) , que de : (2/5) * 1023 = 409 pour une tension de 2V à (4/5) * 1023 = 818 pour une tension de 4V. Votre précision sera donc de 409 pas sur les 1023 possibles, les autres valeurs étant inutilisées. Vous avez donc une perte de précision.
2. Par contre, si vous utilisez une tension de référence de 2V comme Vref- et une de 4V comme Vref+, vous aurez une valeur numérique de 0 pour la tension 2V, et une valeur de 1023 pour la tension de 4V. Vous conservez donc un maximum de précision, puisque votre intervalle de mesure correspond à 1024 paliers. Voici le schéma que vous devrez utiliser, après avoir paramètré ADCON1 en conséquence.



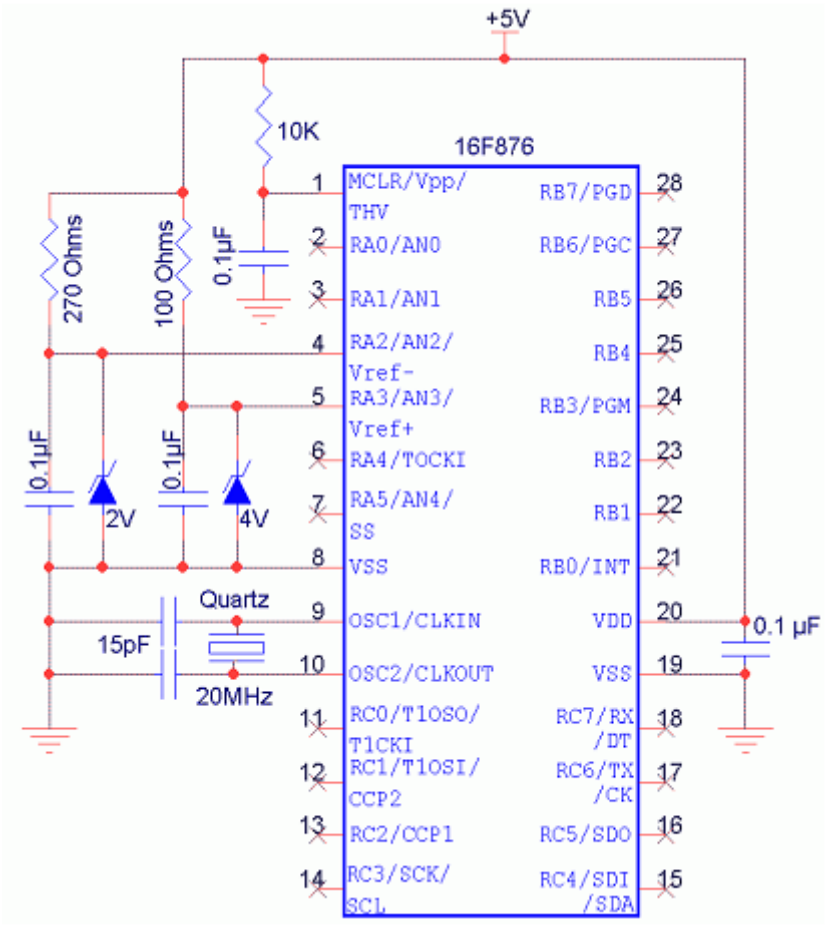
Les résistances seront calculées en fonction des diodes zener et de la loi d'ohm.

Vous constatez que ceci vous permet d'avoir des tensions de référence qui ne varient pas en fonction de la tension d'alimentation. Il peut donc être pratique, si votre alimentation n'est pas stable, d'utiliser des tensions de références fortement stabilisées.

Ceci est un schéma théorique. Mais la bonne pratique électronique recommande de :

1. Placer un condensateur en parallèle avec chaque diode zener, ceci afin d'éliminer le « bruit » des jonctions de ces diodes (pour ce montage particulier).
2. Placer une résistance et un condensateur sur la pin MCLR (pour tous vos montages réels).
3. Placer un condensateur de découplage sur chaque pin d'alimentation Vdd (pour tous vos montages réels).

Bien entendu, ces « accessoires » ne sont absolument pas nécessaires pour vos platines d'expérimentation. Ils sont nécessaires pour une utilisation sur un circuit imprimé comportant d'autres circuits perturbateurs ou pouvant être perturbés, et dans un circuit d'application réel, pour lesquels un « plantage » est toujours gênant. Je vous donne, à titre d'information, le circuit précédent modifié pour une application réelle :

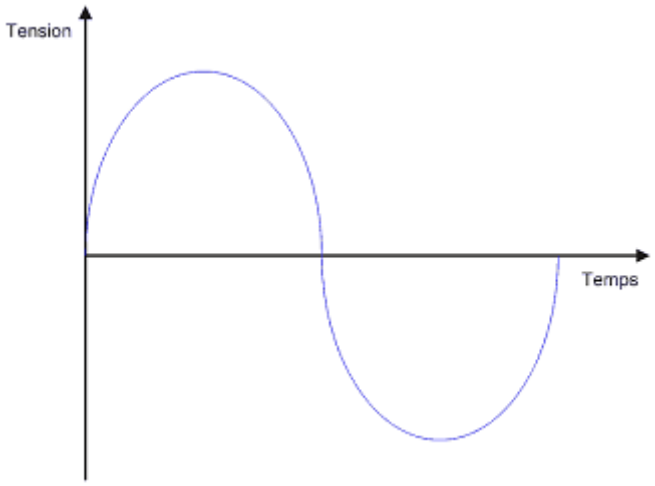



Notez que si vous désirez mesurer, par exemple, la position d'un potentiomètre, l'erreur s'annulera d'elle-même si vous n'utilisez pas les tensions de référence externes. En effet, si la

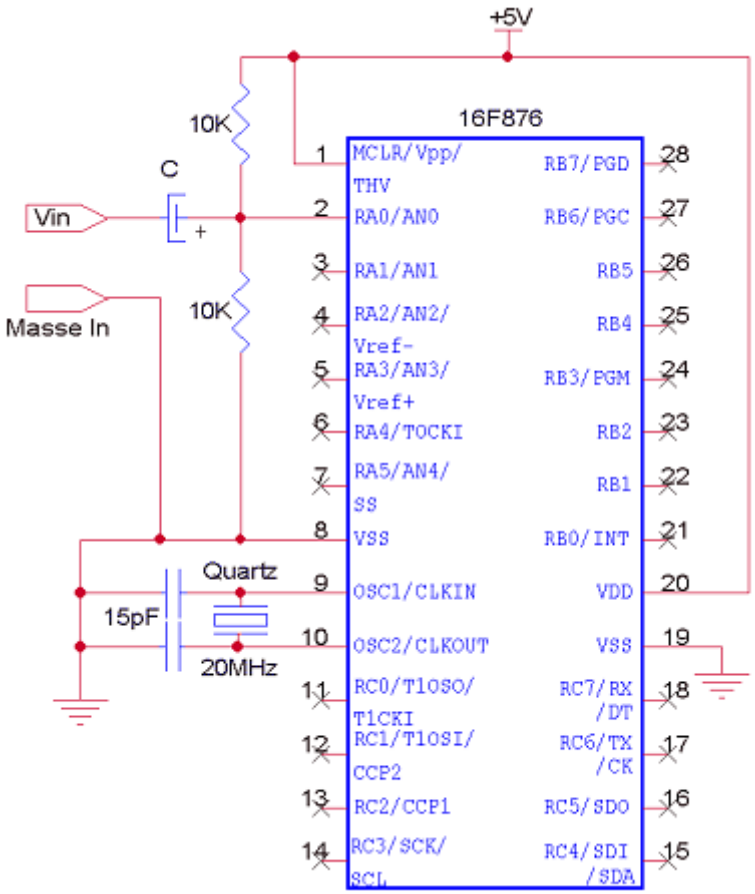
tension d'alimentation varie, la référence interne +Vdd variera également. Comme le potentiomètre pourra être alimenté par la même tension d'alimentation, la tension qu'il fournira variera dans les mêmes proportions, donc l'erreur s'annulera. Il n'est donc pas toujours préférable d'imposer une tension de référence distincte de votre alimentation. Tout dépend de l'application. Nous allons poursuivre l'étude de notre convertisseur par l'étude des registres dont nous venons de dévoiler le nom. Mais, auparavant :

### c) Mesure d'une tension alternative.

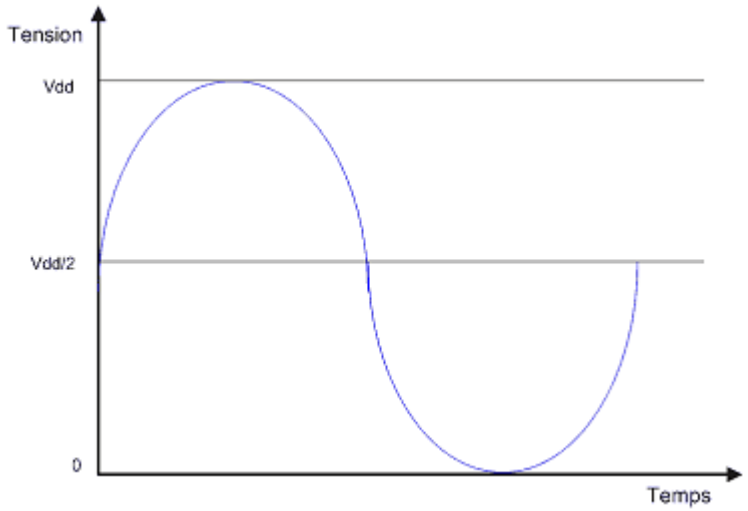
Nous avons vu que notre tension pouvait varier entre Vss et Vdd. Nous ne pouvons donc pas échantillonner directement une tension alternative, comme un signal audio, par exemple. Celui-ci est en effet de la forme suivante.




Nous allons donc devoir nous arranger pour que la tension reste en permanence positive. Ceci s'effectue en forçant la tension d'entrée « à vide » à une tension Vdd/2, et en amenant la tension alternative via un condensateur, de façon à effectuer une addition des 2 tensions (figure ci-dessous).




Ceci ramènera la tension à mesurer vue par le PIC® comme indiquée ci-dessous.




Vous voyez que maintenant, l'intégralité de la tension à mesurer est positive et peut donc être mesurée. A vous d'interpréter que la valeur centrale correspond à une tension alternative d'entrée de 0, et qu'une tension de Vdd ou de 0V correspond à un maximum de tension.

Reste à calculer la valeur du condensateur. C'est assez simple. Vous considérez que la résistance d'entrée vaut approximativement la moitié de la valeur des résistances utilisées, soit dans ce cas 5 Kohms.

Pour que le condensateur n'influence pas sur la mesure, son impédance doit être négligeable vis-à-vis de la résistance d'entrée. Le terme « négligeable » dépend bien entendu de la précision de la conversion souhaitée.

La valeur de l'impédance diminue avec l'augmentation de fréquence suivant la formule :

$$Z_c = \frac{1}{2 \times \pi \times f \times C} \Rightarrow C = \frac{1}{2 \times \pi \times f \times Z_c}$$

Avec :

- ✓ Zc = impédance ;
- ✓ π = 3,1415 ;
- ✓ f = fréquence en Hz ;
- ✓ C = capacité en farads.

Donc, on se place dans le cas le plus défavorable, c'est-à-dire à la fréquence la plus basse. Prenons un cas concret : On doit numériser une fréquence audio, de 50 Hz à 10 Khz. On décide que Zc doit être 10 fois plus petite que 5 Kohms On aura donc un condensateur au moins égal à :

$$C = \frac{1}{2 \times \pi \times 50 \times 500} = 6{,}37\mu F$$

Notez que dans ce cas, l'impédance de la source de votre signal devra également être plus petite que 5Kohms, sous peine d'une forte atténuation.

### V.7.3. Les registres ADRESL et ADRESH.

J'attire votre attention sur le fait que le convertisseur donne un résultat sur 10 bits, et donc que ce résultat devra donc obligatoirement être sauvegardé dans 2 registres. Ces registres sont tout simplement les registres ADRESL et ADRESH.

Comme 2 registres contiennent 16 bits, et que nous n'en utilisons que 10, Microchip® vous a laissé le choix sur la façon dont est sauvegardé le résultat. Vous pouvez soit justifier le résultat à gauche, soit à droite.

La justification à droite complète la partie gauche du résultat par des « 0 ». Le résultat sera donc de la forme :

| ADRESH | | | | | | ADRESL | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | b9 | b8 | b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 |

La justification à gauche procède bien évidemment de la méthode inverse :

| ADRESH | | | | | | | | ADRESL | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| b9 | b8 | b7 | b6 | b5 | b4 | b3 | b2 | b1 | b0 | 0 | 0 | 0 | 0 | 0 | 0 |

La justification à droite sera principalement utilisée lorsque vous avez besoin de l'intégralité des 10 bits de résultat, tandis que la justification à gauche est très pratique lorsque 8 bits vous suffisent. Dans ce cas, les 2 bits de poids faibles se trouvent isolés dans ADRESL, il suffit donc de ne pas en tenir compte. Cette approche est destinée à vous épargner des décalages de résultats.

Le choix de la méthode s'effectue à l'aide du bit 7 de ADCON1.

### V.7.4. Le registre ADCON1.

Ce registre permet de déterminer le rôle de chacune des pins AN0 à AN7. Il permet donc de choisir si une pin sera utilisée comme entrée analogique, comme entrée/sortie standard, ou comme tension de référence. Il permet également de décider de la justification du résultat.

Notez déjà que pour pouvoir utiliser une pin en mode analogique, il faudra que cette pin soit configurée également en entrée par TRISA et éventuellement par TRISE pour le 16F877.

Le registre ADCON1 dispose, comme tout registre accessible de notre PIC®, de 8 bits, dont seulement 5 sont utilisés :

**ADCON1.**

- ✓ b7 : ADFM : A/D result ForMat select.
- ✓ b6 : Inutilisé : lu comme « 0 ».
- ✓ b5 : Inutilisé : lu comme « 0 ».
- ✓ b4 : Inutilisé : lu comme « 0 ».
- ✓ b3 : PCFG3 : Port ConFiGuration control bit 3.
- ✓ b2 : PCFG2 : Port ConFiGuration control bit 2.
- ✓ b1 : PCFG1 : Port ConFiGuration control bit 1.
- ✓ b0 : PCFG0 : Port ConFiGuration control bit 0.

Le bit ADFM permet de déterminer si le résultat de la conversion sera justifié à droite (1) ou à gauche (0).

Nous trouvons dans ce registre les 4 bits de configuration des pins liées au convertisseur analogique/numérique. Ces bits nous permettent donc de déterminer le rôle de chaque pin. Comme nous avons 16 combinaisons possibles, nous aurons autant de possibilités de configuration (en fait, vous verrez que nous n'en avons que 15). Je vous donne le tableau correspondant à ces combinaisons pour le 16F877.

| PCFG 3 à 0 | AN7 RE2 | AN6 RE1 | AN5 RE0 | AN4 RA5 | AN3 RA3 | AN2 RA2 | AN1 RA1 | AN0 RA0 | Vref- | Vref + | A/D/R |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0000 | A | A | A | A | A | A | A | A | Vss | Vdd | 8/0/0 |
| 0001 | A | A | A | A | Vref+ | A | A | A | Vss | RA3 | 7/0/1 |
| 0010 | D | D | D | A | A | A | A | A | Vss | Vdd | 5/3/0 |
| 0011 | D | D | D | A | Vref+ | A | A | A | Vss | RA3 | 4/3/1 |
| 0100 | D | D | D | D | A | D | A | A | Vss | Vdd | 3/5/0 |
| 0101 | D | D | D | D | Vref+ | D | A | A | Vss | RA3 | 2/5/1 |
| 0110 | D | D | D | D | D | D | D | D | - | - | 0/8/0 |
| 0111 | D | D | D | D | D | D | D | D | - | - | 0/8/0 |
| 1000 | A | A | A | A | Vref+ | Vref- | A | A | RA2 | RA3 | 6/0/2 |
| 1001 | D | D | A | A | A | A | A | A | Vss | Vdd | 6/2/0 |
| 1010 | D | D | A | A | Vref+ | A | A | A | Vss | RA3 | 5/2/1 |
| 1011 | D | D | A | A | Vref+ | Vref- | A | A | RA2 | RA3 | 4/2/2 |
| 1100 | D | D | D | A | Vref+ | Vref- | A | A | RA2 | RA3 | 3/3/2 |
| 1101 | D | D | D | D | Vref+ | Vref- | A | A | RA2 | RA3 | 2/4/2 |
| 1110 | D | D | D | D | D | D | D | A | Vss | Vdd | 1/7/0 |
| 1111 | D | D | D | D | Vref+ | Vref- | D | A | RA2 | RA3 | 1/5/2 |

La première colonne contient les 16 combinaisons possibles des bits de configuration PCFG3 à PCFG0. Remarquez déjà que les valeurs 0110 et 0111 donnent les mêmes résultats. Vous avez donc en réalité le choix entre 15 et non 16 combinaisons.

Les colonnes « AN7 à AN0 » indiquent le rôle qui sera attribué à chacune des pins concernée. Un « A » dans une de ces colonnes indique que la pin correspondante est configurée comme entrée analogique (ne pas oublier TRISx), un « D » indiquera que la pin est en mode Digital, c'est-à-dire qu'elle se comportera comme une pin d'entrée/sortie « classique ».

La colonne « AN3/RA3 » peut contenir également la valeur « Vref+ » qui indiquera que cette pin devra recevoir la tension de référence maximale.

Il en va de même pour AN2/RA2 qui pourra contenir « Vref- », qui indiquera que cette pin doit recevoir la tension de référence minimale.

La colonne « Vref- » indique quelle tension de référence minimale sera utilisée par le convertisseur. Il ne pourra s'agir que de la tension d'alimentation Vss ou de la pin RA2. Cette colonne est donc liée au contenu de la colonne « RA2 ».

Raisonnement identique pour la colonne « Vref+ », liée à « RA3 ». Cette colonne indique quelle sera la tension de référence maximale. De nouveau, il ne pourra s'agir que de la tension d'alimentation Vdd ou de la tension présente sur la pin RA3.

La dernière colonne « A/D/R » résume les colonnes précédentes. Le premier chiffre représente le nombre de pins configurées en tant qu'entrées analogiques, le second en tant qu'entrées/sorties numériques, et le dernier le nombre de pins servant à l'application des tensions de référence. Comme il y a 8 pins concernées pour le 16F877, la somme des 3 chiffres pour chaque ligne sera bien entendu égale à 8.

Vous voyez que si vous avez le choix du nombre de pins configurées en entrées analogiques, vous n'avez cependant pas le choix de leur attribution.

Par exemple, si vous avez besoin de configurer ces ports pour disposer de 3 entrées analogiques et de 5 entrées/sorties numériques, vous devez chercher dans la dernière colonne la ligne « 3/5/0 ». Cette ligne vous indique que vous devez configurer les bits PCFGx à 0100, et que les pins utilisées comme entrées analogiques seront les pins RA0,RA1, et RA3. Vous devez donc en tenir compte au moment de concevoir votre schéma.

Une fois de plus, logiciel et matériel sont étroitement liés. Vous voyez également que lors d'une mise sous tension, les bits PCFGx contiennent 0000. Le PORTA et le PORTE seront donc configurés par défaut comme ports complètement analogiques. Ceci vous explique pourquoi l'utilisation de ces ports comme ports d'entrées/sorties classiques implique d'initialiser ADCON1, avec une des valeurs des 2 lignes complètement en bleu dans le tableau.

## V.7.5. Le registre ADCON0.

Ce registre est le dernier utilisé par le convertisseur analogique/numérique. Il contient les bits que nous allons manipuler lors de notre conversion. Sur les 8 bits de notre registre, 7 seront utilisés.

- ✓ b7 : ADCS1 : A/D conversion Clock Select bit 1.
- ✓ b6 : ADCS0 : A/D conversion Clock Select bit 0.
- ✓ b5 : CHS2 : analog Channel Select bit 2.
- ✓ b4 : CHS1 : analog Channel Select bit 1.
- ✓ b3 : CHS0 : analog Channel Select bit 0.
- ✓ b2 : GO/*DONE* : A/D conversion status bit.
- ✓ b1 : Inutilisé : lu comme « 0 » ;
- ✓ b0 : ADON : A/D ON bit.

Nous avons parlé à maintes reprises de diviseur, afin de déterminer l'horloge du convertisseur en fonction de la fréquence du quartz utilisé. Vous pouvez choisir ce diviseur à l'aide des bits ADCSx.

| ADCS1 | ADCS0 | Diviseur | Fréquence maximale du quartz |
|---|---|---|---|
| 0 | 0 | Fosc/2 | 1,25Mhz |
| 0 | 1 | Fosc/8 | 5Mhz |
| 1 | 0 | Fosc/32 | 20 Mhz |
| 1 | 1 | Osc RC | Si > 1MHz, uniquement en mode « sleep » |

- ✓ La conversion durant le mode « sleep » nécessite de configurer ces bits sur « Osc RC », car l'oscillateur principal du PIC® est à l'arrêt durant ce mode.
- ✓ Par contre, l'emploi de l'oscillateur RC pour les PIC® connectées à un quartz de plus de 1MHz vous impose de placer le PIC® en mode « sleep » durant la conversion.

- ✓ La mise en sommeil du PIC® durant une conversion, alors que l'oscillateur n'est pas configuré comme oscillateur RC entraînera un arrêt de la conversion en cours, et une absence de résultat, même au réveil du PIC®.

Vous avez vu que vous pouvez configurer, via ADCON1, plusieurs pins comme entrées analogiques. Vous avez vu également que vous ne pouvez effectuer la conversion que sur une pin à la fois (on parlera de canal). Vous devez donc être en mesure de sélectionner le canal voulu. Ceci s'effectue via les bits CHSx.

| CHS2 | CHS1 | CHS0 | Canal | Pin |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | AN0/RA0 |
| 0 | 0 | 1 | 1 | AN1/RA1 |
| 0 | 1 | 0 | 2 | AN2/RA2 |
| 0 | 1 | 1 | 3 | AN3/RA3 |
| 1 | 0 | 0 | 4 | AN4/RA5 |
| 1 | 0 | 1 | 5 | AN5/RE0 (Uniquement pour 16F877) |
| 1 | 1 | 0 | 6 | AN6/RE1 (Uniquement pour 16F877) |
| 1 | 1 | 1 | 7 | AN7/RE2 (Uniquement pour 16F877) |

Le bit ADON permet de mettre en service le convertisseur. Si le canal a été correctement choisi, le positionnement de ce bit permet de démarrer la charge du condensateur interne, et donc détermine le début du temps d'acquisition.

Quant au bit Go/*DONE*, il sera placé à « 1 » par l'utilisateur à la fin du temps d'acquisition. Cette action détermine le début de la conversion elle-même, qui, je le rappelle, dure 12 Tad. Une fois la conversion terminée, ce bit est remis à 0 (« Done » = « Fait ») par l'électronique du convertisseur. Cette remise à 0 est accompagnée du positionnement du flag ADIF du registre PIR1. Ce bit permettra éventuellement de générer une interruption. Vous disposez donc de 2 façons pratiques de connaître la fin de la durée de conversion :

1. Si votre programme n'a rien d'autre à faire durant l'attente de la conversion, vous bouclez dans l'attente du passage à 0 du bit GO/Done.
2. Si votre programme continue son traitement, vous pouvez utiliser l'interruption générée par le positionnement du flag ADIF.

***Attention*** : Si vous arrêtez manuellement la conversion en cours, le résultat ne sera pas transféré dans les registres ADRESL et ADRESH. Vous n'obtenez donc aucun résultat, même partiel. De plus, vous devrez quand même respecter un temps d'attente de 2Tad avant que ne redémarre automatiquement l'acquisition suivante.

## V.7.6. La conversion analogique/numérique et les interruptions.

Cette partie ne comporte aucune difficulté particulière. En effet, l'interruption générée par le convertisseur est une interruption périphérique, et doit donc être traitée comme telle. Les différentes étapes de sa mise en service sont donc :

- ✓ Positionnement du bit ADIE du registre PIE1.
- ✓ Positionnement du bit PEIE du registre INTCON.

- ✓ Positionnement du bit GIE du registre INTCON Moyennant quoi, toute fin de conversion analogique entraînera une interruption. Il vous suffira de remettre à « 0 » le flag ADIF après traitement de cette interruption.

### V.7.7. Résumé sur l'utilisation pratique du convertisseur.

Arrivé à ce stade, vous disposez de toutes les informations pour effectuer votre mesure de grandeur analogique.

Voici un résumé des opérations concrètes à effectuer pour échantillonner votre signal :

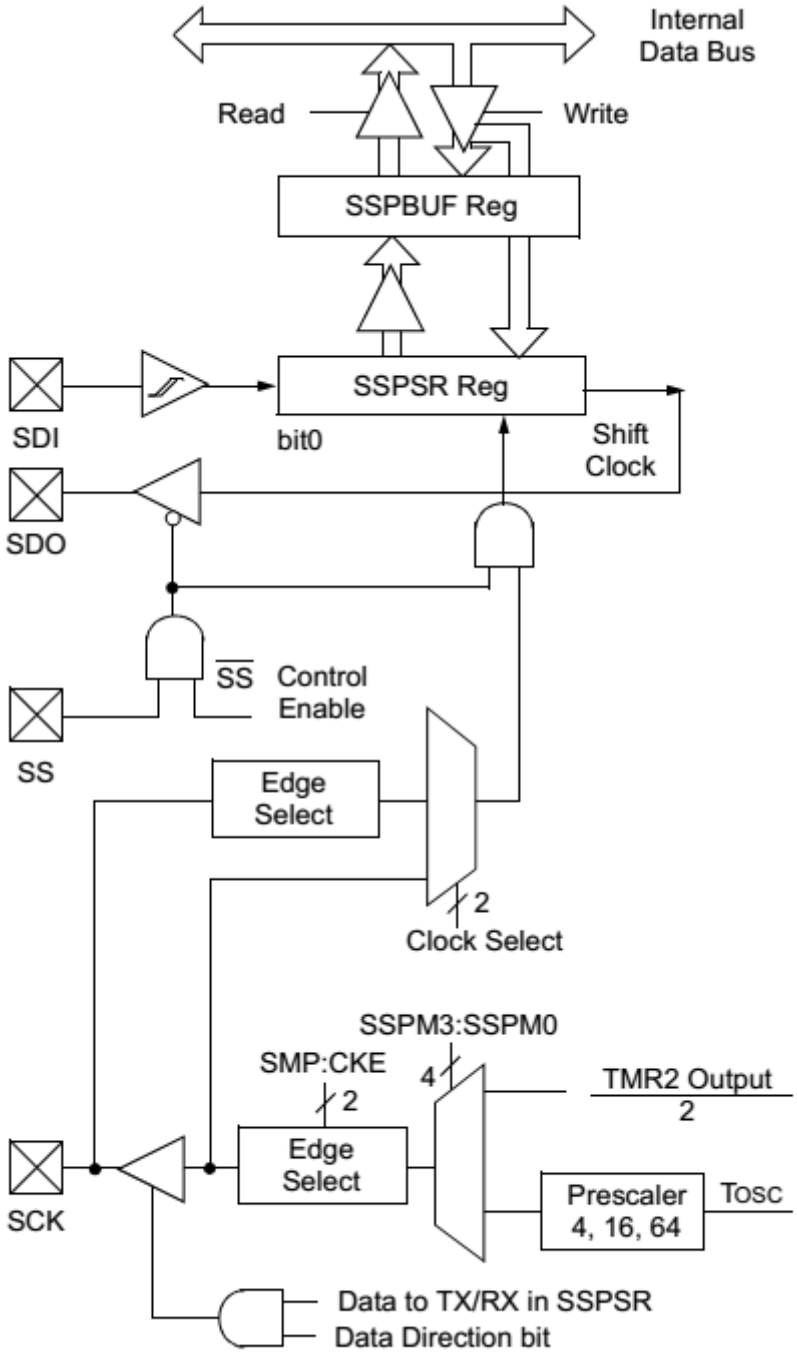
1. Configurez ADCON1 en fonction des pins utilisées en mode analogique, ainsi que les registres TRISA et TRISE si nécessaire.
2. Validez, si souhaitée, l'interruption du convertisseur.
3. Paramètrez sur ADCON0 le diviseur utilisé.
4. Choisissez le canal en cours de digitalisation sur ADCON0.
5. Positionnez, si ce n'est pas déjà fait, le bit ADON du registre ADCON0.
6. Attendez le temps Tacq (typiquement 19,7µs sous 5V).
7. Démarrez la conversion en positionnant le bit GO du registre ADCON0.
8. Attendez la fin de la conversion.
9. Lisez les registres ADRESH et si nécessaire ADRESL.
10. Attendez un temps équivalent à 2Tad (typiquement 3,2µs).
11. Recommencez au point 4.

Notez que puisque l'acquisition redémarre automatiquement après le temps « 2 Tad », vous pouvez relancer l'acquisition directement, à votre charge d'attendre non pas le temps Tacq pour la fin de l'acquisition, mais le temps Tacq + 2Tad. Ceci vous épargne une temporisation.

En effet, 2 temporisations qui se suivent peuvent être remplacées par une temporisation unique de temps cumulé.

## V.8. Module MSSP en mode SPI.

Le module MSSP, pour Master Synchronous Serial Port, permet l'échange de données du PIC® avec le monde extérieur, en utilisant des transmissions série synchrones. Le diagramme correspondant est le suivant :




Il n'est pas le seul à proposer des communications, nous avons déjà vu la liaison parallèle dans le module PSP, et nous verrons d'autres communications série avec le module USART.

Vous verrez que l'ensemble de ces modules vous ouvre toutes les voies royales de la communication, que ce soit avec votre PC, avec des afficheurs, des mémoires de type eeprom, des détecteurs de température, des horloges en temps réel, etc.

Une fois maîtrisées ces possibilités, vous pourrez réaliser des applications mettant en œuvre d'autres circuits que votre simple PIC®.

Nous sommes donc en présence d'un chapitre très important. Ceci explique les nombreuses redondances d'explications que je vais vous fournir. Je préfère en effet répéter plusieurs fois la même chose avec des termes différents, que de laisser des lecteurs en chemin, à cause d'une erreur d'interprétation ou d'une incompréhension.

Nous verrons qu'avec ces modules, nous n'avons plus à nous occuper de la transmission des bits en eux-mêmes, mais nous les recevrons et émettrons directement sous forme d'octet, toute la procédure de sérialisation étant alors automatique. Outre le gain en taille du programme qui en découlera, vous vous rendez compte que les vitesses de transmission pourront être améliorées de façon notable.

Au lieu que votre programme ne soit interrompu lors de la réception ou de l'émission de chaque bit, il ne le sera plus que pour chaque octet. Ceci multiplie allègrement la vitesse maximale possible d'un facteur de plus de 10. Il y a tellement de modes possibles de fonctionnement, qu'il ne me sera pas possible de donner systématiquement des exemples. Je me limiterai donc à des situations courantes.

Nous verrons que le module MSSP peut travailler selon 2 modes. Soit en mode SPI, soit en mode I2C.

### V.8.1. Les liaisons séries de type synchrone.

Une liaison série est une liaison qui transfère les données bit après bit (en série), au contraire d'une liaison parallèle, qui transmet un mot à la fois (mot de 8 bits, 16 bits, ou plus suivant le processeur ou l'interface).

La notion de synchrone est bien entendu de la même famille que « synchronisé ». Ceci signifie simplement que l'émetteur/récepteur fournira un signal de synchronisation qui déterminera non seulement le début et la fin de chaque octet, mais également la position de chaque état stable des bits.

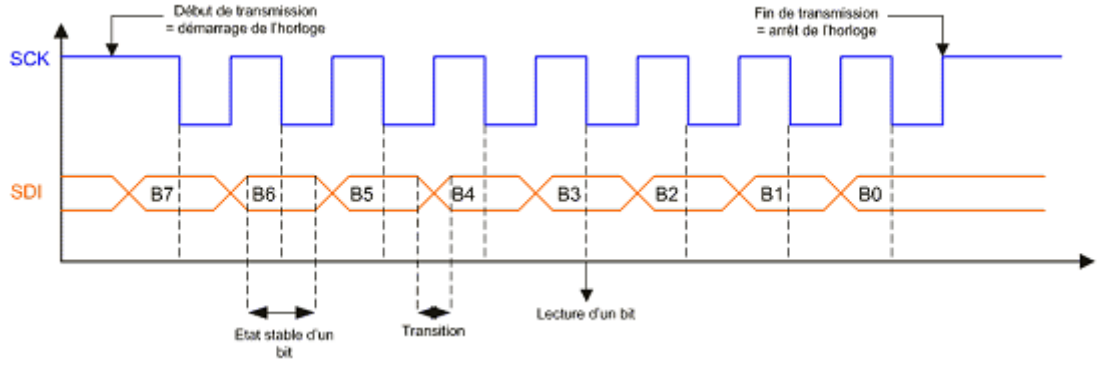
Nous voyons donc que ce fonctionnement nécessite en plus des lignes de communication (entrée et sortie de données), une ligne qui véhicule le signal de synchronisation (on parlera d'horloge). Cette méthode présente donc l'avantage de permettre la lecture des bits toujours au moment le plus favorable, et évite les problèmes dus aux imprécisions de la vitesse. Il ne nécessite pas non plus dans le cas du SPI de bit de départ (start-bit), ni de fin (stop-bit).

Par contre, elle présente l'inconvénient de nécessiter une ligne supplémentaire pour véhiculer l'horloge. Cette ligne, parcourue par définition par un signal à haute fréquence, raccourcira en général la longueur maximale utilisable pour la liaison.

Vous avez bien entendu 2 façons d'envoyer les bits à la suite les uns des autres :

1. Soit vous commencez par le bit 7, et vous poursuivez jusqu'au bit 0. C'est la méthode utilisée par le module MSSP.
2. Soit vous procédez de façon inverse, d'abord le bit 0 jusqu'au bit de poids le plus fort. C'est de cette façon que fonctionnera notre module USART, que nous étudierons plus tard.

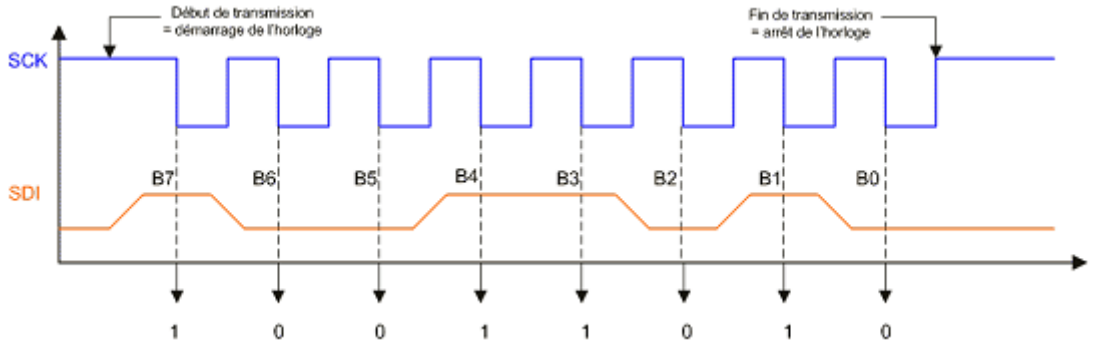
Voici un exemple tout à fait général de réception d'un mot de 8 bits en mode série synchrone. Dans cet exemple, la lecture s'effectue sur le front descendant du signal d'horloge :




Vous constatez que :

- ✓ La lecture s'effectue à un endroit stable du bit concerné (vers le milieu de sa durée).
- ✓ Il y a 2 lignes rouges, car le bit peut prendre 2 valeurs (0 ou 1).
- ✓ Le passage d'un niveau à l'autre n'est pas instantané, ce qui explique les lignes rouges obliques, et la zone de transition est le temps durant laquelle une lecture ne donnerait pas une valeur fiable (la transition entre les niveaux n'est pas encore complètement terminée). Plus la vitesse est faible, plus cette période de transition est petite par rapport à la durée de l'état stable du bit. A une vitesse plus faible, les transitions apparaissent donc de façon pratiquement verticale.

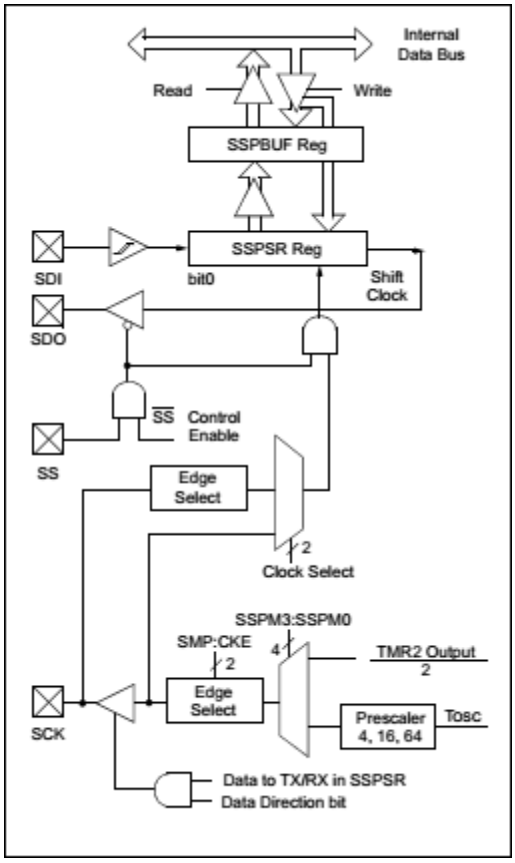
Voici un second exemple, qui donne une lecture concrète de l'octet B'10011010'.




Cet exemple concret vous permet d'assimiler la différence de représentation entre le cas général (première figure avec double ligne rouge) et un cas particulier (seconde figure avec ligne rouge simple définissant l'octet).

## V.8.2. Le mode SPI.

SPI signifie Serial Peripheral Interface. Ce mode correspond donc à un fonctionnement « standard » du port série synchrone. Il permet d'interconnecter de façon flexible et paramétrable différents composants avec les 16F87x. La figure ci-dessous décrit la configuration matérielle du module MSSP en mode SPI.




Le mode SPI nécessite sur notre PIC® :

- ✓ Une entrée de données série (SDI pour Serial Data Input).
- ✓ Une sortie de données série (SDO pour Serial Data Output).
- ✓ Une entrée/sortie horloge (SCK pour Serial ClocK).
- ✓ Une entrée optionnelle de sélection du PIC® (*SS* pour Slave Select).

Vous pouvez retrouver ces pins sur le brochage de nos PIC®. Comme d'habitude, ces fonctions sont multiplexées avec les ports classiques.

Dans ce mode, le module peut évidemment émettre et recevoir. Il peut gérer lui-même l'horloge (mode « master ») ou subir l'horloge gérée par un autre microprocesseur (mode « slave »). Dans ce dernier cas, il peut également y avoir d'autres esclaves (mode «multislave»).

Vous constaterez donc, que, si on excepte les fils d'alimentation, une liaison de ce type nécessite au minimum 2 fils (lignes) reliant les 2 composants. Le nombre de lignes peut cependant croître suivant les besoins de l'utilisateur.

La ligne d'horloge est toujours nécessaire, du fait même de la définition de « synchrone ».On peut donc avoir les modes de liaisons suivants :

- ✓ ***Liaison à 2 fils*** : Permet de véhiculer l'information dans un seul sens à un moment donné. On pourra distinguer la liaison unidirectionnelle (c'est toujours le même composant qui émet, et toujours le même qui reçoit) et la liaison bidirectionnelle half-duplex (chacun émet et reçoit à tour de rôle et sur le même fil).

- ✓ ***Liaison à 3 fils*** : Permet de véhiculer l'information dans les 2 sens (bidirectionnelle), un fil différent étant dévolu à chaque sens de transfert d'information. Comme ceci permet la communication simultanée dans les 2 sens de transmission, on parlera de liaison fullduplex.

Nous verrons plus loin qu'en réalité le PIC® émet et reçoit toujours simultanément lorsqu'on utilise le module SPI. L'octet qui n'est éventuellement pas utilisé sera un octet fictif.

L'ajout d'un fil supplémentaire dédicacé (SS) par esclave, à ces 2 modes précédents, permet de connecter plusieurs esclaves sur la même ligne d'horloge. Le signal de sélection permet de choisir quel esclave réagira sur la génération de l'horloge. Prenez garde que tous ces signaux nécessitent forcément une référence de tension (masse).

Donc, si votre émetteur et votre récepteur ne partagent pas la même alimentation, il vous faudra interconnecter les 2 tensions Vss et Vdd de vos circuits. Ceci nécessitera donc un fil supplémentaire.

Attention de prendre alors toute précaution utile éventuelle du fait de cette liaison. Tout ceci nous donne une pléthore de modes de fonctionnement, que nous allons étudier en séquence.

### a. Les registres utilisés.

Tout au long de l'étude de notre module MSSP, nous allons retrouver 2 registres de configuration et de statuts, à savoir « SSPSTAT », « SSPCON », plus un registre « SSPCON2 » utilisé uniquement pour le mode I2C, et dont nous ne parlerons donc pas dans ce paragraphe.

Ces registres contiennent des bits de contrôle et des indicateurs dont le rôle dépend du mode utilisé actuellement par le module.

A ces registres s'ajoutent le SSPSR (Synchronous Serial Port Shift Register), qui contient la donnée en cours de transfert, et le registre SSPBUF (Synchronous Serial Port BUFfer) qui contient l'octet à envoyer, ou l'octet reçu, suivant l'instant de la communication.

Les autres registres utilisés sont des registres dont nous avons déjà étudié le fonctionnement. Le mécanisme général n'est pas très compliqué à comprendre.

Voyons tout d'abord du côté de l'émetteur. L'octet à envoyer est placé dans le registre SSPBUF. La donnée est recopiée automatiquement par le PIC® dans le registre SSPSR, qui est un registre destiné à sérialiser la donnée (la transformer en bits successifs). Ce second registre, non accessible par le programme, est tout simplement un registre qui effectue des décalages. Comme le premier bit à envoyer est le bit 7, le registre devra décaler vers la gauche. Le bit sortant est directement envoyé sur la ligne SDO. Le mécanisme se poursuit jusqu'à ce que les 8 bits soient envoyés.

Côté récepteur, c'est évidemment le même genre de mécanisme. Le bit reçu sur la ligne SDI est entré par le côté droit du même registre SSPSR, donc par le bit 0. Ce registre subit alors un décalage vers la gauche qui fait passer ce bit en position b1. Le bit suivant sera alors reçu en

position b0, et ainsi de suite. Le dernier bit reçu entraîne automatiquement la copie de la donnée contenue dans SSPSR vers le registre SSPBUF.

Donc, on résume la séquence de la façon suivante :

- ✓ L'émetteur copie sa donnée de SSPBUF vers SSPSR.
- ✓ Pour chacun des 8 bits,
- ✓ Au premier clock, l'émetteur décale SSPSR vers la gauche, le bit sortant (ex b7) est envoyé sur SDO.
- ✓ Au second clock, le récepteur fait entrer le bit présent sur SDI et le fait entrer dans SSPSR en décalant ce registre vers la gauche. Ce bit se trouve maintenant en b0.
- ✓ On recommence pour le bit suivant.
- ✓ Le récepteur copie SSPSR dans SSPBUF à la fin de la transmission, l'octet à envoyer qui avait été placé dans SSPBUF aura été remplacé par l'octet reçu.

Ce qu'il faut retenir, c'est qu'il faut 2 synchronisations différentes.

La première pour placer le bit à envoyer sur la ligne de transmission, et la seconde pour dire au récepteur qu'on peut lire ce bit, qui est maintenant stable. Comme le signal d'horloge présente 2 flancs (un flanc montant et un flanc descendant), nous avons donc nos 2 repères avec une seule horloge.

Vous avez bien entendu compris qu'émission et réception sont simultanées.

Remarquez qu'il n'y a qu'un seul registre SSPSR, et un seul SSPBUF, ce qui vous indique qu'émission et réception se font simultanément au sein d'un même PIC® de la façon suivante :

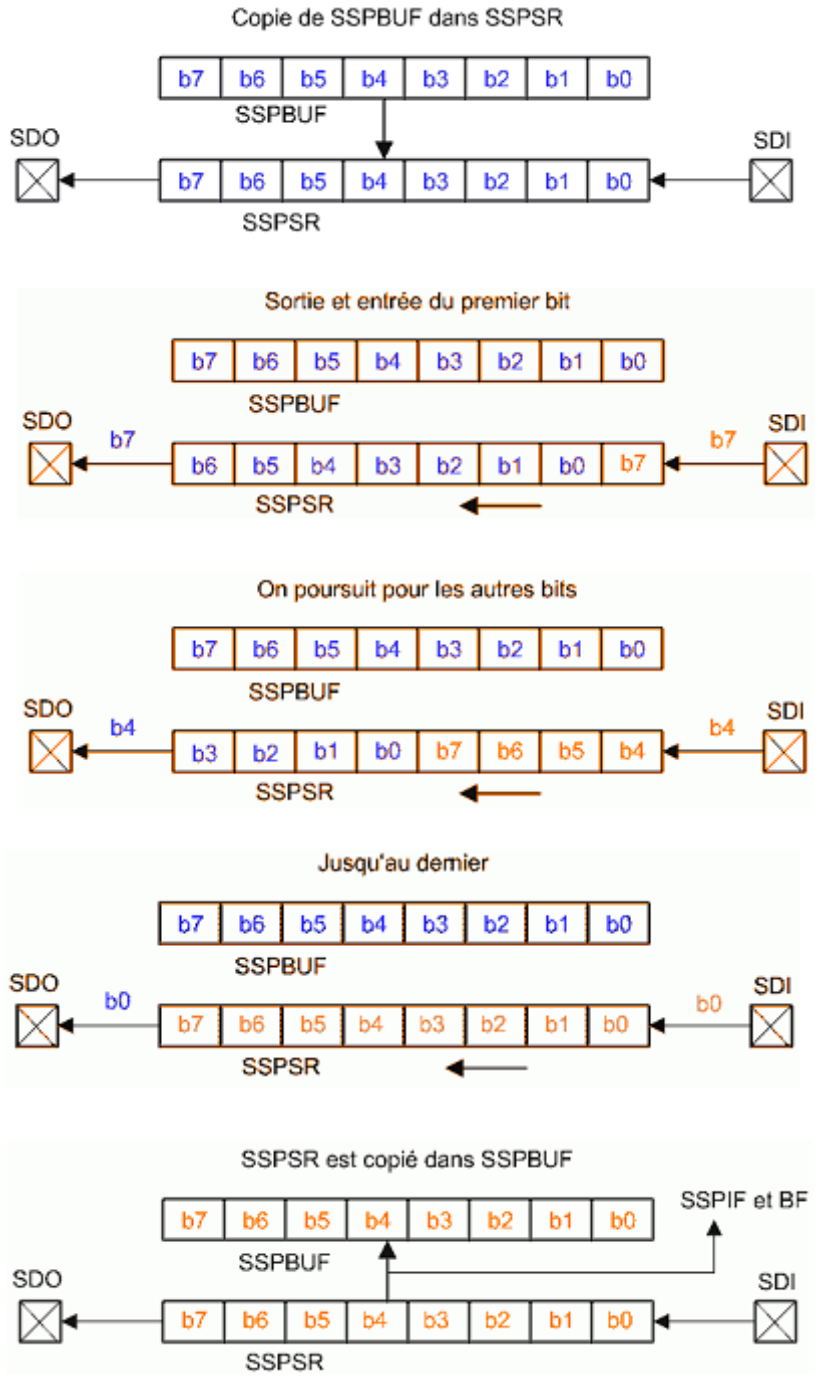
- ✓ On transfère la donnée à émettre dans SSPBUF.
- ✓ Le PIC® copie cette donnée dans SSPSR.
- ✓ On opère 8 décalages vers la gauche, chaque bit sortant est envoyé vers SDO, chaque bit entrant provient de SDI.
- ✓ Le PIC® copie la donnée vers SSPBUF, donc remplace la donnée à émettre par la donnée reçue.
- ✓ A ce moment, le bit BF est positionné, indiquant que SSPBUF contient une donnée à lire, et le flag SSPIF est positionné également pour indiquer la fin du cycle.

Donc, toute émission s'accompagne automatiquement d'une réception, et réciproquement, toute réception nécessite une émission. L'octet éventuellement non nécessaire (reçu ou émis) sera un octet sans signification, souvent appelé « dummy ».

Pour bien comprendre, même si, par exemple, vous n'utilisez pas votre pin SDI, l'émission d'un octet sur SDO s'accompagnera automatiquement de la réception d'un octet fictif (dummy) sur la ligne SDI. A vous de ne pas en tenir compte.

Inversement, pour obtenir la réception d'un octet sur la ligne SDI, le maître est contraint d'envoyer un octet sur sa ligne SDO, même si celle-ci n'est pas connectée. Il enverra donc

également un octet fictif. Voici le tout sous forme de dessins. En rouge les bits reçus, en bleu les bits envoyés.

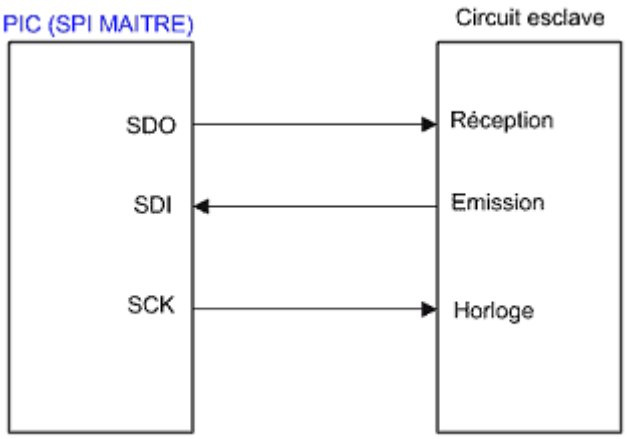



Une dernière petite remarque, avant de passer aux études détaillées : La pin SDO conservera l'état qu'elle avait lors de la fin de la transmission. Etant donné que le bit 0 est transmis en dernier, si l'octet envoyé est un octet pair (b0 = 0), la ligne SDO restera figée à l'état bas jusqu'au prochain envoi d'un octet. Par contre, si l'octet est impair (b0 = 1), cette ligne restera figée à l'état haut dans les mêmes conditions.

## b. Le mode SPI Master.

### *b.1. Mise en œuvre.*

Rien ne vaut une petite figure pour vous montrer l'interconnexion de votre PIC® configurée dans ces conditions. Le schéma de principe suivant illustre une connexion bidirectionnelle sur 3 fils. Pour rappel, ce mode permet la réception et l'émission simultanée (mais non obligatoire) de l'octet utile.




Remarquez qu'il y a bien une ligne par sens de transfert de l'information, et que c'est le maître qui envoie l'horloge. C'est le maître qui gère l'horloge, et c'est lui qui décide de l'instant de la transmission.

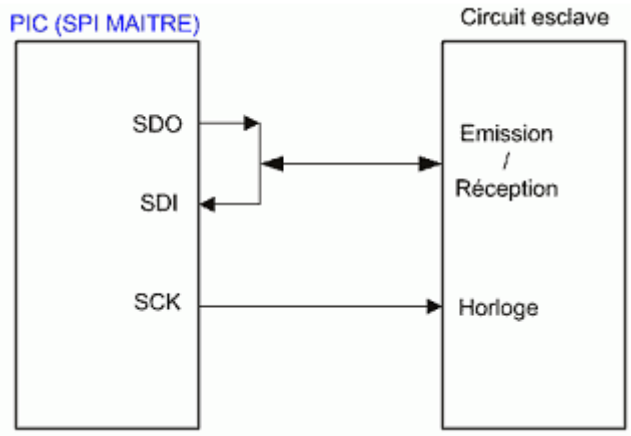
Le circuit esclave peut être aussi bien un autre PIC® qu'un circuit ou un ensemble de circuits quelconque.

Pour le cas où vous n'avez besoin que d'un seul sens de transfert, il suffit de ne pas connecter la ligne non désirée. Souvenez-vous cependant que, malgré que vous n'utilisez pas la ligne en question, au niveau interne au PIC®, il y aura toujours émission et réception simultanées.

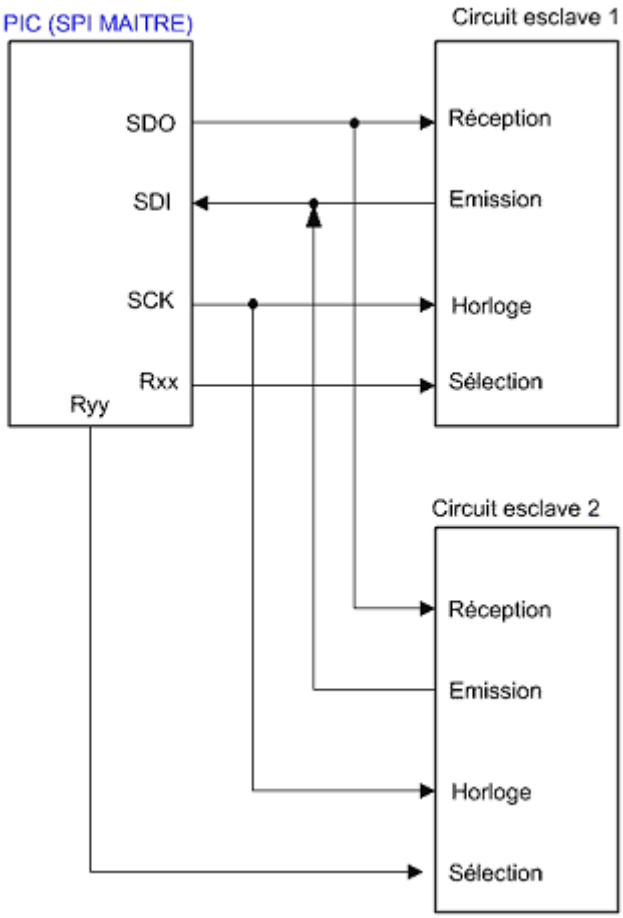
Si vous décidez d'interconnecter 2 PIC®, un étant le maître et l'autre l'esclave, toute transmission se traduit par l'échange du contenu du registre SSPBUF du maître avec celui de l'esclave. Il se peut également que vous désiriez transférer vos données de façon bidirectionnelle, mais en n'utilisant qu'une seule ligne.

La solution sera alors de connecter ensemble SDI et SDO, et de placer SDO en entrée (haute impédance) au moment de la réception d'un octet. Ainsi, SDO n'influencera pas la ligne en mode réception.

Voici le schéma de principe correspondant :




Je termine ces schémas de principe, en indiquant comment notre PIC® peut, par exemple, être connectée à plusieurs esclaves différents.



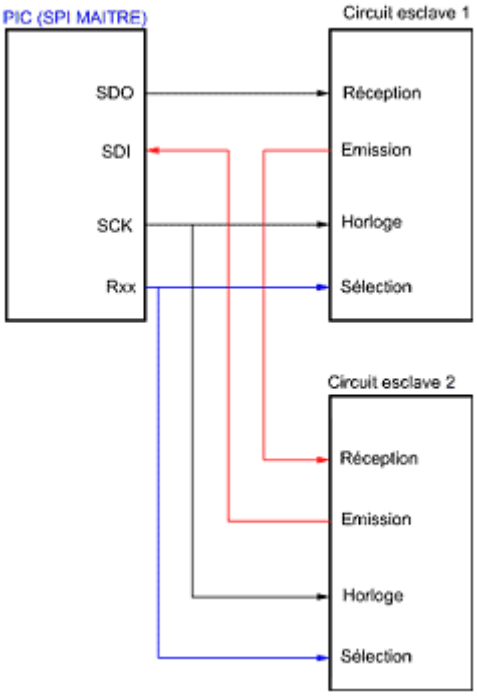

Les pins Rxx et Ryy sont des pins quelconques du PIC® configurées en sortie. Elles sont connectées sur les pins de sélection de la transmission sur chacun des esclaves. Les 3 autres connexions (entrée/sortie/horloge) sont communes à tous les circuits et constituent un bus.

Notez qu'il est impossible qu'un esclave communique avec un autre esclave. Toutes les transmissions sont gérées par le maître. C'est le cas des communications full-duplex, qui nécessitent des lignes croisées. Evidemment, si on croise les lignes entre A et B et entre A et

C, elles ne seront pas croisées entre B et C qui ne pourront donc communiquer (sauf ajout d'électronique de sélection).

Pour communiquer, le PIC® maître sélectionne un esclave via la pin Rxx ou Ryy (n'importe quelle pin du PIC®), puis lance la communication de façon classique. Au niveau du PIC® maître, seule la gestion de ces pins de sélection est supplémentaire par rapport aux transmissions classiques.

Une autre méthode de connexion de plusieurs esclaves en mode SPI est le mode dit « Daisy Chain ». Dans ce mode, les esclaves sont reliés en série.




Vous constaterez que chaque ligne d'émission d'un esclave est reliée sur la ligne de réception du suivant. Le maître envoie ses informations sur le premier esclave et les reçoit du dernier.

Evidemment, ça signifie que chaque esclave va recevoir des données qui ne lui seront pas destinées, il faut donc que les esclaves soient explicitement prévus pour travailler dans ce mode. Il faudra également plusieurs cycles pour que la réponse du premier parvienne au maître, étant donné que cette réponse devra transiter par tous les esclaves suivants.

Ce mode peut fonctionner sans les lignes de sélection (en bleu), ce qui permet d'utiliser plusieurs esclaves avec seulement 3 ou 4 fils.

### *b.2. Le registre SSPSTAT.*

Commençons maintenant l'étude de nos registres pour ce mode particulier.

Attention, je ne parlerai que des bits utilisés dans le fonctionnement décrit actuellement (SPI en mode master). Un bit non décrit ne veut pas dire qu'il n'est pas utilisé dans un autre mode de fonctionnement. En cas de doute, consultez le datasheet.

Ce registre, Synchronous Serial Port STATus register, dispose de 2 bits de configuration (b7 et b6), et de 6 bits d'indication de status (b5 à b0), d'où son nom.

Les 2 premiers cités sont accessibles en lecture et en écriture, les suivants sont accessibles en lecture seule. Le positionnement de ces derniers est automatique en fonction de l'état de la transmission.

**SSPSTAT en mode SPI MASTER** :

- ✓ b7 : SMP : SaMPle bit (0 = milieu, 1 = fin).
- ✓ b6 : CKE : ClocK Edge select (0 = repos vers actif, 1 = actif vers repos).
- ✓ b5 : non.
- ✓ b4 : non.
- ✓ b3 : non.
- ✓ b2 : non.
- ✓ b1 : non.
- ✓ b0 : BF : Buffer Full (0 = buffer vide,1 = octet reçu).

Le bit SMP permet de définir à quel moment du cycle d'horloge on effectue la capture du bit présent sur SDI. Si SMP vaut 0, la capture a lieu au milieu du cycle d'horloge en cours. S'il vaut 1, la capture a lieu à la fin de ce cycle. Bien entendu vous choisirez le mode le plus approprié en fonction du chronogramme de fonctionnement de l'esclave connecté. Vous déterminerez alors à quel moment la donnée présentée par celui-ci sera stable. On peut dire que si l'esclave place sa donnée au début du cycle, le maître devra lire au milieu de ce cycle. Par contre, s'il place sa donnée au milieu, le maître lira celle-ci en fin de cycle.

Le bit CKE détermine quel sens de transition de l'horloge accompagne le placement du bit sur la ligne SDO. Si CKE vaut 0, la ligne d'horloge SCK sera forcée vers son état actif, tandis que si CKE vaut 1, l'horloge passera à l'état de repos au moment de l'apparition de notre bit sur SDO. En milieu de cycle, l'horloge prendra l'état opposé.

Le bit BF est un indicateur (lecture seule) qui, s'il vaut « 1 », indique que le buffer de réception (SSPBUF) contient un octet complet reçu via SDI. Ce bit est à lecture seule, pour l'effacer, vous devez lire le registre SSPBUF, et ce même si vous n'avez aucun usage de l'octet qu'il contient.

### *b.3. Le registre SSPCON.*

Second et dernier registre utilisé pour commander notre mode SPI, le registre SSPCON nous livre maintenant ses secrets.

**SSPCON en mode SPI MASTER**

- ✓ b7 : non.
- ✓ b6 : non.
- ✓ b5 : SSPEN : SSP ENable (1 = module SSP en service).
- ✓ b4 : CKP : ClocK Polarity select bit (donne le niveau de l'état de repos).
- ✓ b3 : SSPM3 : SSP Mode bit 3.

- ✓ b2 : SSPM2 : SSP Mode bit 2.
- ✓ b1 : SSPM1 : SSP Mode bit 1.
- ✓ b0 : SSPM0 : SSP Mode bit 0.

Le bit SSPEN permet tout simplement de mettre le module SSP en service (quel que soit le mode). Les pins SCK, SDO et SDI sont, une fois ce bit validé, déconnectées du PORTC, et prises en charge par le module SSP.

ATTENTION : il vous est toujours, cependant, nécessaire de configurer ces lignes en entrée et en sortie via TRISC.

SCK (RC3) est la pin d'horloge, donc définie en sortie sur notre PIC® maître.

SDI (RC4) est l'entrée des données, donc définie en entrée.

SDO (RC5) est la sortie des données, donc définie en sortie.

Donc TRISC devra contenir, pour le mode SPI master : TRISC = B'xx010xxx'. Souvenez-vous cependant que, si vous utilisez une ligne commune pour l'émission et la réception, vous devrez placer SDO en entrée durant la réception, afin de ne pas bloquer le signal reçu du fait de l'imposition d'un niveau (0 ou 1) sur la ligne par SDO.

CKP détermine ce qu'on appelle la polarité du signal d'horloge. En fait, il détermine si, au repos, la ligne d'horloge se trouve à l'état bas (CKP = 0) ou à l'état haut (CKP = 1). L'état actif étant l'opposé de l'état de repos. Vous trouverez couramment dans les datasheets la notion de « idle » qui précise l'état de repos (à opposer donc à l'état actif).

Les bits SSPMx sont les bits de sélection du mode de fonctionnement. Je vous donne naturellement dans ce chapitre les seules configurations qui concernent le mode SPI MASTER.

**b3 b2 b1 b0 Mode.**

- ✓ 0000 : SPI master, période d'horloge : Tcy = 4 * Tosc.
- ✓ 0001 : SPI master, période d'horloge : Tcy * 4 = Tosc * 16.
- ✓ 0010 : SPI master, période d'horloge : Tcy * 16 = Tosc * 64.
- ✓ 0011 : SPI master, période = sortie du timer 2 * 2

Vous constatez que la sélection d'un de ces modes influence uniquement la fréquence d'horloge. Les 3 premiers modes vous donnent des horloges liées à la fréquence de votre quartz, alors que le dernier de ces modes vous permet de régler votre fréquence d'horloge en fonction de votre timer 2. Dans ce cas, chaque débordement de ce timer2 inverse l'état du signal d'horloge, ce qui explique que le temps d'un cycle complet (2 flancs) nécessite 2 débordements du timer 2.

Bien entendu, vous n'avez qu'un seul timer 2, donc, si vous décidiez d'utiliser ce mode, cela induirait des contraintes dans la mesure où votre timer 2 serait déjà dévolu à un autre usage au sein de votre programme. A vous de gérer ces contraintes.

Si vous utilisez le timer2 comme source d'horloge, le prédiviseur sera actif, mais pas le postdiviseur, qui n'interviendra pas dans le calcul de la fréquence de l'horloge SPI.

Nous avons parlé du début de la transmission, il nous reste à savoir quel événement annonce la fin de la communication. En fait, nous disposons de 2 bits pour indiquer cet état à notre programme :

1. Le bit SSPIF du registre PIR1 est positionné dès que l'échange d'informations est terminé. Ce bit pourra générer une interruption, si celle-ci est correctement initialisée.
2. Le bit BF du registre SSPSTAT sera positionné dès qu'une donnée reçue est inscrite dans le registre SSPBUF. Ce bit ne sera effacé que si on lit la donnée contenue dans ce registre.

### *b.4. Choix et chronogrammes.*

Nous avons vu que nous avons plusieurs bits qui influent sur la chronologie des événements. Je vais tenter de vous montrer en quoi ces choix influent sur la façon dont sont lus et envoyés les octets.

Vous savez maintenant qu'une transmission démarre toujours automatiquement par l'écriture de l'octet à envoyer dans SSPBUF par le maître (pour autant que le module SSP soit en service) et se termine par le positionnement du flag SSPIF du registre PIR1 et par le positionnement du flag BF.

Remarquez que l'émission et la réception commencent, du point de vue des chronologies, par l'apparition des clocks d'horloge. Ceci induit automatiquement l'émission et la réception simultanée des données. Donc, au niveau de votre PIC® maître, vous placerez une donnée dans SSPBUF pour démarrer aussi bien une réception qu'une émission. Si vous n'avez rien à envoyer, il suffira de placer n'importe quoi dans SSPBUF. De même, si l'esclave n'a rien à vous renvoyer en retour, il suffira de ne pas traiter la valeur automatiquement reçue. Donc, du point de vue de votre programme, une émission / réception simultanées d'octets utiles (full-duplex) se traduira par la séquence suivante :

- ✓ On place la donnée à envoyer dans SSPBUF.
- ✓ Une fois SSPIF positionné, on lit dans SSPBUF la valeur reçue simultanément.

Si nous scindons émission et réception d'un octet utile en 2 étapes (half-duplex), nous obtiendrons dans le cas où l'esclave répond à l'interrogation du maître :

- ✓ On place la donnée à envoyer dans SSPBUF.
- ✓ Une fois SSPIF positionné, on ignore la valeur reçue (octet inutile).
- ✓ On place une donnée fictive à envoyer (octet inutile).
- ✓ Une fois SSPIF positionné, on lit la valeur reçue dans SSPBUF.

Ou le contraire, si le maître réceptionne un octet de l'esclave et doit lui répondre :

- ✓ On place une donnée fictive dans SSPBUF.
- ✓ Une fois SSPIF positionné, on traite la valeur lue.

- ✓ On répond en plaçant la réponse dans SSPBUF.
- ✓ Une fois SSPIF positionné, on ignore la valeur fictive reçue dans SSPBUF.

Je vais maintenant vous montrer les chronogrammes. Vous avez déjà compris qu'il y a 4 combinaisons d'horloge possibles en fonction de CKE et de CKP :

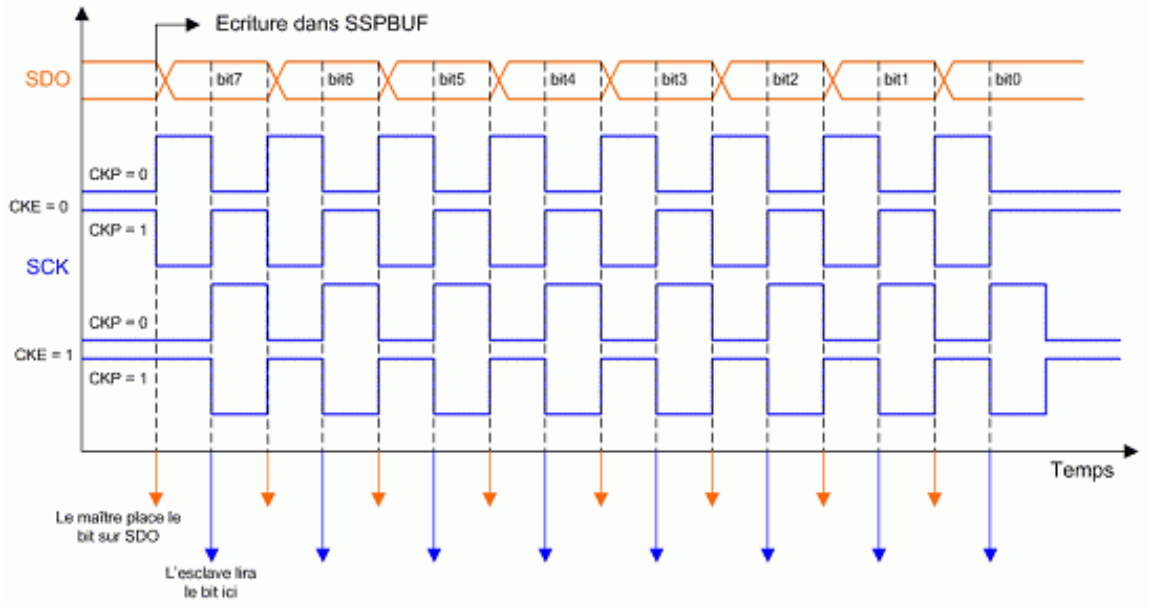
**CKP CKE horloge**.

- ✓ 00 : SCK à 0 au repos, le placement de la donnée induit un flanc montant de SCK.
- ✓ 01 : SCK à 0 au repos, le placement de la donnée induit un flanc descendant de SCK.
- ✓ 10 : SCK à 1 au repos, le placement de la donnée induit un flanc descendant de SCK.
- ✓ 11 : SCK à 1 au repos, le placement de la donnée induit un flanc montant de SCK.

Donc, ceci implique qu'il y a 4 méthodes pour le début de l'émission. Il y a par contre 2 façons de déterminer le moment de la lecture pour la réception, en fonction de SMP.

Soit au milieu du cycle, soit à la fin du cycle. Ceci nous donne 8 modes de fonctionnement possibles au total.

Afin de vous permettre de mieux comprendre, je vous sépare ce chronogramme en 3 parties. D'abord la chronologie de l'émission d'un octet par le maître, et ensuite celles de la réception par le même maître. N'oubliez pas qu'en réalité émission et réception sont simultanées, et donc superposables. Donc, voyons le chronogramme d'émission :



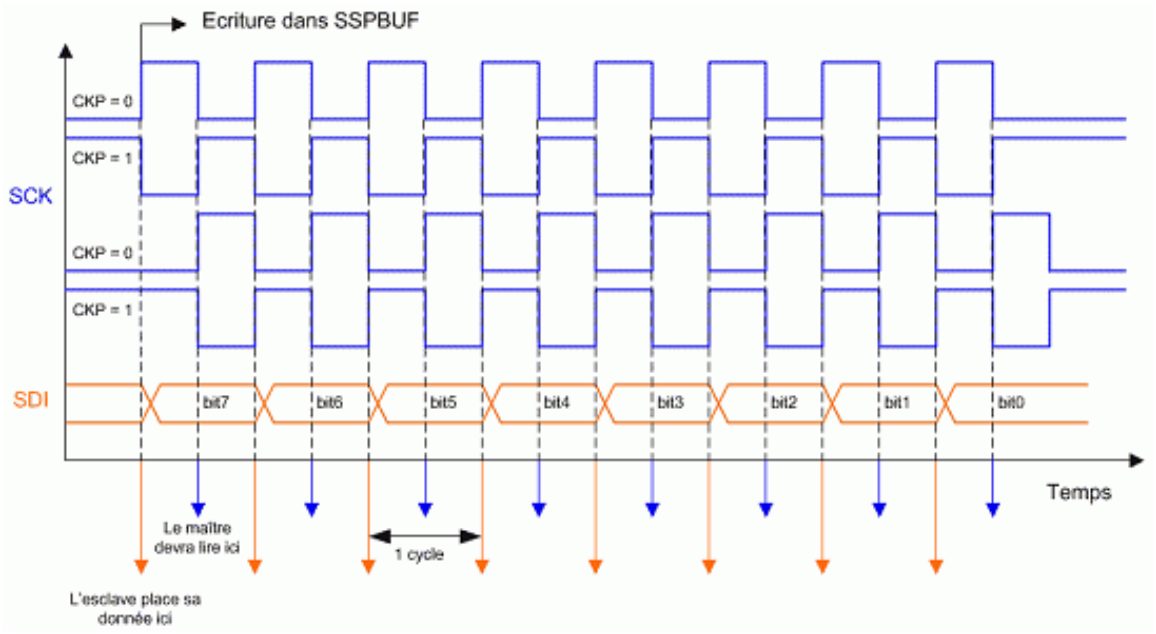

Vous constatez que l'émission des bits sur SDO est synchronisée avec l'horloge SCK, qui peut prendre 4 évolutions différentes. Vous choisirez le mode suivant le fonctionnement de l'esclave connecté. Sur le datasheet de ce dernier, le constructeur vous indiquera quelle forme le signal d'horloge doit prendre au moment de la lecture du bit que votre PIC® aura envoyé. Un cycle est la distance séparant 2 flèches rouges. Vous remarquerez que, quelle que soit la configuration, l'esclave devra toujours lire la donnée du maître au milieu du cycle (flèche bleue) La lecture du bit que vous envoyez, sera impérativement synchronisée par votre

horloge (comme toutes les actions en mode synchrone), et doit se faire dans la zone stable du bit.

Si vous prenez par exemple le mode CKP = 0 et CKE = 0, vous voyez que le bit est placé par le maître sur le flanc montant de SCK. La lecture par l'esclave devra se faire impérativement sur le flanc descendant de SCK, qui est le seul signal présent durant l'état stable du bit.

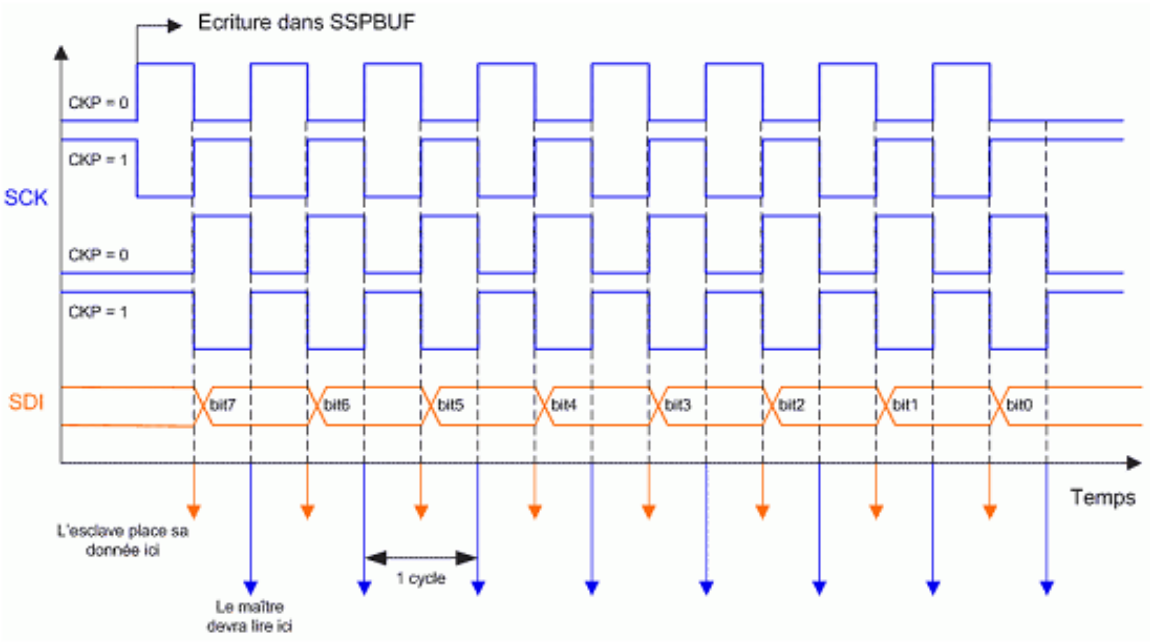
Voyons maintenant la réception d'un octet placé par l'esclave sur la ligne SDI. Nous avons 2 modes possibles, dépendants de SMP. De nouveau, ce choix découle directement de la chronologie de votre composant esclave. Le moment où son fonctionnement provoque le placement de la donnée induit le moment où vous devrez procéder à sa lecture. Je vais de nouveau scinder les 2 cas. Imaginons tout d'abord que l'électronique de l'esclave soit conçue pour que le bit destiné au maître soit placé en début de cycle (donc en même temps que le maître place son propre bit sur SDO).

Nous aurons :




Vous voyez dans ce cas que le choix de l'instant de lecture n'est pas possible. Vous devez lire le bit au milieu du cycle. Ceci vous impose de placer le bit SMP du registre SSPSTAT à 0.

Examinons maintenant le cas où l'esclave « choisit » de placer son bit au milieu du cycle (donc au moment où il procède à la lecture du bit reçu du maître) :




Vous constatez cette fois que, puisque l'esclave place son bit au milieu du cycle, il vous faudra attendre la fin de celui-ci (qui coïncide au début du cycle suivant) pour procéder à la capture du bit concerné. Ceci imposera donc de configurer SMP à « 1 ».

Dans ce dessin, en effet, un cycle est délimité par 2 flèches bleues. Remarquez que bien que vous travailliez en mode « maître », ce mot ne concerne que la génération de l'horloge.

Pour la programmation, vous n'êtes en fait « maître » de rien du tout. Comme c'est vous qui disposez du composant programmable, c'est à vous de vous plier aux exigences du composant « esclave » connecté. C'est donc ce dernier qui va décider de votre façon de travailler, et non l'inverse.

Quand vous travaillerez en mode esclave, vous serez de nouveaux soumis aux exigences du maître connecté. C'est donc toujours vous qui devrez vous soumettre aux exigences matérielles (excepté si vous développez à la fois le logiciel du maître et de l'esclave).

Vous vous souviendrez donc que :

- ✓ Le maître place toujours sa donnée en début de cycle.
- ✓ On en déduit que l'esclave lira toujours la donnée en milieu de cycle.
- ✓ L'esclave peut placer sa donnée, soit en début, soit en milieu de cycle.
- ✓ Ceci implique que le maître lira la donnée reçue, soit en milieu, soit en fin de cycle.

Ces implications sont dues au fait qu'on ne peut lire que lorsque le bit est stable, c'est-à- dire après le moment où le bit est placé, et avant qu'il ne disparaisse au profit du suivant. Comme nous sommes synchronisés à l'horloge, le seul moment possible de lecture se situe donc un demi cycle après le positionnement du bit.

### *b.5. Vitesses de transmission.*

Voyons maintenant, mais j'en ai déjà parlé, des vitesses de transmission disponibles. La sélection de ces vitesses est déterminée par les bits SSPM3 à SSPM0 du registre SSPCON. Vous avez le choix entre une vitesse fixée en fonction de l'horloge principale de votre PIC® et une vitesse fixée en fonction de votre timer2.

Sachant que F = 1/T, avec un quartz de 20Mhz, vous disposerez alors des options suivantes :

- ✓ 0000 donnera une vitesse de Fcycle (fréquence de cycle), soit 5 MHz.
- ✓ 0001 donnera une vitesse de Fcycle / 4, soit 1,25 MHz.
- ✓ 0010 donnera une vitesse de Fcycle / 16, soit 312,5 KHz.
- ✓ 0011 donnera une vitesse dépendant de votre timer 2, suivant la formule suivante :

$$T = (PR2 + 1) \times Tcycle \times prédiviseur \times 2$$

Ou encore :

$$F = \frac{Fcycje}{(PR2 + 1) \times prédiviseur \times 2}$$

Je rappelle que le postdiviseur n'est pas utilisé dans ce cas. La vitesse maximale permise pour la liaison série synchrone est donc de Fosc/4, soit, pour un PIC® cadencé à 20MHz, de 5MHz, 5.000.000 de bits par seconde, ou encore 5.000.000 bauds (5MBauds).

Vous constatez qu'il s'agit d'une vitesse assez importante, qui nécessite des précautions de mise en œuvre (qualité et longueur des liaisons par exemple). La vitesse minimale est celle utilisant le timer 2 avec prédiviseur à 16. Nous aurons, pour un quartz de 20MHz, une vitesse minimale de Fcy / (2 * prédiviseur * (PR2+1)), soit 5Mhz / (2 * 16 * 256) = **610,3 bauds**.

Ceci vous donne une grande flexibilité dans le choix de la fréquence de l'horloge, fréquence qui dépend une fois de plus des caractéristiques de l'esclave connecté.

### c. Initialisation du mode SPI Master.

Voici petit résumé sur les procédures d'initialisation de notre module :

- ✓ On initialise SMP en fonction du moment de capture de la donnée reçue (milieu ou fin du cycle).
- ✓ On initialise CKP en fonction de l'état de repos de la ligne d'horloge (polarité).
- ✓ On sélectionne CKE suivant le flanc d'horloge souhaité au moment de l'écriture d'un bit sur SDO.
- ✓ On choisit la vitesse de l'horloge suivant SSPMx.
- ✓ On met le module en service via SSPEN.
- ✓ On configure SCK et SDO en sortie, SDI en entrée, via TRISC.

Le démarrage d'une émission (et donc de la réception simultanée) s'effectue simplement en plaçant un octet dans SSPBUF. La fin de l'émission (et donc de la réception) s'effectuera en vérifiant le positionnement de SSPIF, en gérant l'interruption SSP correspondante (si configurée) ou en vérifiant le positionnement du bit BF, méthode moins souvent utilisée.

### d. Le mode sleep.

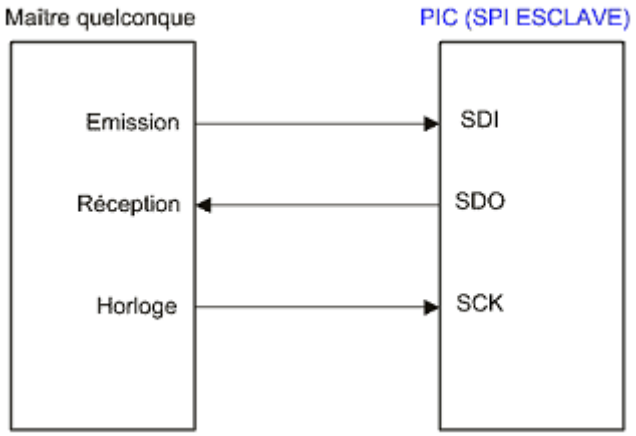
Le passage en mode « sleep » induit l'arrêt à la fois de l'horloge principale du PIC®, et du timer 2. Vous n'avez donc plus aucune possibilité de générer l'horloge. La transmission sera donc suspendue, et reprendra d'où elle se trouvait arrêtée, une fois le PIC® réveillé par un autre événement externe (excepté un reset, bien sûr).

## V.8.3. Le mode SPI SLAVE.

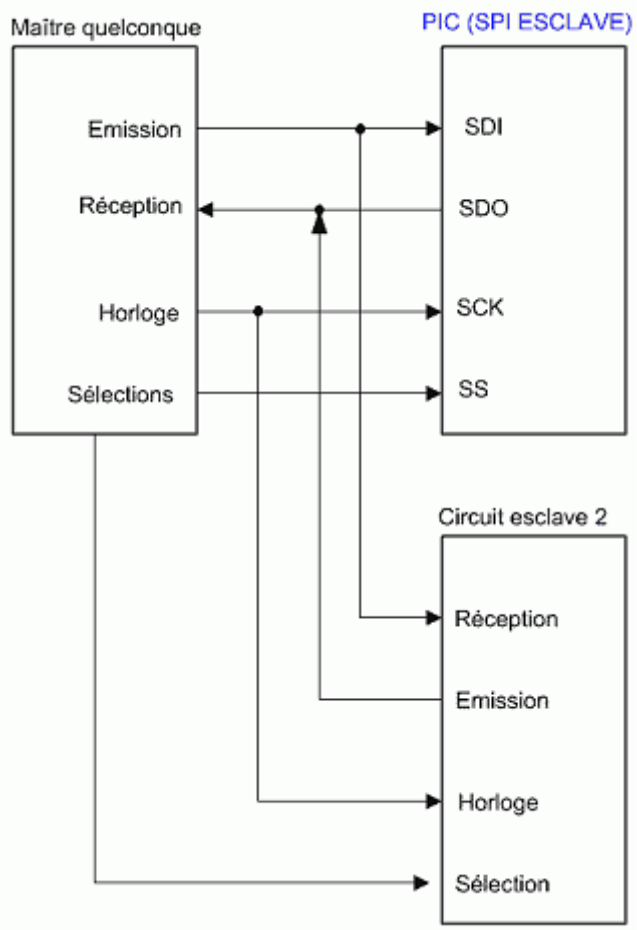
Comme vous l'aurez déjà compris depuis longtemps, ce mode (slave ou esclave) présente la particularité de subir l'horloge de synchronisation au lieu de l'imposer. Ceci va induire des contraintes différentes, contraintes paramètrées de nouveau par les mêmes registres que pour le mode « master ».

### a. Mise en œuvre.

Je vais maintenant vous donner les 2 configurations possibles de votre PIC® connectée en mode SPI esclave. Le premier cas est donné si le PIC® est le seul esclave du système. Nul besoin, alors, à priori, de le sélectionner en particulier. Nous verrons cependant que cela n'est pas toujours vrai.




Bien entendu, on trouve aussi le cas pour lequel votre PIC® n'est pas le seul esclave du système. Dans ce cas, il faut bien que votre PIC® sache quand c'est à lui que le maître s'adresse.




Vous remarquez la présence de la pin *SS*, configurée en entrée. Cette pin, lorsqu'elle est placée au niveau bas, indique au PIC® que la communication en cours lui est destinée. Il prend alors en compte les fluctuations de l'horloge.

Si le PIC® a été configuré pour tenir compte de la pin *SS*, et que celle-ci se trouve à l'état haut, le PIC® ignorera tout signal en cours sur la ligne d'horloge, et donc ne réceptionnera ni n'enverra aucun bit.

Notez que vous disposez également, comme pour le mode « maître », de la possibilité d'interconnecter SDO et SDI pour établir une liaison half-duplex. Les méthodes de gestion et les limitations seront donc les mêmes.

### b. Le registre SSPSTAT.

De nouveau, peu de bits utilisés pour ce mode dans ce registre. Le bit SMP doit être forcé à 0. C'est logique, étant donné que nous avons précisé dans l'étude du fonctionnement en maître que l'esclave est obligé de lire sa donnée au milieu du cycle.

***SSPSTAT en mode SPI SLAVE.***

- ✓ b7 : doit être positionné à 0 (lecture en milieu de cycle).
- ✓ b6 : CKE : ClocK Edge select (0 = repos vers actif, 1 = actif vers repos).
- ✓ b5 : non.

- ✓ b4 : non.
- ✓ b3 : non.
- ✓ b2 : non.
- ✓ b1 : non.
- ✓ b0 : BF : Buffer Full (0 = buffer vide, 1 = octet reçu).

CKE a strictement la même fonction que pour le mode master, je ne m'attarderai donc pas.

- ✓ Si CKE vaut 0, la transition vers l'état actif détermine le début du cycle (instant où le maître place sa donnée).
- ✓ Si CKE vaut 1, la transition vers l'état de repos détermine le début du cycle.

BF indique que le registre SSPBUF contient une donnée reçue en provenance du maître. La seule façon d'effacer ce bit est de lire le registre SSPBUF (même si vous n'avez aucun besoin de la donnée qu'il contient). Si vous ne lisez pas SSPBUF, la prochaine réception d'un octet donnera lieu à une erreur « overflow », et l'octet qui sera alors reçu ne sera pas transféré dans SSPBUF. Il sera donc perdu.

En mode esclave, il est donc impératif de lire chaque octet reçu, sous peine d'arrêt de la réception des octets.

### c. Le registre SSPCON.

Ce registre utilise, pour le mode esclave, deux bits de plus que pour le mode maître.

***SSPCON en mode SPI SLAVE***.

- ✓ b7 : WCOL : Write COLlision detect bit.
- ✓ b6 : SSPOV : SSP receive Overflow indicator bit (1 = perte de l'octet reçu).
- ✓ b5 : SSPEN : SSP ENable (1 = module SSP en service).
- ✓ b4 : CKP : ClocK Polarity select bit (donne le niveau de l'état de repos).
- ✓ b3 : SSPM3 : SSP Mode bit 3.
- ✓ b2 : SSPM2 : SSP Mode bit 2.
- ✓ b1 : SSPM1 : SSP Mode bit 1.
- ✓ b0 : SSPM0 : SSP Mode bit 0.

Le bit SSPOV permet de détecter une erreur de type « overflow ». Cette erreur intervient (SSPOV = 1) si la réception terminée d'un octet induit la tentative de transfert de cette donnée depuis SSPSR vers SSPBUF, alors que votre programme n'a pas encore été lire la donnée précédente contenue dans SSPBUF (bit BF toujours positionné).

Cette situation intervient en général lors de la détection par pooling de la réception d'un octet. La méthode d'interruption étant moins sujette à ce type d'erreur (réaction immédiate à la réception d'un octet). Ce bit reste à 1 jusqu'à ce qu'il soit remis à 0 par votre programme.

Il vous incombera également de lire le registre SSPBUF, afin d'effacer le bit BF qui avait provoqué l'erreur, sans cela, la prochaine réception d'un octet donnerait de nouveau lieu à la même erreur. En cas d'erreur de ce type, l'octet contenu dans SSPSR n'est pas copié dans

SSPBUF. C'est donc le dernier octet reçu (contenu dans SSPSR) qui est perdu, celui contenu dans SSPBUF ne sera pas « écrasé ».

Le positionnement de l'indicateur WCOL vous informe que vous avez écrit un nouvel octet à envoyer dans SSPBUF, alors même que l'émission de l'octet précédemment écrit est toujours en cours. Il vous appartient d'effacer cet indicateur une fois l'erreur traitée.

Concernant les bits SSPMx, ils vont de nouveau déterminer les modes de fonctionnement possibles du SPI en mode esclave. Plus question de vitesse, cette fois, puisqu'elle est déterminée par le maître.

**SSPMx Mode.**

- ✓ 0100 SPI esclave, la pin *SS* permet de sélectionner le port SPI.
- ✓ 0101 SPI esclave, la pin *SS* est gérée comme une pin I/O ordinaire.

Souvenez-vous que la pin *SS* (optionnelle) peut servir dans le cas des esclaves multiples. Dans ce cas, cette entrée valide le signal d'horloge reçu. Elle permet donc de ne réagir que lorsque le PIC® est réellement concerné. Le registre TRISC devra être correctement configuré. Nous allons voir qu'en fait l'utilisation de la pin *SS* est souvent moins facultative qu'elle ne le semble au premier abord.

### d. Les autres registres concernés.

De nouveau, nous allons retrouver nos registres SSPSR et SSPBUF, sur lesquels je ne reviendrai pas.

- ✓ Concernant le registre TRISC, qui définit entrées et sorties, nous pouvons déjà dire que les pins SDO et SDI conservent leur fonction, alors que SCK subit maintenant l'horloge imposée par le maître, et donc devient une entrée.
- ✓ TRISA,5 permet de définir SS en entrée, pour le cas où vous souhaitez utiliser cette possibilité.
- ✓ SCK (RC3) est la pin d'horloge, donc définie en entrée sur notre PIC® esclave.
- ✓ SDI (RC4) est l'entrée des données, donc définie en entrée.
- ✓ SDO (RC5) est la sortie des données, donc définie en sortie.
- ✓ SS (RA5) est l'entrée de sélection du module SPI (optionnelle).

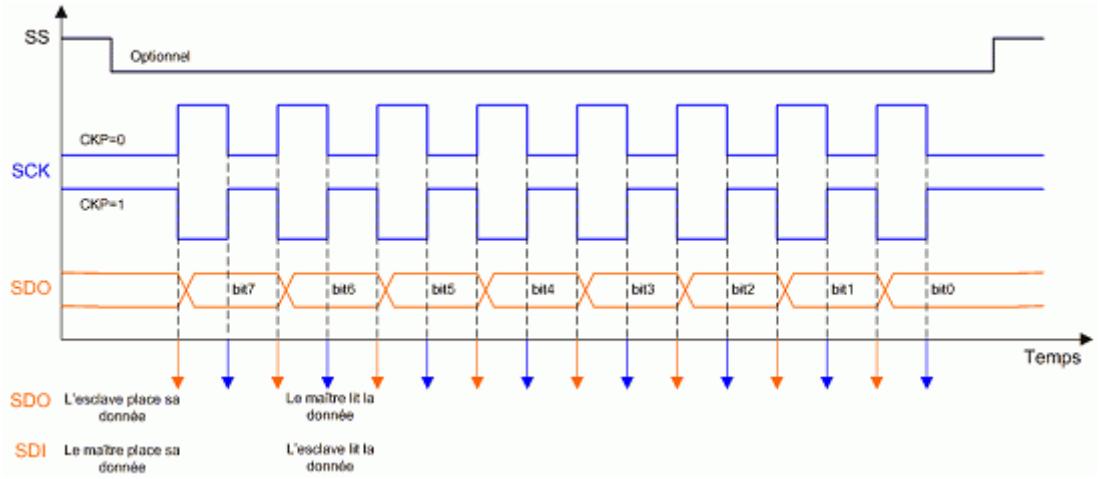
Donc TRISC devra contenir, pour le mode SPI slave : TRISC = B'xx011xxx ' Si vous désirez utiliser SS, vous devrez placer TRISA,5 à « 1 ».

Notez, de plus, que dans ce cas vous devrez aussi définir RA5 comme entrée numérique en configurant correctement le registre ADCON1 (comme expliqué dans le chapitre sur le convertisseur A/D).

### e. Choix et chronogrammes.

Nous pouvons distinguer ici 2 chronogrammes différents, dépendant de l'état de CKE. Pour chacun des niveaux de CKE, nous avons 2 signaux d'horloge possibles. Voyons tout d'abord le cas le plus simple, celui pour lequel CKE = 0.

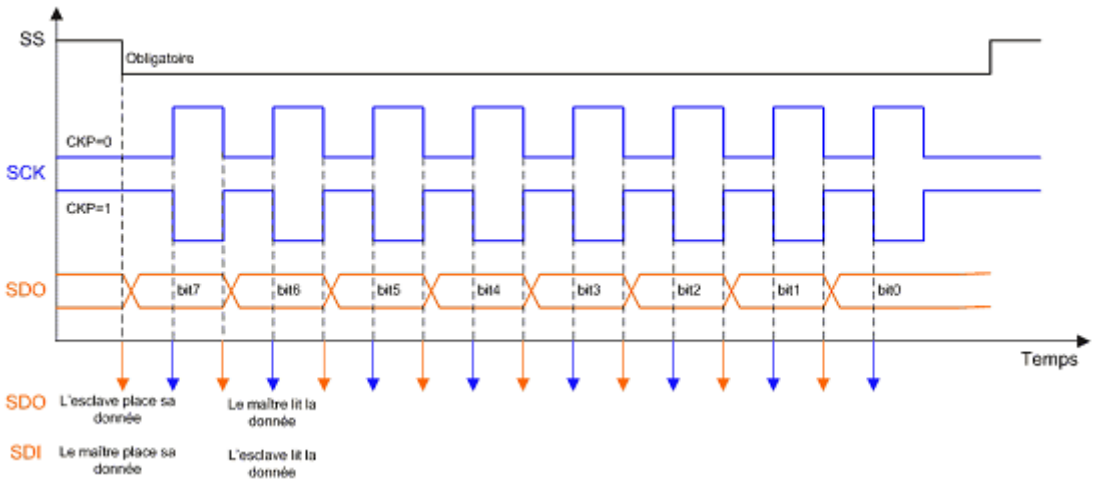
Souvenez-vous que vous disposez de la possibilité d'utiliser la pin SS. Dans ce cas, elle devra être placée à 0 au début de la transmission, faute de quoi, le PIC® esclave ne réagirait pas à l'horloge envoyée par le maître. Cette fois, ce n'est plus le placement d'une valeur dans SSPBUF qui déclenche le transfert. C'est en effet le PIC® maître qui décide du début de cette transmission. Si SCK vaut 0, le PIC® esclave détecte le début de la transmission au moment de la détection du premier flanc actif de SCK.




Le PIC® esclave placera sa donnée sur sa ligne SDO au moment de la première transition entre état de repos et état actif de SCK (si l'entrée *SS* optionnelle est utilisée, il faudra de plus qu'elle se trouve au niveau bas). Le maître est sensé faire de même sur la ligne SDI de notre PIC® esclave. Donc, la lecture de la ligne SDI s'effectuera lors de la transition entre état actif et état de repos de SCK.

Tous les instants de transition sont parfaitement déterminés par l'horloge. La ligne *SS*, qui valide la communication est optionnelle, et dépend du mode choisi (avec ou sans gestion de *SS*). Si le mode sans gestion de *SS* est choisi, l'état de la ligne *SS* n'aura aucune importance. Si, par contre, le mode avec gestion de *SS* est programmé, alors les transferts ne s'effectueront que si cette ligne est placée au niveau bas par le maître.

Voyons maintenant ce qui se passe si nous choisissons de travailler avec CKE = 1.



Dans ce cas, le placement de la valeur s'effectue sur la transition entre le niveau actif et le niveau de repos de SCK. Malheureusement, il va de soi que la première transition, concernant le bit 7, est impossible à détecter.

En effet, la ligne étant au repos, la première transition modifierait la ligne SCK depuis son niveau actuel (repos) vers le niveau repos. Il n'y a donc pas de transition. Rien ne permet donc à l'esclave de savoir que la transmission a commencé. Il lui est donc impossible de placer son bit b7 sur la ligne.

La solution trouvée est de se servir de la ligne de sélection *SS*, dont le passage à l'état bas remplace la première transition manquante de SCK. Le premier bit sera donc placé par l'esclave au moment de la sélection de celui-ci via la pin *SS*. Il va donc de soi que dans cette configuration, la ligne *SS* devient indispensable au fonctionnement de l'ensemble. Donc, pour résumer, si vous utilisez la communication en mode SPI esclave et que vous choisissez CKE = 1, alors vous devrez choisir le mode SSPMX = 0100, qui met la pin *SS* en service.

### f. Vitesses de transmission.

Je ne vais pas entrer ici dans trop de détails. Si vous décidez d'interfacer 2 PIC® identiques ensemble, vous pouvez estimer que la vitesse maximale en mode master est égale à la fréquence maximale en mode slave. Si vous utilisez un autre composant externe comme maître, il faudra vous assurer que ses signaux présentent certaines caractéristiques compatibles avec le datasheet du PIC® (par exemple, le temps de maintien de la ligne SDI après le moment de la lecture). Ceci vous permettra de calculer avec précision la vitesse maximale commune entre les 2 composants.

### g. Initialisation du mode SPI SLAVE.

Voici les procédures d'initialisation de notre module :

- ✓ On initialise CKP en fonction de l'état de repos de la ligne d'horloge (polarité).
- ✓ On sélectionne CKE suivant le flanc d'horloge correspondant au moment de l'écriture par le maître d'un bit sur SDI.
- ✓ On choisit la mise en service ou non de *SS*, via SSPMx.
- ✓ On met le module en service via SSPEN.
- ✓ On configure SDO en sortie, SDI et SCK en entrée, via TRISC.
- ✓ On configure éventuellement *SS* (RA5) en entrée via TRISA et ADCON1.

Le démarrage d'un transfert s'effectue simplement lors de la première transition de SCK (si CKE = 0) ou lors du flanc descendant de SS (si CKE = 1). La fin du transfert s'effectuera en vérifiant le positionnement de SSPIF, en gérant l'interruption SSP correspondante (si configurée) ou en vérifiant le positionnement du bit BF.

### h. Le mode sleep.

Le passage en mode sleep n'arrête pas l'horloge SCK, puisque cette dernière est générée par le maître. Le PIC® en mode esclave est dès lors parfaitement capable de recevoir et d'émettre des données dans ce mode. Chaque fin d'émission/réception pourra alors réveiller le PIC®, qui pourra lire l'octet reçu et préparer le suivant à émettre lors du prochain transfert.

### i. Sécurité de la transmission.

Au niveau du PIC® en esclave risque de se poser un problème. En effet, supposons qu'on perde une impulsion de l'horloge SCK, ou, au contraire qu'un parasite ajoute une fausse impulsion lors du transfert d'un octet. Pour prendre exemple du premier cas, voici ce qui va se passer :

- ✓ Le maître envoie son octet et 8 impulsions d'horloge.
- ✓ L'esclave ne reçoit que 7 impulsions, et donc délivre (du point de vue du maître) 2 fois le même bit. Il lui reste donc le bit 0 à envoyer. Notez que le maître ne peut savoir que l'esclave n'a envoyé que 7 bits utiles, de même l'esclave ne peut savoir que le maître a procédé à la réception de 8 bits.
- ✓ Le maître envoie l'octet suivant et 8 nouvelles impulsions d'horloge.
- ✓ L'esclave croit alors que la première impulsion concerne le bit 0 du transfert précédent, et interprète les 7 suivantes comme les 7 premières du transfert courant. Dans cette seconde transaction, le maître aura donc reçu un octet composé du bit 0 de l'octet précédent, complété par les bits 7 à 1 de l'octet courant.

Toutes les communications seront donc décalées. Il importe donc de permettre à l'esclave de se resynchroniser afin qu'une erreur de réception de l'horloge n'ait d'influence que sur un seul octet. Ceci peut s'effectuer de diverses façons, mais la plus simple est le recours systématique à l'utilisation de la pin *SS*. En effet, chaque fois que cette pin va repasser à l'état haut, le module SPI va se remettre à 0, et, dès lors, saura que la prochaine impulsion d'horloge concerne un nouvel octet. Je conseille de ce fait de toujours utiliser en mode SPI esclave, si c'est possible, le mode qui met en service la gestion de la pin *SS*.

## V.9. Le bus I²C.

### V.9.1. Caractéristiques fondamentales.

Le bus I²C permet d'établir une liaison série synchrone entre 2 ou plusieurs composants. Il a été créé dans le but d'établir des échanges d'informations entre circuits intégrés se trouvant sur une même carte. Son nom, d'ailleurs, traduit son origine : Inter Integrate Circuit, ou I.I.C., ou plus communément I²C (I carré C). Ce bus est le descendant du CBUS, qui est de moins en moins utilisé. Son domaine d'application actuel est cependant bien plus vaste, il est même, par exemple, utilisé en domotique. Il comporte des tas de similitudes avec le SMBUS d'Intel (System Management BUS).

Le bus I²C est constitué de 2 uniques lignes bidirectionnelles :

1. La ligne SCL (Serial Clock Line), qui, comme son nom l'indique, véhicule l'horloge de synchronisation.
2. La ligne SDA (Serial DAta line), qui véhicule les bits transmis.

Il est important de rappeler que :

- ✓ la ligne SCL est gérée par le maître (nous verrons que par moment, l'esclave peut prendre provisoirement le contrôle de la ligne).

- ✓ la ligne SDA, à un moment donné, est pilotée par celui qui envoie une information (maître ou esclave).

Tous les circuits sont connectés sur ces 2 lignes. Il existe 2 sortes de circuits pouvant être connectés :

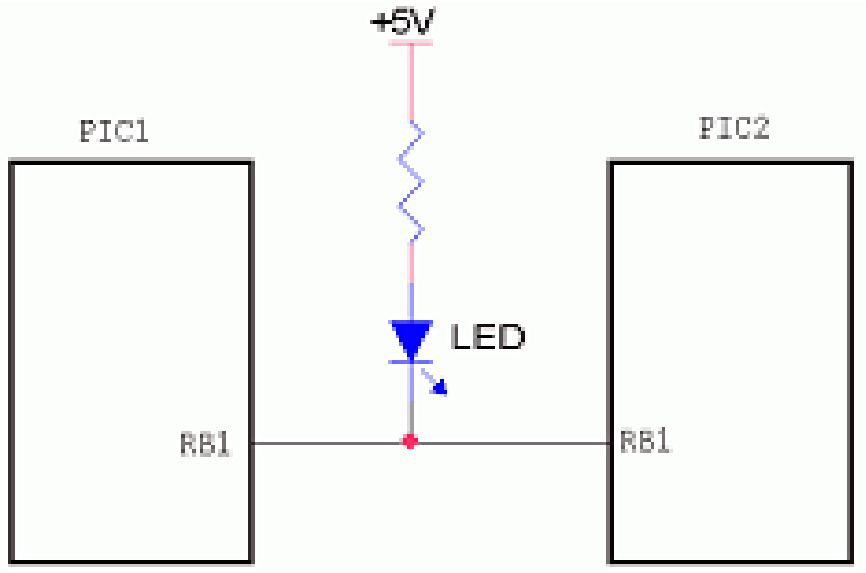
- ✓ Les circuits maîtres, qui dirigent le transfert et pilotent l'horloge SCL.
- ✓ Les circuits esclaves, qui subissent l'horloge et répondent aux ordres du maître.

Chacun de ces 2 types peut émettre et recevoir des informations. Une particularité est qu'on peut placer plusieurs maîtres sur le même bus I²C. Ceci implique que, sans précautions, si 2 maîtres désirent prendre le contrôle du bus en même temps, on encourrait, sans précautions particulières, 2 risques :

- ✓ La destruction de l'électronique des circuits, pour le cas où l'un d'entre eux impose un niveau haut et l'autre un niveau bas (court-circuit).
- ✓ La corruption des données destinées ou en provenance de l'esclave.

Fort heureusement, vous vous doutez bien que Philips (l'inventeur de l'I²C) a trouvé des solutions pour parer à ces éventualités.

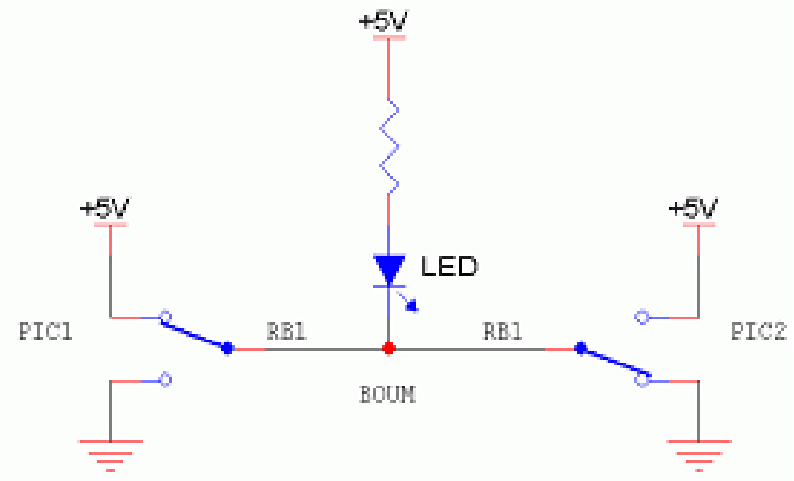
Pour la partie électronique, la parade est simple. On travaille avec des étages de sortie qui ne peuvent imposer qu'un niveau 0, ou relâcher la ligne, qui remonte d'elle-même au niveau 1 via des résistances de rappel. De cette façon, on n'aura jamais de court-circuit, puisque personne ne peut placer la tension d'alimentation sur la ligne. Ces étages sont des montages que vous trouverez dans la littérature sous la dénomination de « collecteur ouvert » ou de « drain ouvert » suivant la technologie utilisée. La pin RA4 de votre PIC® utilise d'ailleurs la même technique. Notez que vous pouvez faire de même, si un jour vous décidez de piloter une ligne à partir de 2 PIC®, par exemple. Imaginons le montage suivant :




Imaginons que vous désiriez allumer la LED depuis n'importe lequel des 2 PIC®. Si vous créez les sous-routines suivantes dans chaque PIC® :

RB1 = 1 ; RB1 = 1.

Si maintenant, votre PIC1 tente d'éteindre la LED, il placera +5V sur la ligne RB1. Si au même instant votre PIC2 désire allumer la LED, il placera 0V sur la ligne RB1. Et nous voici en présence d'un beau court-circuit, avec risque de destruction des 2 PIC®. Si on regarde de façon schématique ce qui se passe au niveau des 2 PIC®, on voit :

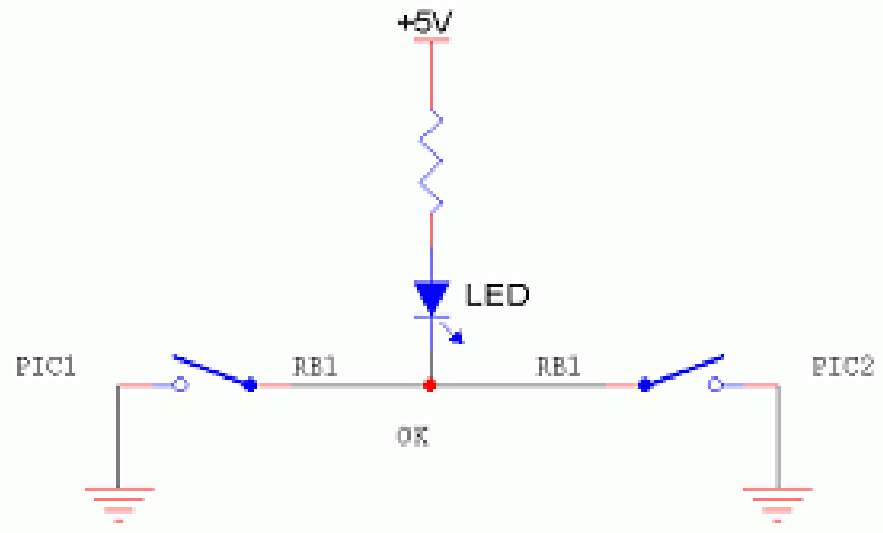



Le courant passe directement depuis le +5V du PIC1 vers le 0V du PIC2, ce qui provoque un court-circuit. Il va de soi que si on avait utilisé la pin RA4 (prévue pour cet usage) on n'aurait pas rencontré ce problème. Mais il existe une solution simple permettant d'utiliser n'importe quelle pin, et qui consiste à ne jamais autoriser le PIC® à envoyer un niveau haut. On modifie donc le programme comme suit :

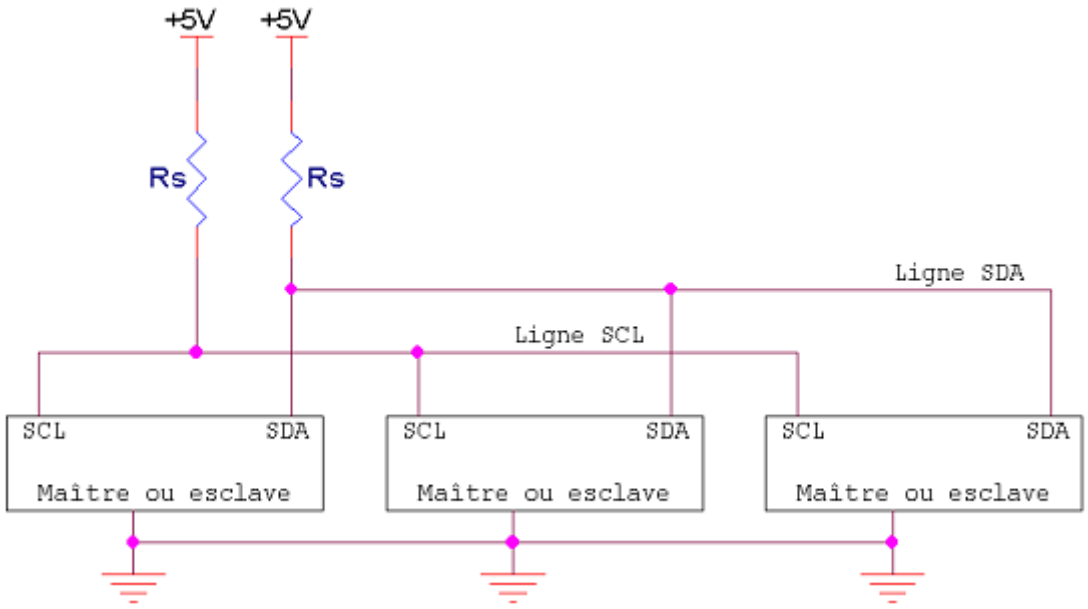
Allumage : RB1 en sortie et RB1 = 0.

Extinction : RB1 en entrée (haute impédance).

On obtient alors un schéma équivalant théorique du type :



Si on enclenche un ou l'autre interrupteur, on allume la LED (on force la ligne RB1 à l'état bas). Si on tente d'éteindre la LED, il faut que les 2 PIC® libèrent la ligne. C'est exactement ce qui se passe pour le bus I²C. Chaque circuit peut forcer la ligne SCL ou SDA à 0, mais aucun circuit ne peut la forcer à l'état haut. Elle repassera à l'état haut via les résistances de rappel (pull-up) si tous les circuits connectés ont libéré la ligne. Le schéma d'interconnexion des circuits sur un bus I²C est donc le suivant :




***ATTENTION*** : Souvenez-vous que la masse de tous ces circuits, qui sert de référence, doit évidemment être commune. De ce fait, si ces circuits se situent sur des cartes différentes, ou ne partagent pas la même référence, il vous faudra une ligne supplémentaire dans votre connexion, afin de transmettre la tension de référence. On parle donc couramment de liaison à 2 lignes, mais en réalité 3 lignes sont nécessaires pour communiquer.

## V.9.2. Les différents types de signaux.

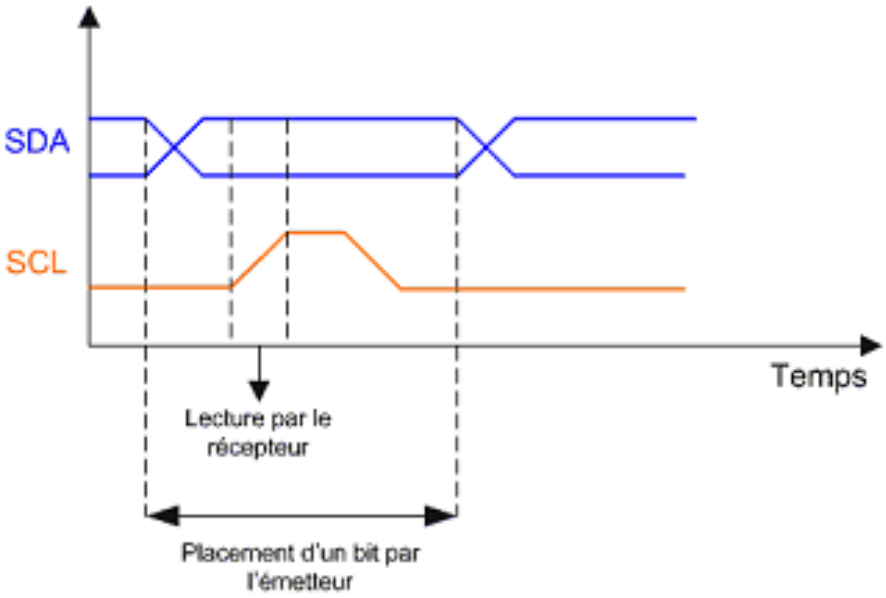
Nous allons voir que cette norme joue intelligemment avec les niveaux présents sur les 2 lignes, afin de réaliser diverses opérations. Plutôt que de vous donner directement un chronogramme de transmission, je vais scinder les différentes étapes. De cette façon je pense que ce sera beaucoup plus simple à comprendre.

### a. Le bit « ordinaire ».

Tout d'abord, la méthode utilisée pour écrire et recevoir des bits est différente de celle utilisée dans l'étude de notre module SPI. En effet, pour ce dernier, un bit était émis sur un flanc de l'horloge, et était lu sur le flanc opposé suivant de la dite horloge.

Au niveau du bus I²C, le bit est d'abord placé sur la ligne SDA, puis la ligne SCL est placée à 1 (donc libérée) durant un moment puis forcée de nouveau à 0. L'émission du bit s'effectue donc sans aucune correspondance d'un flanc d'horloge. Il est lu lors du flanc montant de cette horloge.

Notez qu'il est interdit de modifier la ligne SDA durant un niveau haut de SCL. La dérogation à cette règle n'est valable que pour les séquences dites « de condition » que vous allons voir plus loin. Voici à quoi ressemble l'émission et la lecture d'un bit :

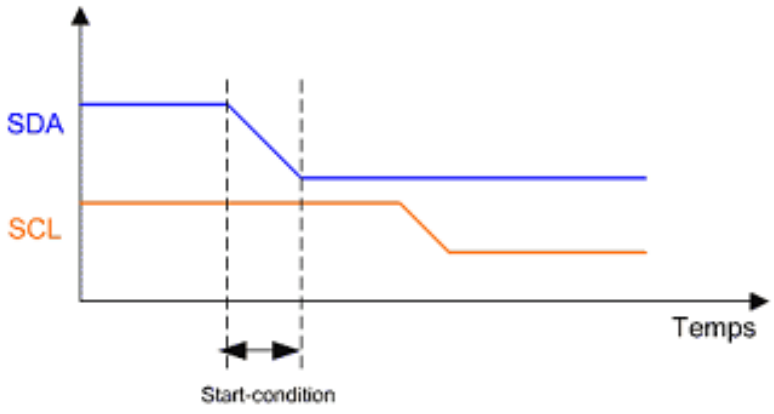



Vous constatez que le bit présent sur la ligne SDA doit être présent avant la montée du signal SCL, et continuer un certain temps avant sa redescente.

### b. Le start-condition.

Comme pour tout signal synchrone, le protocole ne définit pas de start et de stop-bit. Mais il intègre toutefois les notions de « start-condition » et de « stop-condition ». Ces séquences particulières, obtenues en modifiant la ligne SDA alors que la ligne SCL est positionnée à l'état haut permettent de définir début et fin des messages.

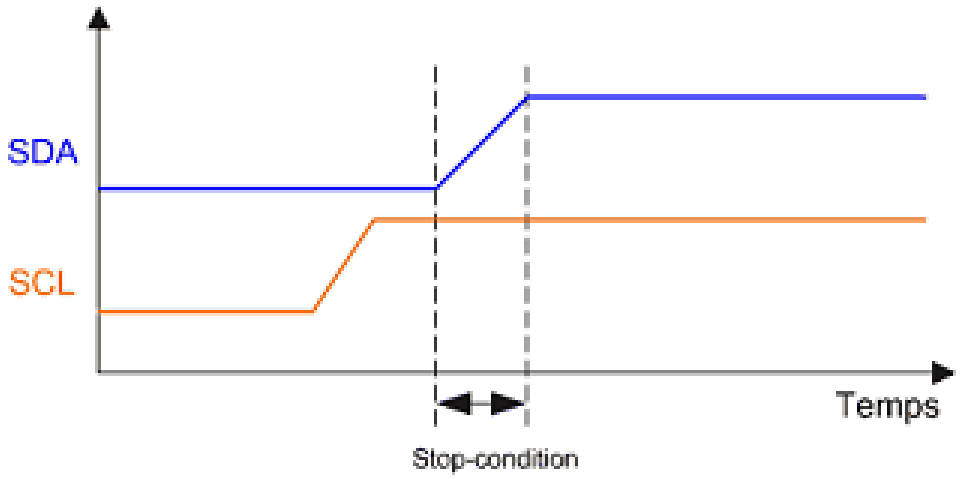
Nous verrons que ceci est indispensable pour repérer le premier octet du message, qui a un rôle particulier. Si on se souvient qu'au repos, SCL et SDA se trouvent relâchés, et donc à l'état haut, le start-condition (symbole conventionnel : S) est réalisé simplement en forçant la ligne SDA à 0, tout en laissant la ligne SCL à 1. La ligne SCL est ensuite mise à 0 pour préparer l'émission du premier bit.



Il existe un dérivé de cette condition, appelé « repeated start condition », qui est utilisé lorsqu'un message en suit directement un autre. Il s'agit donc en fait d'un second star-tcondition au sein d'un même message. Nous verrons son intérêt plus tard. Sachez à ce niveau qu'un « repeated start-condition » peut être considéré comme un « start-condition » sur une ligne déjà active.

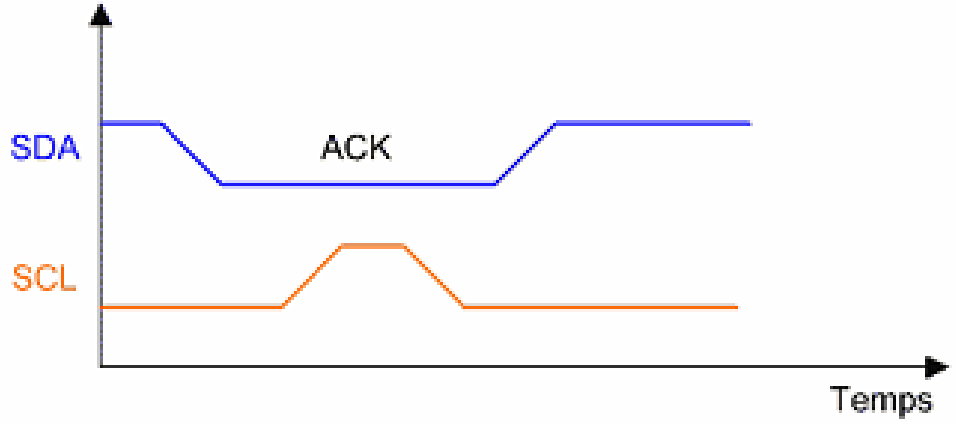
### c. Le stop-condition.

Nous venons d'y faire allusion. Cette condition indique la fin du message en cours. Elle remet les lignes SDA et SCL au repos, mais en respectant la chronologie suivante : La ligne SDA est ramenée à 1, alors que la ligne SCL se trouve déjà à 1. Voici ce que cela donne :

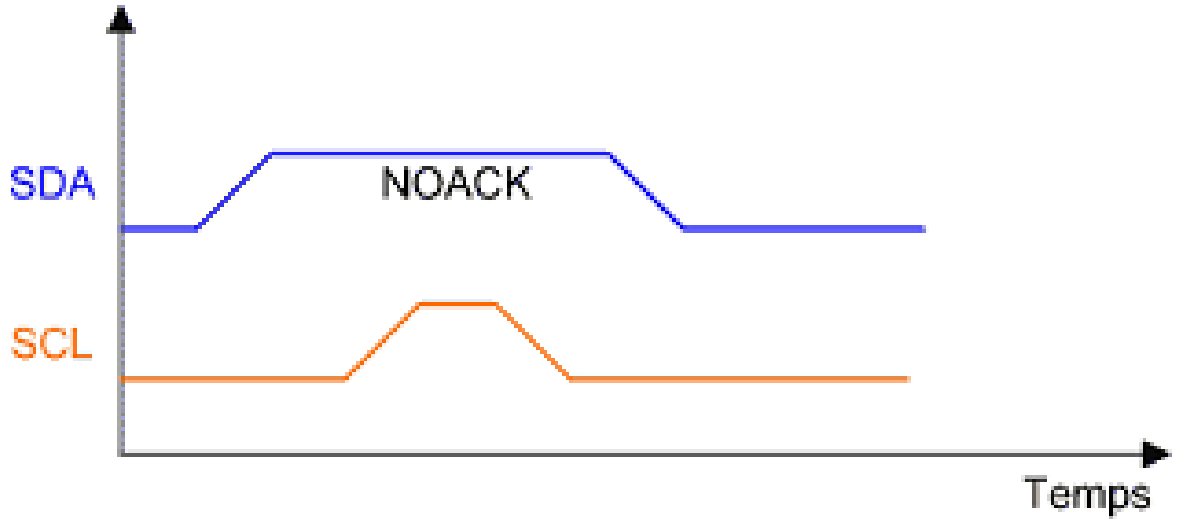



### d. L'acknowledge.

Voici de nouveau un nom barbare. L'acknowledge (ACK) est en fait l'accusé de réception de l'octet envoyé. C'est donc le récepteur qui émet ce bit pour signaler qu'il a bien lu l'octet envoyé par l'émetteur. Cet accusé de réception est lu comme un bit classique. Il vaudra 0 si l'accusé de réception signifie « OK », et 1 pour toute autre raison (récepteur dans l'impossibilité de répondre, par exemple). Voici un ACK (OK) :



Et voici une absence d'accusé de réception, encore dénommée NOACK. En effet, un NOACK équivaut à une absence de réaction, puisque seul le niveau bas est imposé, le niveau haut étant lié à la libération de la ligne (ou à sa non appropriation).




### e. Le bit read/write.

Il nous faut encore voir un bit particulier. Le bit R/W indique à l'esclave si les bits de données contenus dans la trame sont destinés à être écris (R/W = 0) ou lus (R/W = 1) par le maître. Dans le cas d'une écriture, le maître enverra les données à l'esclave, dans le cas d'une lecture, c'est l'esclave qui enverra ses données au maître.

## V.9.3. La notion d'adresse.

Nous avons vu que nous pouvions connecter un grand nombre de composants I²C sur le même bus. C'est d'ailleurs le but recherché. Mais nous ne disposons d'aucune ligne de sélection, comme nous l'avions pour le module SPI (ligne *SS*). C'est donc de façon logicielle que le destinataire va être sélectionné. Ceci fait naturellement appel à la notion d'adresse.

La première norme I²C limitait la taille des adresses à 7 bits. Cependant, au fil de son évolution, on a vu apparaître la possibilité d'utiliser des adresses codées sur 10 bits. Nous verrons comment cela est possible. Nous pouvons donc dire que seul l'esclave dont l'adresse correspond à celle envoyée par le maître va répondre.

Toutes les adresses ne sont pas autorisées, certaines sont réservées pour des commandes spécifiques. Parmi celles-ci, nous trouvons :

- ✓ 0b0000000 : utilisée pour adresser simultanément tous les périphériques (general call address). Les octets complémentaires précisent le type d'action souhaité (par exemple, reset).
- ✓ 0b0000001 : utilisée pour accéder aux composants CBUS (ancêtre de l'I²C).
- ✓ 0b0000010 : réservée pour d'autres systèmes de bus.
- ✓ 0b0000011 : réservée pour des utilisations futures.
- ✓ 0b00001xx : pour les composants haute-vitesse.

- ✓ 0b11111xx : réservée pour des utilisations futures.
- ✓ 0b11110xy : permet de préciser une adresse sur 10 bits.

Concernant cette dernière adresse, cette séquence permet de compléter les 2 bits de poids forts « xy » reçus par un octet complémentaire contenant les 8 octets de poids faible. Nous obtenons donc une adresse comprenant 10 bits utiles. Nous verrons ceci en détails. Souvenez-vous que si vous devez attribuer une adresse à votre PIC® configuré en esclave, il vous faudra éviter les adresses précédentes, sous peine de problèmes lors de l'utilisation avec certains composants spécifiques.

Corollaires de tout ce qui précède :

- ✓ Seuls les esclaves disposent d'adresses.
- ✓ Ce sont toujours les maîtres qui pilotent le transfert.
- ✓ Un maître ne peut parler qu'à un esclave (ou à tous les esclaves), jamais à un autre maître. Rien n'interdit cependant qu'un composant passe du status de maître à celui d'esclave et réciproquement. Le bus I²C est donc d'une complète souplesse à ce niveau.

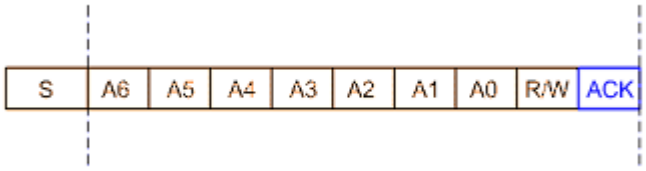
## V.9.4. Structure d'une trame I²C.

Nous avons vu tous les signaux possibles, il nous faut maintenant étudier le protocole de communication des intervenants en I²C. Nous savons déjà que la transmission commence par un « start-condition » (S). Vient ensuite l'adresse, codée sur 7 ou 10 bits, complétée par le bit R/W qui précise si les données qui vont suivre seront écrites ou lues par le maître.

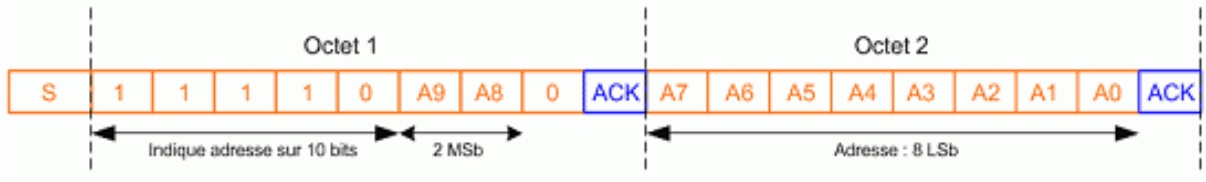
Notez que certains considèrent que ce bit fait partie de l'adresse, ce qui implique alors que l'adresse réelle se retrouve multipliée par deux, et que les adresses paires sont destinées à des écritures, et les adresses impaires à des lectures. Chaque octet envoyé est toujours accompagné d'un accusé de réception de la part de celui qui reçoit. Un octet nécessite donc 8 + 1 = 9 impulsions d'horloge sur la pin SCL. On distingue dès lors 2 cas : soit l'adresse est codée sur 7, soit sur 10 bits.

Voici comment se présente le début d'une trame (à partir de cet instant, dans les chronogrammes concernant l'I²C, je note en rouge ce qui est transmis par le maître et en bleu ce qui est transmis par l'esclave).

Notez donc, et c'est logique, que c'est toujours le maître qui envoie l'adresse, qu'il soit récepteur ou émetteur pour le reste du message.

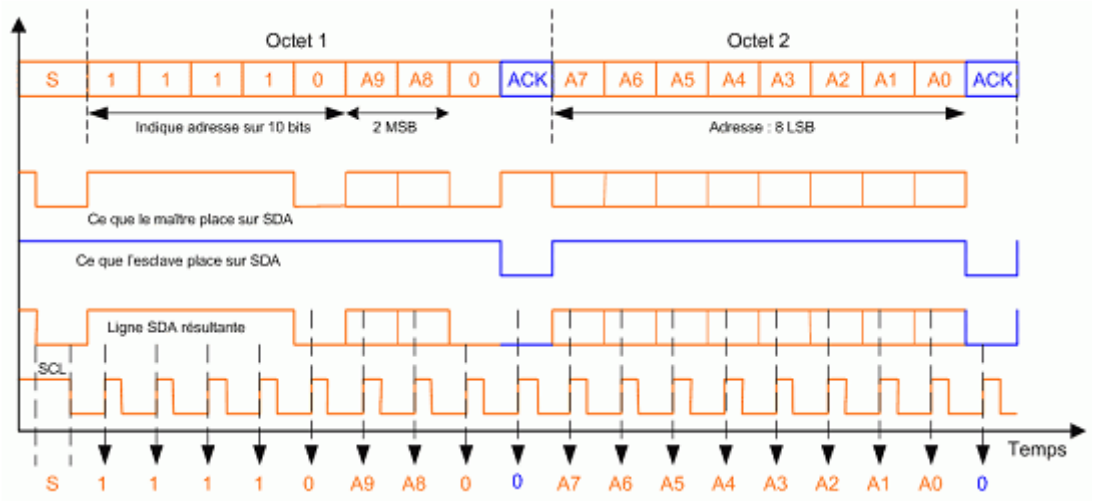


Pour coder une adresse sur 10 bits, on utilisera comme premier octet l'adresse réservée 0b11110xy0 qui précise qu'un second octet est nécessaire. Voici ce que donne l'envoi d'une adresse sur 10 bits.




Remarquez que si vous codez l'adresse sur 2 octets, le bit R/W doit toujours impérativement être égal à 0. Vous précisez donc toujours une écriture. Nous verrons plus loin ce que nous devons faire si nous avons besoin d'une lecture. De plus, le bit R/W ne se situe que dans le premier octet de l'adresse, cela explique pourquoi vous pouvez caser 8 bits dans l'octet 2, alors que vous êtes limités à 7 bits dans le cas d'une adresse sur 1 octet. Notez enfin que si vous utilisez une adresse sur 7 bits, elle ne pourra jamais, évidemment, commencer par 0b11110.

En conclusion voici le chronogramme correspondant (j'ai considéré que les flancs des signaux étaient verticaux, c'est plus simple à dessiner à cette échelle) :




Vous constaterez en lisant les chronogrammes présents dans les documentations techniques des composants, qu'on ne vous donne que la ligne SDA résultante. Celle-ci définit le niveau présent sur la ligne, mais il ne faut jamais oublier que le maître et l'esclave placent des bits à tour de rôle sur la même ligne.

Vous voyez ici, par exemple, que durant l'acknowledge, le maître libère la ligne SDA (niveau 1). A ce moment, c'est l'esclave qui impose le niveau bas de confirmation. Vous en déduirez à ce moment que lorsque le maître écrit (en rouge), l'esclave lit, lorsque l'esclave écrit (en bleu), le maître lit. Nous venons de créer le début de notre trame, à savoir le start-condition (S) envoyé par le maître, le premier octet d'adresse également envoyé par le maître, suivi par l'accusé de réception de l'esclave (ACK) envoyé par l'esclave. Suit éventuellement un second octet d'adresse (adresse sur 10 bits), complété de nouveau par l'accusé de réception.

Remarquez que, si vous vous êtes posés la question, après l'envoi du premier octet, dans le cas d'une adresse codée sur 10 bits, il pourrait très bien y avoir plusieurs esclaves concernés par ce premier octet (plusieurs esclaves dont l'adresse tient sur 10 bits). Il se peut donc qu'il y ait plusieurs esclaves différents qui envoient en même temps leur accusé de réception. Ceci n'a aucune importance, lors de l'envoi du second octet, il n'y aura plus qu'un seul esclave concerné. Si on ne reçoit pas un « ACK », c'est soit que l'esclave sélectionné n'existe pas, soit qu'il n'est pas prêt.

A ce stade, nous avons choisi notre esclave, reste à savoir ce qu'on attend de lui. Nous avons à ce stade, 2 possibilités :

1. Soit nous allons lui envoyer un ou plusieurs octets de donnée.
2. Soit nous allons recevoir de lui un ou plusieurs octets de donnée.

La sélection de l'un ou l'autre cas dépend de la valeur que vous avez attribué à votre bit R/W.

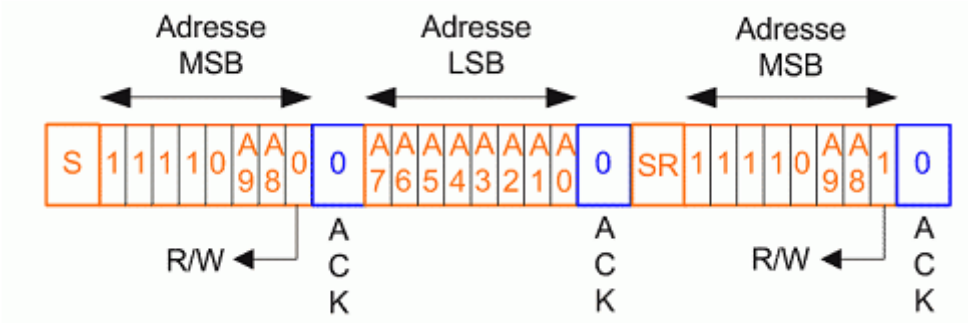
- ✓ Si R/W = 0, les octets suivants sont envoyés par le maître et lus par l'esclave (écriture).
- ✓ Si R/W = 1, les octets suivants sont envoyés par l'esclave et lus par le maître (lecture).

Corollaire : il n'est pas possible, en I²C, d'écrire et de lire en même temps. Si vous désirez effectuer une lecture et une écriture, ceci nécessitera 2 opérations, donc 2 envois du start sequence.

Il faut comprendre que l'imposition du bit R/W à « 0 » se justifie pour l'électronique des circuits concernés. Les esclaves devront en effet lire le second octet d'adresse, il s'agit donc d'une écriture (vu du côté du maître). La solution pour la lecture dans les adresses à 10 bits est la suivante :

- ✓ Le maître envoie le start-condition.
- ✓ Le maître envoie le premier octet d'adresse, avec R/W à « 0 ».
- ✓ Le maître envoie le second octet d'adresse.
- ✓ Le maître envoie un repeated start-condition.
- ✓ Le maître envoie le premier octet d'adresse, avec R/W à « 1 ».
- ✓ Le maître commence la réception des données.

Vous remarquez que le maître doit repréciser le bit R/W à « 1 ». Comme ce bit est contenu dans le premier octet d'adresse, il doit donc réenvoyer ce dernier. Et comme il conserve le dialogue (il n'a pas envoyé de stop-condition), il s'adresse donc toujours en fait au même esclave, et n'a pas besoin de répéter le second octet d'adresse. Voici ce que donne l'envoi d'une adresse 10 bits pour une lecture :




Voyons tout d'abord le cas de l'écriture. Je parlerai pour faire simple d'adresses codées sur 7 bits, pour les adresses sur 10 bits, il suffira d'ajouter un ou deux octet, vous avez compris la manœuvre. La procédure est la suivante :

- ✓ Le maître envoie le start-condition (S).
- ✓ Le maître envoie l'octet d'adresse, avec le bit R/W à 0 (écriture).
- ✓ L'esclave répond par « ACK ».
- ✓ Le maître envoie le premier octet de donnée.
- ✓ L'esclave répond par « ACK ».
- ✓ …….
- ✓ Le maître envoie le dernier octet de donnée.
- ✓ L'esclave répond par « ACK » ou par « NOACK ».
- ✓ Le maître envoie le stop-condition (P).

Notez qu'il est possible, pour l'esclave d'envoyer un « NOACK » (c'est-à-dire un bit « ACK » à « 1 ») pour signaler qu'il désire que le maître arrête de lui envoyer des données (par exemple si sa mémoire est pleine, ou si les octets reçus sont incorrects etc.). Le maître dispose de la possibilité de mettre fin au transfert en envoyant le stop-sequence (P).

Voici ce que ça donne dans le cas où le maître envoie une adresse codée sur 7 bits et 2 octets de données à destination de l'esclave.


Le cas de la lecture n'est pas plus compliqué. Voici la séquence, toujours pour une adresse codée sur 7 bits :

- ✓ Le maître envoie le start-condition.
- ✓ Le maître envoie le premier octet d'adresse, avec le bit R/W à 1 (lecture).
- ✓ L'esclave répond par « ACK ».
- ✓ L'esclave envoie son premier octet de donnée.
- ✓ Le maître répond par « ACK ».
- ✓ …….
- ✓ L'esclave envoie le $n^{ème}$ octet de donnée.
- ✓ Le maître répond « NOACK » pour signaler la fin du transfert.

- ✓ Le maître envoie le stop-sequence (P).

Notez que c'est le maître seul qui décide de mettre fin à la réception. Il doit alors commencer par envoyer un « NOACK », ce qui signale à l'esclave qu'il n'a plus besoin de transmettre, suivi par un stop-sequence qui met fin à la transmission.

Comme c'est le maître qui envoie le ACK à chaque fin de lecture, l'esclave n'a aucun moyen de faire savoir qu'il désire que la lecture s'arrête. Voici ce que cela donne :


Vous voyez que de nouveau, tout est logique. Le début de la séquence est toujours identique, le maître envoie toujours le start, l'adresse, et le bit R/W. L'esclave donne toujours le premier acknowledge. A partir de l'octet qui suit l'envoi de l'adresse, c'est l'esclave qui place les données sur SDA, et donc le maître qui répond par l'acknowledge. La dernière requête se termine par l'envoi, par le maître, d'un "NOACK" précédent le stop-condition. Notez cependant que c'est toujours le maître qui pilote la ligne SCL.

## V.9.5. Evénements spéciaux.

Il nous reste à voir quelques séquences particulières qui complètent cette gestion du bus I²C.

### a. La libération du bus.

Nous avons vu que le maître prend le contrôle du bus avec le start-condition. A partir de ce moment, les autres maîtres éventuels doivent attendre que le maître ait terminé ses transactions avant de tenter de prendre à leur tour le contrôle du bus. La libération du bus est effective une fois que 4,7 µs se sont écoulées depuis l'envoi du stop-sequence, aucun nouveau start-sequence n'ayant été reçu.

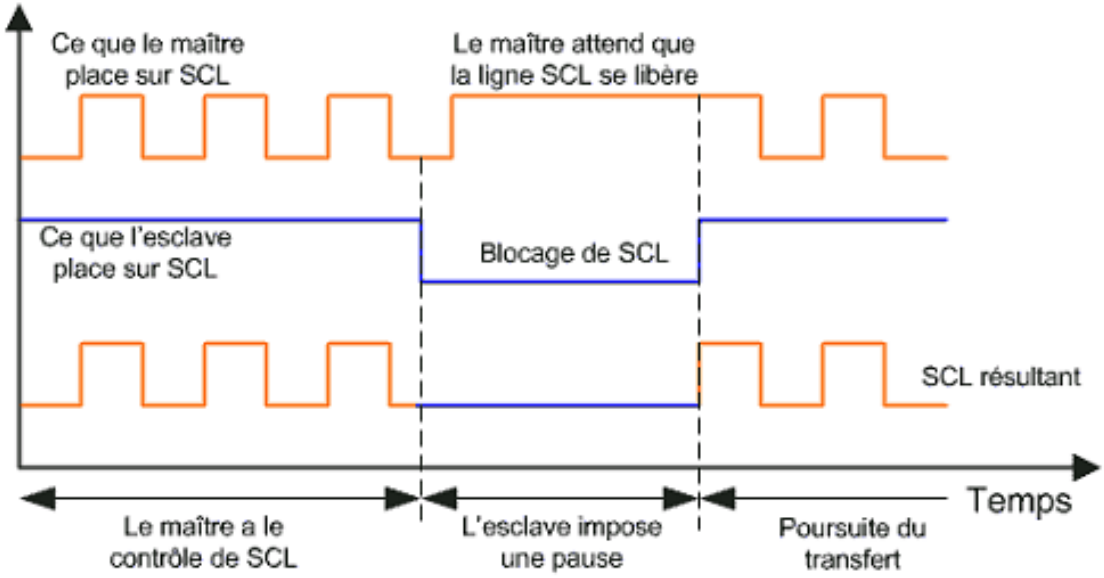
### b. Le repeated start-condition.

Si le maître désire envoyer une seconde trame, au lieu de terminer la trame en cours par un stop-condition, il la terminera par un nouveau start-condition. Ce dernier porte dans ce cas le nom de SR pour « repeated start-condition ». Sa fonction est strictement identique au start condition, mais ne libère pas le bus comme risquerait de le faire la combinaison stop-sequence/start-sequence. Le maître actuel empêche donc un autre maître de s'accaparer le bus. Le repeated start-condition marque en même temps la fin d'une transaction, et le début d'une suivante.

### c. La pause.

Un autre cas particulier est celui ou, lors d'une écriture, l'esclave demande au maître de patienter durant un certain temps (par exemple le temps mis pour traiter l'octet précédent). La méthode est simple, l'esclave bloque la ligne d'horloge SCL à l'état bas durant la pause demandée. Lorsque le maître essaye d'envoyer l'impulsion d'horloge, il n'y arrive pas, puisqu'elle est forcée à l'état bas par l'esclave.

Le maître dispose d'une électronique interne qui lui permet de vérifier l'état de SCL. Lorsqu'il est de nouveau prêt, l'esclave libère la ligne SCL, qui prend alors l'état haut. Le maître peut alors poursuivre. Tant que la ligne SCL est bloquée, le maître resette son générateur de temps interne. Autrement dit, à la libération de la ligne SCL, les chronogrammes de génération d'horloge seront respectés. Vous voyez que l'impulsion de SCL qui suit la fin de la pause a strictement la même durée que toutes les autres impulsions.




### d. Arbitrage du bus.

Vous trouverez partout, dans les datasheets concernant les composants I²C, la notion d'« arbitration », pour arbitrage. C'est dans cette capacité qu'ont ces composants de gérer les conflits que réside toute la puissance du bus I²C. Ces méthodes permettent de disposer d'un bus comportant plusieurs maîtres, et permettent des systèmes totalement décentralisés, dans lesquels chacun peut librement prendre la parole.

Il faut pour comprendre ce qui suit, avoir bien compris que le niveau « 0 » est prioritaire sur le niveau « 1 ». Si un circuit quelconque passe une ligne à 0, cette ligne sera forcée à 0. Si par contre le circuit demande que la ligne passe à « 1 » (libération de la ligne), celle-ci ne passera à 1 que si aucun autre circuit ne l'a forcée à « 0 ». Bien entendu, les circuits ne peuvent pas prendre la parole n'importe quand ni n'importe comment. Il importe, comme lors d'une discussion entre humains, de respecter les règles de courtoisie élémentaires (quoi que certains, au vu de certains forums, semblent s'en passer sans problème).

La première question qui se pose est : « quand un maître peut-il prendre la parole » ? La réponse est simple : quand le bus est libre. Il nous faut alors établir quelles sont les conditions dans lesquelles le bus peut être considéré comme tel.

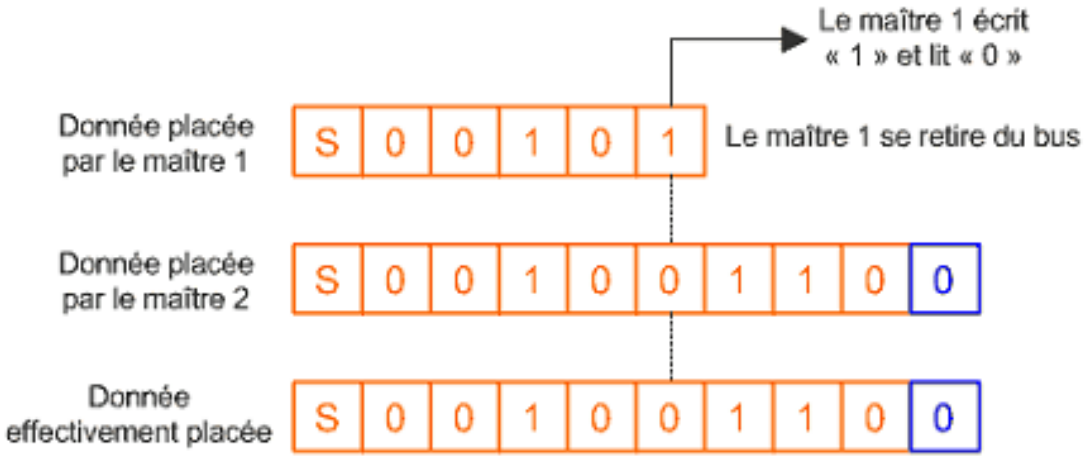
Tout maître est capable de détecter automatiquement les « start » et « stop-conditions ». Dès lors, dès qu'un « start-condition » est détecté, le bus est considéré occupé jusqu'à la détection du « stop-condition » correspondant. A partir de ce moment, si le maître précédent ne reprend pas la parole avant 4,7 µs, le bus est considéré comme libre. Si le nouveau maître vient de se

connecter au réseau, il lui suffit de s'assurer que la ligne SCL soit à l'état haut depuis au moins 4,7µs pour vérifier qu'un autre maître ne dirige pas le bus.

Dans la pratique, certains maîtres n'attendent même pas cette limite de 4,7µs pour s'assurer que le bus soit libre, ça ne pose pas de problème de fonctionnement particulier.

Imaginons que le bus soit libre, et que les maîtres 1 et 2 s'accaparent le bus en même temps. Chaque maître dispose d'une électronique qui vérifie en permanence la correspondance entre les niveaux appliqués sur SDA et SCL, et les niveaux lus en réalité sur ces lignes. S'il y a discordance, c'est qu'un tiers prend le contrôle de cette ligne. Nous en avons déjà vu un exemple lorsque l'esclave imposait une pause via la ligne SCL. Nous savons donc que les 2 maîtres vont placer leurs données sur la ligne SDA et que l'esclave lira la donnée placée sur le flanc montant de SCL. Mais les maîtres également vont lire automatiquement la ligne SDA.

Si un maître place un niveau 0, le niveau lu sera obligatoirement 0. Si, par contre, le maître place un niveau 1, et qu'un autre maître a placé la valeur 0, le premier va constater qu'il y a un autre maître, puisque la ligne SDA vaudra « 0 » alors qu'il a placé « 1 ». Le second maître ne va s'apercevoir de rien du tout, puisque la ligne SDA est bien à 0. Dans ce cas, le maître qui n'a pas pu imposer son niveau « 1 » perd le contrôle du bus, et se retire au profit du maître « prioritaire ». Voici ce que ça donne :




Dans cet exemple, le maître 2 désire placer l'adresse 0b0010011, soit D'19'. Le maître 2 tente de placer l'adresse B'0010100', soit D'20'. Vous voyez que le premier maître qui envoie un bit à « 1 » en même temps que le maître 2 place un « 0 » est mis hors-jeu. On peut déjà en déduire les règles suivantes :

- ✓ Les adresses les plus faibles sont prioritaires par rapport aux adresses plus élevées. Dans la construction d'un système, les esclaves les plus prioritaires se verront donc affecter les adresses les plus basses.
- ✓ En cas d'adresses identiques, l'écriture a priorité sur la lecture (car le bit R/W vaut 0 pour l'écriture).
- ✓ Les « collisions » sont dites « non destructives », puisque la donnée prioritaire arrive intacte au destinataire.

**Corollaire :** ce ne sont pas les maîtres qui sont prioritaires les uns par rapport aux autres, les priorités sont fonctions des trames envoyées.

Donc, si, au même moment, un maître tente de sélectionner l'esclave xx, et qu'un autre maître tente de sélectionner l'esclave yy, celui qui envoie l'adresse la plus élevée devra se retirer du bus.

Dans le cas où les 2 maîtres accèdent au même esclave, nous avons 2 possibilités :

1. Tout d'abord, les 2 maîtres effectuent une lecture. Dans ce cas, il n'y a absolument aucun problème. Les 2 maîtres envoient tous les deux une requête de lecture de l'esclave, qui va leur répondre à tous les deux en même temps. La requête étant la même, la réponse est donc identique et valide.
2. Si, par contre, les 2 maîtres effectuent une écriture, l'adresse va être complétée par les octets de donnée. Dans ce cas, si les 2 maîtres écrivent la même chose, tous les intervenants n'y verront que du feu. L'esclave écrira la donnée, valide puisqu'elle est identique pour les 2 maîtres.

Si les 2 maîtres n'écrivent pas la même chose, nous allons nous retrouver avec un conflit du même type que celui concernant l'adresse. A un moment donné, un des maîtres va tenter d'écrire un « 1 », et lira « 0 » sur la ligne SDA. Il se retirera donc de la communication. La priorité sera une fois de plus affectée à l'octet de donnée de valeur la plus faible.

Vous voyez qu'à partir de ce moment, tous les problèmes de conflits de bus sont résolus. Bien entendu, tout ceci est automatique dans les circuits intégrés. Le programmeur reçoit donc simplement des indicateurs qui l'informent de l'état actuel de la transmission.

Et si 2 esclaves différents tentent d'écrire en même temps ? Et bien, c'est tout simplement impossible, chaque esclave doit, par définition, avoir une adresse différente, et ne peut répondre que s'il est sélectionné. Si vous avez besoin de rendre un maître prioritaire, il suffit que le délai qui sépare la détection du stop-condition de la prise de contrôle du bus soit plus court que pour les autres maîtres.

Autrement dit, vous introduirez un délai plus ou moins long entre la détection du stop-condition et le fait de considérer que le bus est libre. Ainsi, le maître qui disposera du délai le plus court pourra s'imposer en premier. De cette façon, vous pouvez choisir entre donner priorité à un maître donné, ou priorité à l'adresse de destination la plus petite.

Dernière petite question : un maître peut-il parler à un autre maître ? En fait, non.

D'une part, seuls les esclaves ont une adresse, et seuls les esclaves subissent l'horloge. Par contre, rien n'interdit à un maître, lorsqu'il n'a rien à dire, de se configurer en esclave. Alors il peut échanger des informations avec un maître.

# V.10. Le module MSSP en mode I²C.

## V.10.1. Introduction

Nous avons vu que le bus I²C permettait plusieurs configurations différentes, nécessitait la gestion de l'arbitrage du bus, des différents événements etc.

Notre 16F87x est capable de gérer tout ceci de façon automatique. Nous allons donc trouver :

- ✓ La gestion des modes esclave, maître, et multi-maître.
- ✓ La génération et la détection de start et stop-condition.
- ✓ La génération et la reconnaissance des signaux « ACK ».
- ✓ La génération automatique des « NOACK » en cas de problème logiciel.
- ✓ Les modes d'adressage sur 7 et 10 bits.
- ✓ L'utilisation de toutes les vitesses de transfert compatibles.

De plus, les pins SDA et SCL, lorsque le module est utilisé en mode I²C, adaptent les flancs des signaux en suivant les spécifications I²C. Ceci est rendu possible par l'intégration de filtres spécifiques sur les pins concernées.

En effet, comme le signal SDA, en émission, doit rester présent un certain temps après le retour à 0 de la ligne SCL, ce temps étant dépendant de la vitesse du bus I²C, il faudra préciser ce paramètre. Les 16F87x sont prévus pour travailler avec des fréquences d'horloge SCL de 100KHz, 400KHz, et 1MHz. Vous configurerez les filtres en fonction des fréquences choisies.

Dans le même esprit, les pins disposent d'entrées de type « trigger de Schmitt », qui permettent une détection plus sécurisée des signaux parasités. Les pins SDA et SCL sont naturellement multiplexées avec d'autres pins, en l'occurrence RC4 et RC3. Nous utiliserons dans ce chapitre les registres étudiés dans le mode SPI, mais le mode I²C nécessite l'utilisation de registres supplémentaires.

En tout, nous aurons besoin de 6 registres. Sachant que le registre SSPSR contient la donnée en cours de transfert et que SSPBUF contient la donnée reçue ou à émettre (voir mode SPI), il nous reste 4 registres à examiner.

## V.10.2. Le registre SSPSTAT.

Tout comme pour l'étude du mode SPI, je décris les fonctions des bits des registres étudiés uniquement pour le mode I²C. Certains bits ont une autre signification suivant le mode, pour plus de détail, consultez le datasheet. Il faut noter que les noms de plusieurs bits ont été choisis en fonction de leur rôle dans le mode SPI.

Comme le mode I²C peut utiliser ces bits dans un but complètement différent, leur nom n'a parfois plus aucun rapport avec leur fonction. Dans ce cas, je ne rappellerai pas ce nom, j'utiliserai un nom plus approprié.

Un exemple est donné par le bit SMP (SaMple bit). Ce bit permettait de choisir l'instant d'échantillonnage (sample) dans le mode SPI. Dans le mode I²C, il sert à mettre en service le filtre adéquat. Son nom dans ce mode n'a donc plus aucun rapport avec sa fonction.

**Le registre SSPSTAT en mode I²C.**

1. b7 : SMP : Slew rate control set bit (1 pour 100 KHz et 1MHz, 0 pour 400 KHz).
2. b6 : CKE : Input levels set bit (0 = bus I²C, 1 = bus SMBUS).
3. b5 : D_A : Data / Address bit (0 = adresse, 1 = data).
4. b4 : P : stoP bit.
5. b3 : S : Start bit.
6. b2 : R_W : Read/Write bit information (0 = write, 1 = read).
7. b1 : UA : Update adress.
8. b0 : BF : Buffer Full / data transmit in progress.

Vous constatez que, dans ce mode, tous les bits sont maintenant utilisés. Nous avons beaucoup plus d'indicateurs que dans le mode SPI. Voyons tout ceci en détail.

- ✓ Le bit SMP prend ici une nouvelle signification. Il détermine si le filtre doit être ou non mis en service. Il vous suffit de savoir que si vous travaillez avec une fréquence de 100 KHz ou de 1 MHz, vous devez mettre ce bit à « 1 » (hors-service). Par contre, pour une vitesse de 400 KHz, vous le laisserez à « 0 » (en service).
- ✓ Le bit CKE prend également une signification différente. Si vous mettez ce bit à « 1 », vous travaillerez avec des niveaux d'entrée compatibles avec le bus SMBUS. Par contre, en le laissant à « 0 », vous travaillerez avec des signaux compatibles I²C (ce qui est notre objectif ici).
- ✓ D_A vous indique si le dernier octet reçu ou transmis était un octet de donnée (1) ou un octet d'adresse (0). Ceci vous permet de savoir où vous en êtes dans le transfert d'informations.
- ✓ Le bit P est également un indicateur. Il vous informe si le dernier événement détecté était un stop-condition (P = 1). Il s'efface automatiquement dès qu'un autre événement est reçu. Il s'efface également lors d'un reset, ou si vous mettez le module MSSP hors-service.
- ✓ Le bit S est un indicateur qui procède exactement de la même façon, mais pour un startcondition.
- ✓ R_W a 2 fonctions distinctes selon qu'on travaille en I²C maître ou en I²C esclave :
  - ➢ En I²C esclave, il vous indique l'état du bit R/W (bit 0 du premier octet qui suit le startcondition). Si R/W vaut « 0 », une écriture est donc en cours, s'il vaut « 1 », il s'agit d'une lecture. Ce bit est valide entre la détection de la correspondance d'adresse (si c'est votre PIC® qui est sélectionnée par le maître) et la fin de la trame en cours, donc jusqu'au prochain stop-sequence, repeated start-sequence, ou bit NOACK.
  - ➢ En I²C maître, il vous informe si un transfert est en cours (1) ou non (0).
- ✓ Le bit UA est un indicateur utilisé uniquement en esclave sur adresses 10 bits. Il vous prévient quand vous devez mettre votre adresse à jour en mode I²C. En effet, vous ne

disposez que d'un seul registre de 8 bits pour inscrire votre adresse esclave. Comme le mode 10 bits nécessite 2 octets, le positionnement automatique de ce bit UA vous signale que le premier octet d'adresse a été traité, et qu'il est temps pour vous de placer votre second octet d'adresse dans le registre concerné (et inversément). Le bit UA force la ligne SCL à 0 (pause), et est effacé automatiquement par une écriture dans SSPADD.

- ✓ Reste maintenant le bit BF, qui indique si le registre SSPBUF est plein (1) ou vide (0). Evidemment, positionné, en réception, il signifie que l'octet a été reçu, alors qu'en émission il signale que la transmission de l'octet n'est pas encore terminée.

## V.10.3. Le registre SSPCON.

**SSPCON en mode I²C.**

1. b7 : WCOL : Write COLlision detect.
2. bit b6 : SSPOV : SSP OVerflow indicator.
3. bit b5 : SSPEN : SSP ENable select bit.
4. b4 : CKP : Pause (si 0).
5. b3 : SSPM3 : SSP select bit M3.
6. b2 : SSPM2 : SSP select bit M2.
7. b1 : SSPM1 : SSP select bit M1.
8. b0 : SSPM0 : SSP select bit M0.

Voyons ces bits en détail.

- WCOL est, comme son nom l'indique, un indicateur qui signale que nous rencontrons une collision d'écriture (WCOL = 1). Il y a 2 cas possibles :
  - En mode master, cette collision arrive si vous tentez d'écrire dans le registre SSPBUF alors que ce n'est pas le bon moment (bus pas libre, start-condition pas encore envoyée…). Cette collision a pour conséquence que la donnée n'est pas écrite dans le registre SSPBUF (bloqué en écriture).
  - En mode slave, ce bit sera positionné si vous tentez d'écrire dans le registre SSPBUF alors que le mot précédent est toujours en cours de transmission. Vous devrez remettre ce flag à « 0 » par logiciel.
- L'indicateur SSPOV vous informe d'une erreur de type « overflow ». C'est-à-dire que vous venez de recevoir un octet, alors que vous n'avez pas encore lu le registre SSPBUF qui contient toujours l'octet précédemment reçu. Le registre SSPBUF ne sera pas écrasé par la nouvelle donnée, qui sera donc perdue. Vous devez remettre ce flag à « 0 » par logiciel. Ce bit n'a donc de signification qu'en réception.
- SSPEN permet de mettre tout simplement le module en service. Les pins SCL et SDA passent sous le contrôle du module MSSP. Les pins SCL et SDA devront cependant être configurées en entrée via TRISC.
- CKP joue dans ce mode I²C un rôle particulier. Il permet de mettre l'horloge SCL en service (1), ou de la bloquer à l'état bas, afin de générer une pause (0). De fait, CKP ne sera utilisé que dans le cas de l'I²C esclave, qui est le seul mode où la pause peut s'avérer nécessaire.

- SSPMx sont les bits qui déterminent de quelle façon va fonctionner le module I²C. Je vous donne donc seulement les modes relatifs à l'I²C.

  **b3 b2 b1 b0 Fonctionnement.**

  0 1 1 0 mode I²C esclave avec adresse sur 7 bits.
  0 1 1 1 mode I²C esclave avec adresse sur 10 bits.
  1 0 0 0 mode I²C maître : fréquence déterminée par le registre SSPADD.
  1 0 0 1 réservé.
  1 0 1 0 réservé.
  1 0 1 1 Mode esclave forcé à l'état de repos.
  1 1 0 0 réservé.
  1 1 0 1 réservé.
  1 1 1 0 Mode esclave avec adresses 7 bits, interruptions sur événements S et P.
  1 1 1 1 Mode esclave avec adresses 10 bits, interruptions sur événements S et P.

  Nous ne parlerons que des 3 premiers modes, les autres n'étant pas nécessaires.

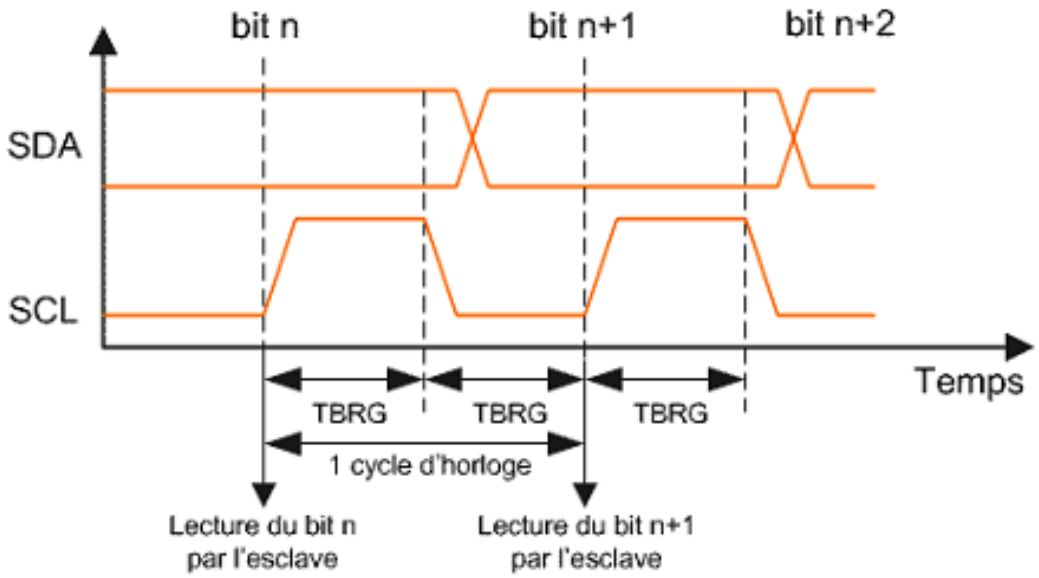
## V.10.4. Le registre SSPADD.

Ce nom signifie Synchronous Serial Port ADDress register. Il a deux fonctions complètement distinctes, selon que l'on travaille en I²C maître ou en I²C esclave.

Commençons par le mode esclave. C'est dans ce registre que vous placerez l'adresse à laquelle devra répondre votre PIC®. Si vous décidez de travailler en mode 10 bits, vous placerez les bits de poids fort dans SSPADD. Vous serez alors prévenus par le bit UA du registre SSPSTAT qu'il est temps d'y placer les bits de poids faible afin de compléter l'adresse, et inversément.

En mode maître, ce registre permet de déterminer la fréquence de travail du signal d'horloge. Tous les signaux sont synchronisés sur base d'une durée unitaire qui dépend de la valeur placée dans ce registre. Le PIC® utilise un registre spécial, équivalant à un timer, et qu'on dénomme BRG pour Baud Rate Generator. Ce dernier se charge avec la valeur contenue dans SSPADD (en fait les seuls 7 bits de poids faible), et décrémente 2 fois par cycle d'instruction. Une fois arrivé à 0, il se recharge si nécessaire pour produire un nouvel événement.

Par exemple, pour la lecture d'un octet, on aura un temps TBRG durant lequel la ligne SDA est positionnée, SCL étant à 0, ensuite la ligne SCL est mise à 1 durant un temps TBRG. A la fin de ce temps, la ligne SCL redescend et le nouveau bit SDA est positionné. Ce qui nous donne une lecture de bit de la forme :




Remarquez que le positionnement de SDA intervient peu après la descente de SCL. Ceci est impératif pour la norme I²C, qui impose que SDA ne peut varier tant que SCL se trouve à l'état haut. Ce retard est provoqué par les filtres de sortie (flew rate control), ce qui explique leur importance.

Nous constatons que la durée d'un cycle d'horloge SCL est de 2 fois TBRG. Comme BRG décrémente 2 fois par cycle d'instructions (donc tous les 2 Tosc), et qu'il met 2 Tosc supplémentaires pour se recharger, on peut dire que la durée d'un bit est de :

$$Durée\ d'unbit = Tosc \times 2 \times 2 \times (SSPADD + 1)$$

La fréquence de l'horloge est donc déterminée par la formule :

$$FSCL = \frac{Fosc}{4 \times (SSPADD + 1)}$$

En général, vous connaissez la fréquence (100KHz, 400KHz, 1MHz), et vous cherchez à déterminer SSPADD. D'où :

$$SSPADD = \frac{Fosc}{4 \times FSCL} - 1$$

Avec, bien entendu, Fosc la fréquence de votre oscillateur principal. Pour rappel, seuls 7 bits étant transférés vers BRG, SSPADD ne pourra être supérieur à 127.

Examinons maintenant à quoi sert SSPADD en mode esclave. En réalité, il prend ici le rôle que lui confère son nom, autrement dit il contient l'adresse à laquelle va répondre le PIC® aux requêtes du maître.

Dans le cas des adresses codées sur 7 bits, il contiendra les 7 bits en question. Mais attention, l'adresse en question sera écrite dans les bits 7 à 1 de SSPADD. Le bit 0 devra toujours être laissé à 0. En fait, tout se passe comme si le PIC® comparait le premier octet reçu avec l'octet mémorisé dans SSPADD. Si on a identité, il s'agit d'une écriture dans le PIC®.

Le PIC® compare ensuite l'octet reçu avec SSPADD+1, si on a identité, il s'agit d'une lecture du PIC®.

Autrement dit, si l'adresse que vous avez choisie pour votre PIC® est 0x03, vous devez placer cette adresse dans les bits 7 à 1, autrement dit décaler l'adresse vers la gauche, et positionner b0 à 0. Vous inscrirez donc dans SSPADD la valeur 0x06.

- Adresse : 0 0 0 0 0 1 1 : 0x03 (sur 7 bits).
- SSPADD : 0 0 0 0 0 1 1 0 : 0x06.

Si l'adresse est codée sur 10 bits, vous commencez par écrire le premier octet dans SSPADD (précédé bien entendu par B'11110'). Vous attendez le positionnement du bit UA, puis vous placez dans SSPADD votre second octet. En d'autres termes :

- Vous placez : « 1 1 1 1 0 A9 A8 0 » dans SSPADD.
- Vous attendez le bit UA.
- Vous placez : « A7 A6 A5 A4 A3 A2 A1 A0 » dans SSPADD.

Une fois la transaction terminée, le bit UA est de nouveau positionné, vous replacez B'1 1 1 1 0 A9 A8 0' dans SSPADD pour être prêt pour le transfert suivant. Tant que le bit UA reste positionné, le bus I²C est maintenu en pause automatiquement par votre PIC® esclave. Quand vous mettez SSPADD à jour, le bit UA est automatiquement effacé, et le bus est libéré.

## V.10.5. Le registre SSPCON2.

Voici un registre que nous n'avons pas encore examiné.

Voyons ce que ce registre contient. Le bit 7 ne concerne que le mode esclave, tandis que les autres bits ne concernent que le mode maître :

**SSPCON2.**

1. b7 : GCEN : General Call ENable bit (réponse appel général).
2. b6 : ACKSTAT : ACKnowledge STATus bit (état du bit ACK reçu).
3. b5 : ACKDT : ACKnowledge DaTa bit (valeur transmise).
4. b4 : ACKEN : ACKnowledge ENable bit (placer l'acknowledge).
5. b3 : RCEN : ReCeive ENable bit (lance la réception).
6. b2 : PEN : stoP-condition ENable bit (générer stop-condition).
7. b1 : RSEN : Repeated Start-condition ENable bit (générer SR).
8. b0 : SEN : Start-condition ENable bit (générer S).

Examinons maintenant ces bits en détail :

- ✓ GCEN est le seul bit qui concerne le mode esclave. Si vous validez ce bit, le PIC® répondra non seulement à l'adresse que vous avez placée dans SSPADD, mais également à l'adresse réservée d'appel général (0x00). Les octets qui suivront dans ce cas auront des fonctions particulières et réservées (reset par exemple). Si vous décidez que votre PIC® doit répondre à l'adresse d'appel général, il vous appartiendra de gérer ces commandes.

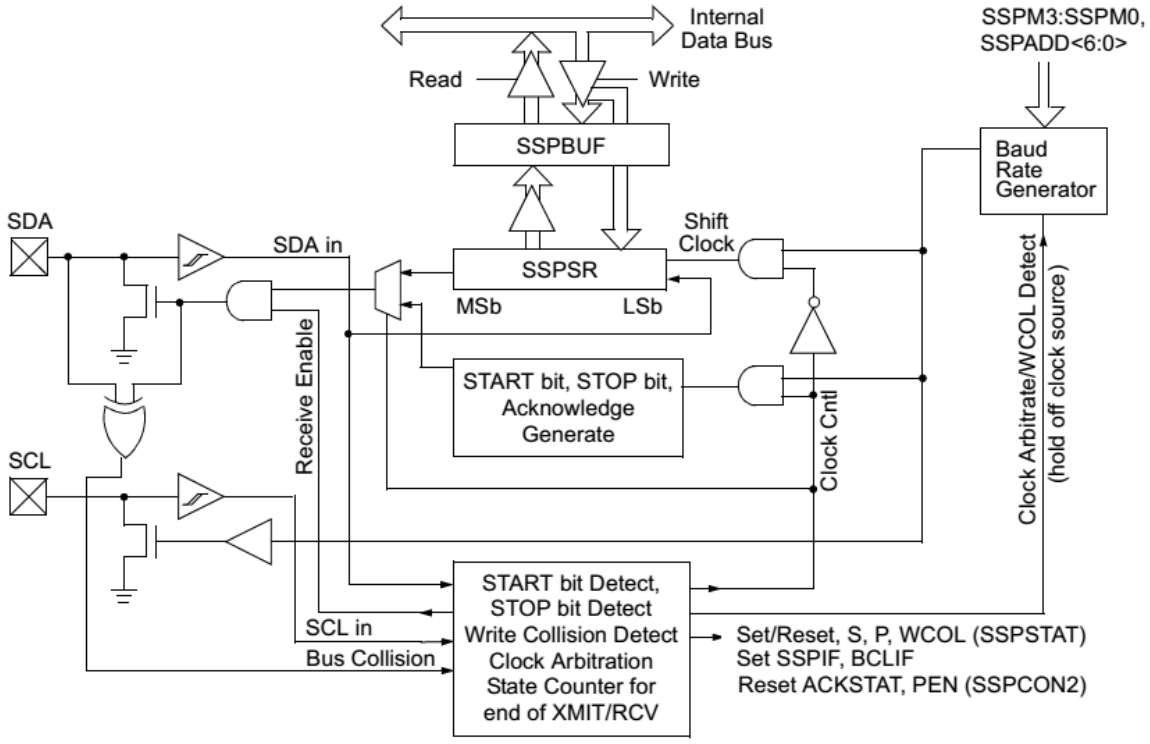
- ✓ ACKSTAT vous donne l'état du bit d'acknowledge envoyé par l'esclave lorsque vous écrivez des données. Si ACKSTAT vaut « 0 », c'est que l'esclave vous a envoyé un « ACK », s'il vaut « 1 », c'est qu'il n'y a pas eu d'acknowledge (NOACK).
- ✓ ACKDT est la valeur de l'acknowledge à envoyer à l'esclave lorsque vous procédez à une lecture. Ce bit sera envoyé lorsque vous positionnerez le bit ACKEN. De nouveau, une valeur « 0 » signifie que vous envoyez un « ACK », une valeur « 1 » sera placée pour un NOACK.
- ✓ ACKEN lance la génération de l'acknowledge. La valeur de ce bit (ACK ou NOACK) est déterminée par la valeur que vous avez placée dans ACKDT. Ces 2 bits sont donc liés.
- ✓ RCEN : Lance la réception d'un octet en provenance de l'esclave. Pour lancer une écriture, on place la valeur à écrire dans SSPBUF, pour lancer une lecture, on met RCEN à « 1 ».
- ✓ PEN : Lance la génération automatique du stop-condition.
- ✓ RSEN : Lance la génération du repeated start-condition.
- ✓ SEN : Lance la génération du start-condition. Attention de ne pas confondre, au niveau des noms, les bits CREN, RCEN, SREN. Une erreur à ce niveau est très difficile à détecter par relecture du code.

### V.10.6. Les collisions.

Il nous reste à voir ce qui se passe en cas de collision sur le bus, si le PIC® configuré en maître se voit ravir le bus I²C durant une opération (mode multi-maître). La réponse est simple : le bit BCIF (Bus Collision Interrupt Flag) du registre PIR2 est positionné, l'opération courante est abandonnée, et une interruption est éventuellement générée si elle est configurée.

## V.11. Le module MSSP en mode I²C maître.

Nous voici maintenant dans l'utilisation concrète de notre PIC® configurée en I²C maître. L'électronique du mode I²C en mode esclave est donnée ci-dessous.




Je ne traite pour l'instant que le cas où votre PIC® est le seul maître du bus (mode monomaître), c'est le cas que vous rencontrerez probablement le plus souvent. Je vais partir d'un cas théorique, dans lequel on réalise les opérations suivantes :

- ✓ On écrit un octet dans l'esclave.
- ✓ Puis, sans perdre le contrôle du bus, on lit un octet de réponse en provenance de l'esclave.

Ceci se traduira par la séquence d'événements suivante :

- ✓ On configure le module MSSP.
- ✓ On envoie le start-condition.
- ✓ On envoie l'adresse de l'esclave concerné avec b0 = 0 (écriture).
- ✓ On envoie la donnée à écrire.
- ✓ On envoie le repeated start-condition (SR).
- ✓ On envoie l'adresse de l'esclave concerné avec b0 = 1 (lecture).
- ✓ On lance la lecture.
- ✓ On envoie un NOACK pour indiquer la fin de la réception.
- ✓ On envoie un stop-condition.

Je vais maintenant vous expliquer comment tout ceci s'organise en pratique dans notre PIC®.

## V.11.1. La configuration du module.

Avant de pouvoir utiliser notre module, il importe de le configurer. Nous allons imaginer que nous travaillons avec une fréquence d'horloge principale de 20MHz (le quartz de notre PIC®), et avec une fréquence d'horloge I²C de 400 KHz. Les étapes seront les suivantes :

1. On configure TRISC pour avoir les pins SCL (RC3) et SDA (RC4) en entrée.
2. On configure SSPSTAT comme ceci :
    a. On positionne SMP (slew rate control) suivant la fréquence du bus I²C. Dans le cas de 400KHz, ce bit sera mis à 0, ce qui met le slew rate en service.
    b. On place CKE à 0 pour la compatibilité I²C.
3. On calcule la valeur de recharge du Baud Rate Generator, et on place la valeur obtenue dans SSPADD.

    SSPADD = (Fosc / (FSCL * 4)) – 1.

    Dans notre cas :

    SSPADD = (20 MHz / (400KHz * 4)) – 1 SSPADD = (20.000 / (400 * 4))-1 SSPADD = 11,5.

    On prendra la valeur 12 (décimal, faites attention). Pourquoi 12 et pas 11 ? Tout simplement parce que plus SSPADD est petit, plus la fréquence est grande. Il vaut donc mieux arrondir en utilisant une fréquence inférieure à la limite acceptable que supérieure.
4. On configure SSPCON, comme suit :
    a. On positionne le bit SSPEN pour mettre le module en service.
    b. On choisit SSMPx = 1000 pour le mode I²C maître.

## V.11.2. La vérification de la fin des opérations.

A ce stade, il faut que vous sachiez qu'il n'est pas autorisé, ni d'ailleurs possible, de générer une action si la précédente n'est pas terminée.

Par exemple, vous ne pouvez pas lancer un start-condition si le précédent stop-condition n'est pas terminé, de même vous ne pouvez pas envoyer un ACK si la réception de l'octet n'est pas achevée, et ainsi de suite.

Vous avez donc 3 solutions pour réaliser une séquence d'opérations sur le port I²C.

- ✓ Soit vous lancez votre commande et vous attendez qu'elle soit terminée avant de sortir de la sous-routine Ceci se traduit par la séquence suivante :
    - Commande.
    - Lancer la commande.
    - Commande terminée ?
    - Non, attendre la fin de la commande.
    - Oui, fin.
- ✓ Soit vous sortez de votre sous-routine sans attendre, mais vous devrez tester si l'opération précédente est terminée avant de pouvoir la lancer. Ceci se traduit par la séquence suivante :
    - Commande.
    - Y a-t-il toujours une commande en cours ?
    - oui, attendre qu'elle soit terminée.
    - non, alors lancer la nouvelle commande et fin.
- ✓ Soit vous utilisez les interruptions, mais je verrai ce cas séparément.

Dans le premier cas, on connaît l'événement dont on détecte la fin (la commande qu'on exécute), dans le second, on doit tester toutes les possibilités, afin de conserver une sous routine unique.

Il faut de plus savoir que lorsque vous lancez une commande (par exemple un acknowledge), le bit d'exécution est effacé automatiquement une fois l'action terminée. Donc, pour l'acknowledge, vous placez ACKEN pour lancer la commande, ACKEN est automatiquement effacé lorsque l'action est terminée.

Je vous présente tout d'abord la façon de détecter si une commande quelconque est en cours d'exécution. Il faut déterminer tous les cas possibles, les commandes peuvent être :

- ✓ Transmission en cours (signalée par le bit R/W du registre SSPSTAT).
- ✓ Start-condition en cours (signalé par le bit SEN du registre SSPCON2).
- ✓ Repeated start-condition en cours (signalé par le bit RSEN de SSPCON2).
- ✓ Stop-condition en cours (signalé par le bit PEN de SSPCON2).
- ✓ Réception en cours (signalé par le bit RCEN de SSPCON2).
- ✓ Acknowledge en cours (signalé par le bit ACKEN de SSPCON2).

### V.11.3. La génération du start-condition.

Commençons donc par le commencement, à savoir la génération du start-condition. Les étapes nécessaires sont les suivantes :

- ✓ On vérifie si l'opération précédente est terminée.
- ✓ On lance le start-condition.

Et son alternative, comme expliqué précédemment :

- ✓ On lance le start-condition.
- ✓ On attend que le start-condition soit terminé.

### V.11.4. L'envoi de l'adresse de l'esclave.

Nous allons maintenant devoir envoyer l'adresse de l'esclave. Nous allons construire notre sous-routine en imaginant que l'adresse de l'esclave est « SLAVE ». Cette adresse est codée sur 7 bits.

### V.11.5. Le test de l'ACK.

Nous aurons souvent besoin, après avoir envoyé l'adresse de l'esclave, de savoir si ce dernier est prêt, et donc a bien envoyé l'accusé de réception « ACK ». Avant de pouvoir tester ce bit, il faut que l'émission soit terminée, ce qui est automatiquement le cas si vous avez utilisé la seconde méthode, mais n'est pas vrai si vous avez préféré la première.

Voici donc le cas correspondant à la première méthode :

- ✓ On vérifie si l'opération précédente est terminée.
- ✓ On teste le bit ACK et on décide en conséquence.

### V.11.6. L'écriture d'un octet.

Nous en sommes arrivés à l'écriture de l'octet dans l'esclave. Cette procédure est strictement identique à l'envoi de l'adresse de l'esclave.

Nous allons donc réaliser les opérations suivantes :

- ✓ On vérifie si l'opération précédente est terminée.
- ✓ On place l'octet à envoyer dans SSPBUF, ce qui lance l'émission.

Et son alternative :

- ✓ On place l'octet à envoyer dans SSPBUF.
- ✓ On attend la fin de l'émission.

### V.11.7. L 'envoi du repeated start-condition.

Nous allons devoir procéder à une lecture. Comme nous étions en mode écriture, nous devons renvoyer un nouveau start-condition. Comme nous ne désirons pas perdre le bus, nous n'allons pas terminer l'opération précédente par un stop-condition, nous enverrons donc directement un second start-condition, autrement dit un repeated start-condition.

- ✓ On vérifie si l'opération précédente est terminée.
- ✓ On lance le repeated start-condition.

Et son alternative, toujours sur le même principe :

- ✓ On lance le repeated start-condition.
- ✓ On attend que le repeated start-condition soit terminé.

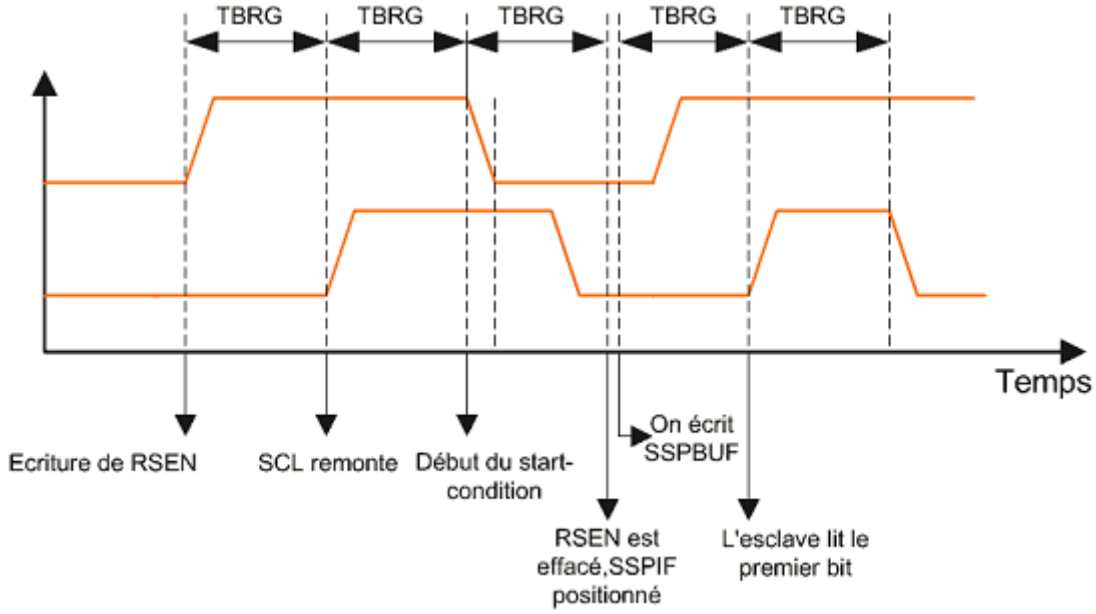
Rappelons qu'un start-condition part de la ligne au repos (SCL = SDA = 1) et place SDA à 0 alors que SCL reste à son état de repos (1). Le stop-condition, lui, remet les lignes dans leur état de repos en suivant la logique inverse (remontée de SDA alors que SCL est déjà remonté à 1).

Vous avez besoin du repeated start-condition lorsque les lignes ne sont pas dans leur état de repos, puisque SCL est à 0, et SDA peut l'être également (ACK). Donc, avant de pouvoir régénérer un nouveau start-condition, il faut remettre ces lignes dans leur état de repos. C'est ce que fait le repeated start-condition. La séquence générée par ce dernier est donc la suivante :

- ✓ Dès le positionnement de SREN, la ligne SDA est placée à l'état haut.
- ✓ Après un temps égal à TBRG, la ligne SCL est amenée également à l'état haut.
- ✓ Après une nouvelle durée égale à TBRG, on repasse SDA à 0, ce qui équivaut alors à un start-condition.
- ✓ TBRG plus tard, le bit SREN est effacé, et le bit SSPIF positionné.

Dès lors, il vous suffira de placer votre adresse dans SSPBUF. Vous voyez que les 2 premières lignes sont supplémentaires par rapport à un « startcondition » simple. Elles permettent de ramener les 2 lignes à l'état haut sans générer de « stop-condition ». Elles

nécessitent 2 temps TBRG supplémentaires. Les 2 dernières lignes correspondent évidemment au cycle normal d'un « startcondition ». Et hop, un petit chronogramme.




Si vous avez tout compris, vous constatez que rien ne vous empêche d'utiliser un « SR » au lieu d'un « S », mais vous ne pouvez pas utiliser un « S » en lieu et place d'un « SR ». Dans ce dernier cas, les lignes n'étant pas au repos, le « start-condition » ne serait tout simplement pas généré.

## V.11.8. L'envoi de l'adresse de l'esclave.

Nous allons maintenant devoir envoyer encore une fois l'adresse de l'esclave. Nous devrons cette fois positionner le bit 0 à 1 pour indiquer une lecture. Nous allons donc réaliser les opérations suivantes :

- ✓ On vérifie si l'opération précédente est terminée.
- ✓ On place l'adresse de l'esclave décalée vers la gauche et complétée par le bit 0 (R/W) à « 1 » (lecture) dans SSPBUF, ce qui lance l'émission.

Et son alternative :

- ✓ On place l'adresse de l'esclave comme précédemment.
- ✓ On attend la fin de l'émission.

## V.11.9. La lecture de l'octet.

Nous devons maintenant lire l'octet en provenance de l'esclave. De nouveau 2 méthodes. Tout d'abord :

- ✓ On vérifie si l'opération précédente est terminée.
- ✓ On lance la lecture.

Et l'alternative suivante :

- ✓ On lance la lecture.
- ✓ On attend la fin de l'opération pour sortir.

### V.11.10. L'envoi du NOACK.

Pour clôturer une lecture, le maître doit envoyer un NOACK, voici comment opérer :

- ✓ On vérifie si l'opération précédente est terminée.
- ✓ On lance le NOACK.

Et l'alternative suivante :

- ✓ On envoie NOACK.
- ✓ On attend que l'envoi soit terminé.

### V.11.11. L'envoi du stop-condition.

Il ne reste plus qu'à envoyer notre stop-condition pour voir si tout s'est bien passé.

- ✓ On vérifie si l'opération précédente est terminée.
- ✓ On lance le stop-condition.

Et son alternative, comme expliqué plus haut :

- ✓ On lance le stop-condition.
- ✓ On attend que le repeated start-condition soit terminé.

Nous avons maintenant terminé nos opérations. Remarquez cependant que si vous avez utilisé la seconde méthode, le transfert est terminé. Par contre, si vous avez préféré la première, il vous faudra encore attendre la fin du stop-condition avant que le bus ne soit effectivement libéré.

## V.12. L'utilisation des interruptions.

Il se peut que vous désiriez utiliser les interruptions pour traiter tous ces événements, en particulier si vous désirez que votre PIC® continue à effectuer d'autres tâches durant les temps des opérations sur le bus eeprom.

Il existe cependant une certaine difficulté pour mettre en œuvre les interruptions à ce niveau. En effet, vous avez un seul flag (SSPIF), alors que l'interruption sera générée pour chacune des opérations. Afin de savoir où vous en êtes au moment où vous arrivez dans votre routine d'interruption, il vous faudra utiliser un compteur que vous incrémenterez à chaque opération.

## V.13. Le module MSSP en mode I²C multi-maître.

Le passage en mode multi-maître signifie tout simplement que votre PIC® n'est pas le seul maître du bus I²C. Il doit donc s'assurer de ne pas entrer en conflit avec un autre maître durant les diverses utilisations du bus.

### V.13.1. L'arbitrage.

Il y a plusieurs choses à prendre en considération lorsque vous travaillez avec plusieurs maîtres.

Tout d'abord, vous ne pouvez prendre la parole que si le bus est libre, ce qui implique de vérifier si le bus est libre avant d'envoyer le start-condition.

En second lieu, lorsque vous prenez la parole, vous devez vous assurer qu'un autre maître qui aurait pris la parole en même temps que vous n'a pas pris la priorité. Auquel cas, vous devez vous retirer du bus.

Il y a donc 5 moments durant lesquels vous pouvez perdre le contrôle du bus :

5. Durant le start-condition.
6. Durant l'envoi de l'adresse de l'esclave.
7. Durant l'émission d'un NOACK.
8. Durant l'émission d'un octet.
9. Durant un repeated start-condition.

### V.13.2. La prise de contrôle du bus.

Une fois de plus, Microchip® nous a facilité la vie à ce niveau. Votre PIC® surveille le bus I²C en permanence, et vous informe de l'état du bus. En effet, chaque fois qu'il voit passer un « start-condition », il positionne le bit « S » du registre SSPSTAT à « 1 » et efface le bit « P ».

Chaque fois qu'un stop-condition est émis, il positionne le bit « P » du même registre, et efface le bit « S ».

Il vous suffira alors de tester si le bit « S » est effacé pour vous assurer que le bus est libre. Microchip® recommande cependant de tester les bits « S » et « P » simultanément pour s'assurer de la libération du bus. Quoi que je ne comprenne pas bien l'intérêt de cette technique, je vous la décrit cependant.

- ✓ Si le bit « P » est positionné, alors le bus est libre.
- ✓ Si « P » et « S » sont effacés, alors le bus est libre.

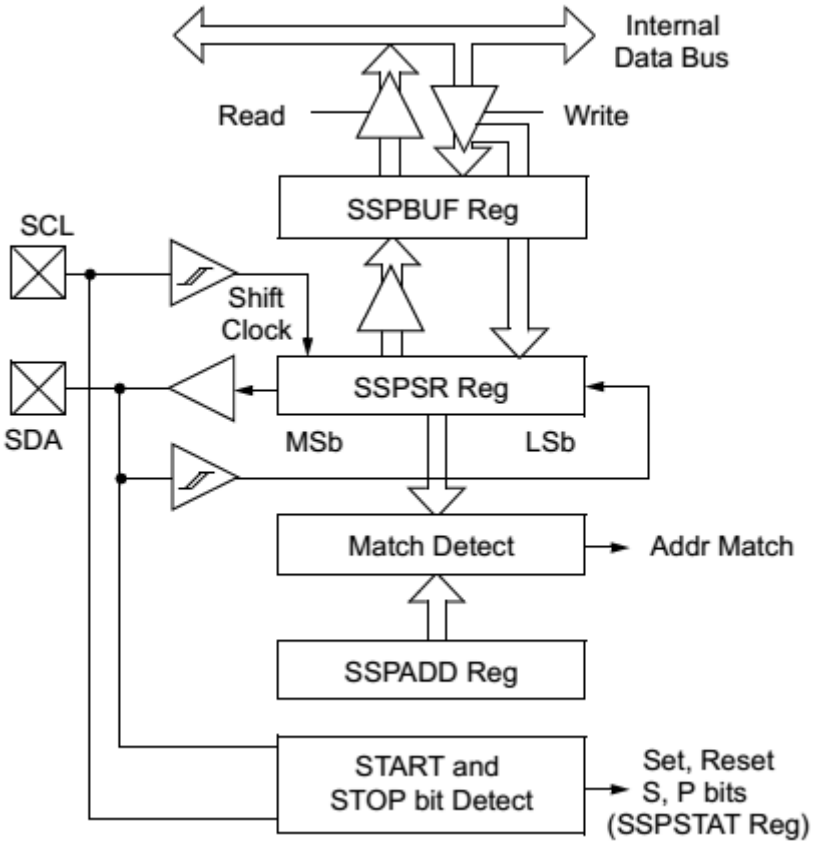
### V.13.3. La détection de la perte de contôle.

De nouveau, la détection de la perte de contrôle, quel que soit le cas, nécessite de ne tester qu'un seul bit. Le bit BCIF (Bus Collision Interrupt Flag) du registre PIR2. La détection de la perte de contrôle est en effet automatique et câblé de façon hardware dans le PIC®. C'est donc lui qui s'occupe de tout gérer à ce niveau. Il vous suffit donc, après ou avant chaque opération, suivant la méthode retenue, de tester l'état du bit BCIF. S'il est positionné, alors vous avez perdu le contrôle du bus et vous devez abandonner le traitement en cours. Vous le reprendrez dès que le bus sera à nouveau libre. Si vous utilisez les interruptions, vous pourrez également générer une interruption sur base de BCIF.

## V.14. Le module MSSP en mode I²C esclave 7 bits.

Je vais maintenant vous expliquer comment utiliser votre PIC® en I²C esclave. Vous rencontrerez ce cas lorsque vous voudrez que votre PIC® communique avec un autre maître, qui peut être un circuit programmable spécialisé, un microcontrôleur, un autre PIC® etc.

L'électronique du mode $I^2C$ en mode esclave est donnée ci-dessous.




La première chose à comprendre, c'est que, dans ce mode, vous pouvez émettre et recevoir, mais ce n'est pas vous qui décidez quand le transfert à lieu. Donc, puisque la transmission peut avoir lieu à n'importe quel moment, la meilleure méthode d'exploitation sera d'utiliser les interruptions.

Ainsi, en mode esclave, vous êtes déchargés de toute une série d'opération, comme la génération des start et stop-conditions, l'envoi de l'adresse etc. Vous disposez de 2 possibilités d'utilisation du mode esclave, selon que vous utiliserez une adresse codée sur 7 ou sur 10 bits.

Nous verrons les différences que cela implique. Pour l'instant, je me contente d'un adressage sur 7 bits. Ce mode correspond à une valeur de SSPMx de B'0110'.

### V.14.1. L'adressage et l'initialisation.

Nous allons cette fois utiliser notre registre SSPADD pour mémoriser l'adresse à laquelle répondra notre PIC®. En effet, en mode esclave, nul besoin de générer l'horloge, ce qui

explique le double rôle de ce registre. L'adresse sera mémorisée, bien entendu, dans les bits 7 à 1 du registre SSPADD. Le bit 0 devra être laissé à « 0 ».

Vous allez voir que bon nombre d'opérations sont maintenant prises automatiquement en charge par votre PIC®. Nous allons imaginer la séquence suivante pour traiter tous les cas particuliers :

- ✓ Le maître écrit un octet dans votre PIC®.
- ✓ Ensuite le maître lit un octet de réponse.

Voici, chronologiquement, tous les événements qui vont intervenir :

1. La réception du start-condition.
2. La réception de l'adresse en mode écriture.
3. La génération de l' « ACK ».
4. La réception d'un octet de donnée.
5. La réception de l'adresse en mode lecture.
6. L'émission d'un octet.

### V.14.2. la réception du start-condition.

Le maître va commencer par envoyer le start-condition. Votre PIC® esclave, qui surveille le bus, va positionner le bit « S » du registre SSPSTAT. Cependant, vous n'avez même pas à vous en préoccuper, le PIC® va poursuivre l'examen du bus.

### V.14.3. La réception de l'adresse en mode écriture.

Deux cas peuvent maintenant se produire.

1. Soit l'adresse qui vient d'être émise par le maître est celle de votre PIC®,
2. Soit elle concerne un autre esclave.

Dans le second cas, vous ne verrez rien du tout, le PIC® n'aura aucune réaction suite à une adresse qui ne le concerne pas. Il attend alors le prochain « stop-condition » pour poursuivre l'analyse du bus.

Par contre, si le message est bien destiné à votre PIC®, et si votre PIC® est en condition valable pour le recevoir, les événements suivants vont se produire :

- ✓ L'adresse sera transférée dans le registre SSPBUF.
- ✓ En conséquence, le bit BF sera positionné, pour indiquer que le buffer est plein.
- ✓ Le bit SSPIF sera positionné, et une interruption aura éventuellement lieu.
- ✓ Le bit D_A du registre SSPSTAT sera mis à 0 (adresse présente).
- ✓ Le bit R_W sera mis à 0 (écriture).
- ✓ L'accusé de réception « ACK » va être généré.

La première chose dont il faut se souvenir, c'est qu'il faut toujours lire une donnée présente dans SSPBUF, sinon la donnée suivante ne pourra pas être traitée. Ceci, même si vous n'avez pas besoin de cette donnée.

En effet, un octet présent dans SSPBUF est signalé par le positionnement à « 1 » du bit BF (Buffer Full). La lecture du registre SSPBUF provoque l'effacement automatique du bit BF. Si vous recevez votre adresse alors que l'octet BF était toujours positionné (vous n'aviez pas lu la donnée précédente), alors :

- ✓ L'adresse ne sera pas transférée dans SSPBUF.
- ✓ Le bit SSPOV sera positionné.
- ✓ L'accusé de réception ne sera pas transmis (NOACK) indiquant au maître que vous n'êtes pas prêt à recevoir.
- ✓ Le bit SSPIF sera positionné.

Le bit SSPOV devra impérativement être effacé par votre programme. Autrement dit, si vous avez des raisons de penser qu'il est possible que votre programme ne réagisse pas assez vite à un événement, vous devez toujours tester SSPOV. S'il vaut « 1 », alors, vous devez l'effacer, vous devez lire SSPBUF, et vous savez que vous avez perdu l'octet reçu. Rassurez-vous, comme vous avez envoyé un « NOACK », le maître le sait aussi, il peut donc répéter l'information.

L'exception étant que s'il s'agit de l'adresse générale (0x00), et qu'un autre esclave qui répond également à cette adresse génère le « ACK », le maître ne pourra pas savoir que vous n'avez pas répondu. Vous savez maintenant que vous devez lire l'adresse afin d'effacer le bit BF.

Si vous avez positionné le bit « CGEN », votre PIC® réagira, soit à l'adresse contenue dans SSPADD, soit à l'adresse générale 0x00. Le test de cette valeur vous indiquera de quelle adresse il est question.

### V.14.4. La génération du « ACK ».

Comme je viens de vous le dire, elle est automatique. On distingue 2 cas :

1. Soit le buffer était vide (BF = 0) et le bit SSPOV était également à 0 lors de la réception de l'adresse : dans ce cas, le bit BF sera positionné et un « ACK » sera automatiquement généré.
2. Soit le bit BF ou le bit SSPOV ou les deux était à « 1 » au moment du transfert, et dans ce cas SSPOV et BF seront positionnés à 1, et un « NOACK » sera généré.

En réalité, générer un « NOACK », je vous le rappelle, équivaut tout simplement à ne rien générer du tout. Si le bit SSPOV était positionné et pas le bit BF, alors c'est que votre programme n'est pas bien écrit (vous avez raté un octet et vous ne vous en êtes pas aperçu, puisque vous avez lu SSPBUF sans effacer SSPOV). Ce cas ne devrait donc jamais arriver.

Le bit SSPIF sera positionné dans tous les cas. Une interruption pourra donc toujours avoir lieu même si la transaction ne s'est pas correctement effectuée, ce qui vous permettra de traiter l'erreur.

### V.14.5. La réception d'un octet de donnée.

Et bien, la réception d'un octet de donnée est strictement similaire à la réception de l'adresse. La seule différence est que le bit D_A sera positionné pour indiquer que l'octet reçu est un octet de donnée. Les cas d'erreurs sur BF et SSPOV fonctionneront de façon identique, je n'ai donc pas grand-chose à dire ici.

De nouveau, un « ACK » sera généré si tout s'est bien passé, et un « NOACK » dans le cas contraire, mettant fin au transfert.

### V.14.6. La réception de l'adresse en mode lecture.

Vous allez maintenant recevoir le repeated start-condition, géré en interne par votre PIC®. Suit directement après l'adresse de votre PIC®, mais avec le bit R/W mis à « 1 » (lecture). Voici ce qui va en résulter :

- ✓ L'adresse sera transférée dans le registre SSPBUF, avec R/W à « 1 ».
- ✓ Dans ce cas particulier, BF ne sera pas positionne.
- ✓ Le bit SSPIF sera positionné, et une interruption aura éventuellement lieu.
- ✓ Le bit D_A du registre SSPSTAT sera mis à 0 (adresse présente).
- ✓ Le bit R_W sera mis à 1 (lecture).
- ✓ L'accusé de réception « ACK » va être généré.

Il est important de constater que lorsqu'on reçoit une adresse en mode esclave et en lecture, bien que SSPBUF contienne l'adresse reçue, BF n'est pas positionné. Ceci est logique, car SSPBUF contient forcément l'adresse mémorisée dans SSPADD, nul besoin de la lire, vous connaissez son contenu. En effet, les commandes générales sont par définition des commandes d'écriture. Votre PIC® va maintenant générer un « ACK » automatiquement, vous n'avez donc pas à vous en préoccuper.

### V.14.7. L'émission d'un octet.

Une fois que votre PIC® a détecté que le maître avait besoin de recevoir un octet, il force automatiquement le bit CKP à « 0 », ce qui place la ligne SCL à « 0 », et donc génère une pause. Ceci est bien entendu nécessaire pour éviter que le maître ne lise votre PIC® avant que vous n'ayez eu le temps d'écrire votre octet de donnée dans SSPBUF.

Ne traînez cependant pas, n 'oubliez pas que vous bloquez l'intégralité du bus I²C. Il vous reste donc à placer votre octet à envoyer dans SSPBUF, puis à placer le bit CKP à 1, ce qui va libérer la ligne SCL et va permettre au maître de recevoir votre donnée. L'écriture de SSPBUF provoque le positionnement de BF, qui sera effacé une fois l'octet effectivement envoyé.

Cette fois, c'est le maître qui enverra le « ACK » s'il désire lire un autre octet, ou un « NOACK » s'il a décidé de terminer. Vous ne disposez d'aucun moyen direct pour lire l'état du « ACK » envoyé. Votre PIC® gère automatiquement ce bit pour savoir s'il doit attendre un nouveau start-condition, ou s'il doit se préparer à l'envoi d'une nouvelle donnée.

Cependant, une astuce vous permet de savoir si un NOACK a été généré par le maître. En effet, la réception du NOACK provoque le reset du bit R_W. Ceci se traduit, en fin de lecture, par la configuration suivante :

- ✓ R_W = 0 (donc on peut croire qu'on est en écriture).
- ✓ SSPIF = 1 (donc l'octet a été reçu).
- ✓ BF = 0 (et le buffer est vide).

Il est évident que si l'opération est terminée et que le buffer est vide, c'est qu'il s'agissait d'une lecture, et non d'une écriture (auquel cas SSPBUF sera plein et BF vaudrait « 1 »). Cette « anomalie » vous permet donc de détecter la présence du NOACK si besoin était.

En effet, le bit CKP réagit différemment selon que le maître a renvoyé un ACK ou un NOACK. Dans le premier cas, CKP passe à 0, alors que dans le second il reste à 1.

## V.14.8. Le mode esclave 7 bits par les interruptions.

Je vous ai dit qu'en mode esclave, le traitement par interruptions était le plus approprié. Il est temps maintenant de vous expliquer comment procéder. De nouveau, vous n'avez qu'une seule interruption, à savoir « SSPIF ». Le bit BCIF n'a aucune raison d'être ici, car seul un maître peut générer et détecter une collision de bus. Or, vous avez plusieurs événements susceptibles de provoquer cette interruption. Il vous faudra donc une méthode pour distinguer quel événement vous avez à traiter.

Tout se base sur l'état des bits suivants, contenus dans le registre SSPSTAT :

- ✓ S : s'il vaut « 1 », une opération est en cours.
- ✓ R_W : « 1 » indique une lecture en cours, « 0 » une écriture.
- ✓ D_A : « 1 » signale que l'octet qui vient d'arriver est une donnée, « 0 » une adresse.
- ✓ BF : « 1 » signale que SSPBUF est plein.

Nous allons, en toute logique, rencontrer les cas suivants :

### a. Premier cas.

- ✓ S = 1 ; transfert en cours.
- ✓ R_W = 0 ; il s'agit d'une écriture.
- ✓ BF = 1 ; la donnée a été reçue.
- ✓ D_A = 0 ; et il s'agit de l'adresse.

Nous voyons donc que le maître désire écrire (nous envoyer des octets). L'adresse de notre PIC® ou l'adresse générale est maintenant dans notre SSPBUF. Nous devons le lire afin d'effacer BF.

### b. Second cas.

- ✓ S = 1 ; transfert en cours.
- ✓ R_W = 0 ; il s'agit d'une écriture.
- ✓ BF = 1 ; la donnée a été reçue.
- ✓ D_A = 1 ; et il s'agit d'une donnée.

Nous venons ici de recevoir une donnée, qu'il nous incombe de ranger dans l'emplacement de notre choix.

***Attention :***

> *Si votre bus I²C travaille à grande vitesse, ou que le traitement de votre interruption est retardé (autre interruption), vous pourriez avoir le bit S=0 et le bit P (stop-condition) = 1. Ceci signifie simplement que le stop-condition a déjà été reçu avant que vous n'ayez eu le temps de traiter la réception de l'octet.*

### c. Troisième cas.

- ✓ S = 1 ; transfert en cours.
- ✓ R_W = 1 ; il s'agit d'une lecture.
- ✓ BF = 0 ; Le buffer est vide (l'adresse en lecture n'influence pas BF).
- ✓ D_A = 0 ; l'adresse a été reçue.

Votre PIC® vient de recevoir son adresse. Comme il s'agit de requête de lecture, le bus est maintenu en pause par votre PIC® qui a placé le bit CKP automatiquement à « 0 ». Une fois que vous avez placé la première donnée à écrire dans votre registre SSPBUF (BF passe à « 1 »), vous remettez CKP à 1, ce qui va permettre l'émission.

Remarquez que, bien que l'adresse soit chargée dans SSPBUF, le bit BF n'a pas été positionné automatiquement. Il est donc inutile de lire SSPBUF.

### d. Quatrième cas.

- ✓ S = 1 ; transfert en cours.
- ✓ R_W = 1 ; il s'agit d'une lecture.
- ✓ BF = 0 ; l'octet a été envoyé (le buffer est vide).
- ✓ D_A = 1 ; et il s'agissait d'une donnée.

Ceci indique que vous devez maintenant placer un octet de donnée qui n'est pas le premier. Le maître vous en réclame donc d'autres. Vous devez donc placer un nouvel octet dans SSPBUF, puis remettre CKP à 1 pour libérer la ligne SCL. Si, suite à une erreur du maître, il vous demandait un octet, alors que vous n'avez plus rien à envoyer, vous n'aurez d'autre solution que de lui envoyer n'importe quoi. Une absence de réaction de votre part entraînerait le blocage du bus I²C.

### e. Cinquième cas.

- ✓ S = 1 ; transfert en cours.
- ✓ R_W = 0 ; il s'agit d'une écriture.
- ✓ BF = 0 ; le buffer est vide.
- ✓ D_A = 1 ; le dernier octet envoyé était une donnée.

En fait, nous avons ici une impossibilité apparente dont j'ai déjà parlé. En effet, si nous sommes dans notre routine d'interruption, c'est que l'opération a été terminée, puisque SSPIF a été positionné.

De quelle opération s'agit-il ? Et bien, d'une écriture. Donc, puisque l'écriture est terminée, la donnée se trouve dans SSPBUF. Or le bit BF n'est pas positionné, c'est donc une situation paradoxale. L'explication est qu'un « NOACK » a été reçu lors d'une lecture, indiquant que le transfert est terminé. Ce NOACK a comme effet de réinitialiser le bit R_W, ce qui nous donne cette configuration paradoxale.

Ce cas est donc tout simplement la concrétisation de la réception d'un NOACK. Vous savez à partir de ce moment que le maître ne vous demandera plus de nouvel octet, à vous de traiter ceci éventuellement dans votre programme.

## V.15. Le module MSSP en mode I²C esclave 10 bits.

Je ne vais pas reprendre l'intégralité des fonctions pour ce cas, je vais me contenter de vous expliquer ce qui diffère lorsque vous décidez de travailler avec des adresses codées sur 10 bits. Nous allons distinguer 2 cas, celui de la lecture et celui de l'écriture.

Commençons par le second. Les opérations à réaliser sont les suivantes (pour le cas où on a correspondance d'adresse) :

- ✓ Le PIC® reçoit le start-condition.
- ✓ On reçoit le premier octet d'adresse, avec R/W à « 0 ».
- ✓ On génère le ACK.
- ✓ On reçoit le second octet d'adresse.
- ✓ On génère le ACK.
- ✓ On reçoit le premier octet.
- ✓ ….

Tout ceci est déjà connu, il ne reste qu'une question : comment savoir que l'on doit placer le second octet d'adresse ? Et bien, tout simplement parce que le bit UA (Update Adress) est automatiquement positionné lorsque vous devez changer l'octet qui se trouve dans SSPADD. Ce positionnement de UA s'accompagne de la mise en pause automatique du bus, l'horloge étant bloquée à l'état bas par le PIC® esclave.

Dès que vous écrivez votre autre octet dans SSPADD, le bit UA est automatiquement effacé, et la ligne SCL est libérée afin de mettre fin à la pause. Les étapes sont donc les suivantes :

- ✓ On reçoit le premier octet d'adresse :
  - Le ACK est généré automatiquement,
  - le bit UA est positionné ainsi que le bit SSPIF.
  - l'horloge SCL est maintenue à l'état bas, plaçant le bus en mode pause.
- ✓ On écrit le second octet d'adresse dans SSPADD :
  - le bit UA est réinitialisé,
  - la ligne SCL est libérée.
- ✓ On reçoit le second octet d'adresse :
  - Le ACK est généré automatiquement,
  - le bit UA est positionné ainsi que le bit SSPIF.
  - Le bus est placé en pause.

- ✓ On écrit le premier octet d'adresse dans SSPADD :
  - o on est alors prêt pour une prochaine réception,
  - o le bit UA est réinitialisé automatiquement et la pause prend fin.

Nous avons donc reçu 2 fois une interruption avec le bit UA positionné. Il suffit donc, en début de notre routine d'interruption, de tester UA. S'il est mis à « 1 », on met à jour SSPADD. S'il contenait l'octet 1 d'adresse, on y place l'octet 2, et réciproquement.

Voyons maintenant le cas de l'écriture :

- ✓ Le PIC® reçoit le start-condition.
- ✓ On reçoit le premier octet d'adresse, avec R/W à « 0 ».
- ✓ On génère le AC.
- ✓ On reçoit le second octet d'adresse.
- ✓ On génère le ACK.
- ✓ On reçoit le repeated start-condition.
- ✓ On reçoit le premier octet d'adresse, avec R/W à « 1 ».
- ✓ On génère le ACK.
- ✓ On envoie le premier octet demandé.

Ceci se traduit, au niveau du PIC® :

- ✓ On reçoit le premier octet d'adresse :
  - o Le ACK est généré automatiquement,
  - o le bit UA est positionné ainsi que le bit SSPIF.
  - o L'horloge SCL est maintenue à l'état bas, plaçant le bus en mode pause.
- ✓ On écrit le second octet d'adresse dans SSPADD :
  - o le bit UA est réinitialisé,
  - o la ligne SCL est libérée.
- ✓ On reçoit le second octet d'adresse :
  - o Le ACK est généré automatiquement,
  - o le bit UA est positionné ainsi que le bit SSPIF.
  - o Le bus est placé en pause.
- ✓ On écrit le premier octet d'adresse dans SSPADD :
  - o on est alors prêt pour une prochaine réception,
  - o le bit UA est réinitialisé automatiquement et la pause prend fin.
- ✓ On reçoit de nouveau le premier octet d'adresse, mais UA n'est pas positionné (c'est une fonction automatique du PIC®). On se retrouve donc maintenant dans le cas d'une lecture avec des adresses de 7 bits (mêmes conditions). Donc, notre algorithme reste valable :
  - o on ne met à jour SSPADD que si UA est positionné.
  - o Sinon, on poursuit le traitement ordinaire de notre interruption.

## V.16. Synthèse.

Nous avons maintenant étudié tous les cas possibles, à savoir :

- ✓ L'émission et la réception en I²C maître.
- ✓ L'émission et la réception en I²C multi-maître.
- ✓ L'émission et la réception en mode esclave sur adresses 7 bits.
- ✓ L'émission et la réception en mode esclave sur adresses 10 bits.
- ✓ La gestion des interruptions.

Je vous rappelle que vous disposez de plusieurs façons de traiter les événements (pooling, interruptions…). Je vous donne ici les méthodes que je vous préconise :

- ✓ Pour le mode maître ou multimaître:
  - Si vous avez le temps d'attendre, je vous conseille d'exécuter les commandes et d'attendre la fin de leur exécution pour sortir de la sous-routine correspondante.
  - Si vous voulez gagner un peu de temps, vous sortez de suite, et vous testez lors de la prochaine sous-routine si la précédente est terminée. Cette méthode est également plus pratique pour le mode multimaître, car le test du bus libre peut être intégré dans la routine.
  - Si vous avez absolument besoin de libérer le PIC® le plus rapidement possible, vous devrez utiliser les interruptions.
- ✓ Pour le mode esclave :
  - Je vous conseille dans tous les cas le traitement par interruption, puisque les événements arrivent de façon asynchrone au déroulement de votre programme, sans oublier qu'un retard de réaction provoque le blocage du bus.
  - Si vous ne souhaitez absolument pas utiliser les interruptions, alors n'oubliez pas de tester toutes les conditions d'erreurs possibles (SSPOV, BF …) afin d'éviter un blocage définitif du bus I²C.

## V.17. Le module I²C en mode sleep.

Si votre PIC® est placé en mode de sommeil, il sera capable dans tous les cas de recevoir des octets de donnée ou d'adresse. Si l'interruption SSPIF est activée, le PIC® sera réveillé par toute correspondance d'adresse ou toute fin d'opération.

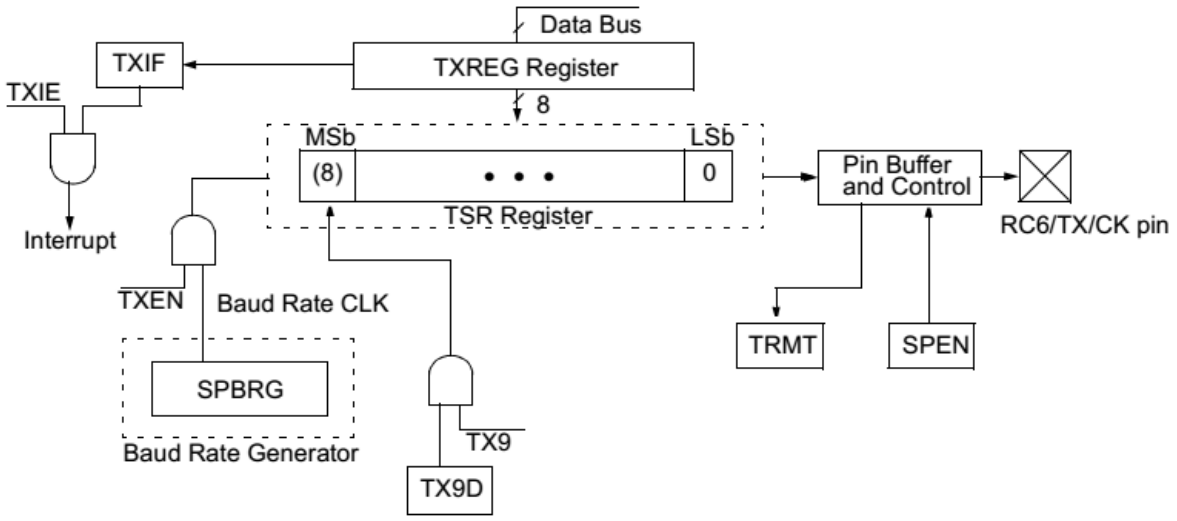
## V.18. Module USART en mode synchrone.

### V.18.1. Introduction.

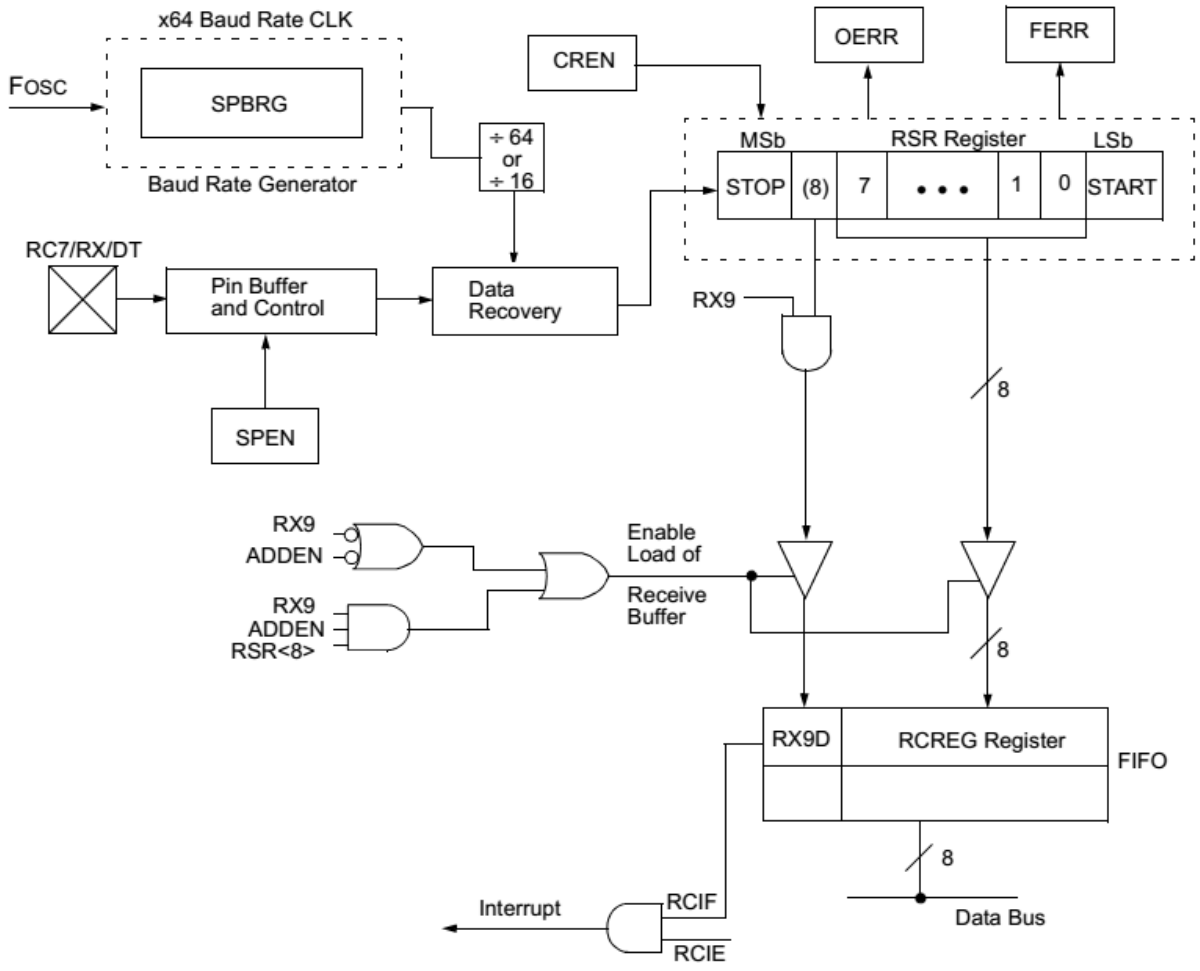
Voici un second module de communication série. Selon l'habitude de Microchip®, celui-ci fonctionne de façon différente par rapport au module MSSP, afin de vous offrir un maximum de possibilités.

USART signifie « Universal Synchronous Asynchronous Receiver Transmitter ». C'est donc un module qui permet d'envoyer et de recevoir des données en mode série, soit de façon synchrone, soit asynchrone. Dans certaines littératures, vous retrouverez également le terme générique de SCI pour « Serial Communications Interface ».

L'électronique correspondant au module USART en transmission est le suivant.




L'électronique correspondant au module USART en réception est le suivant.



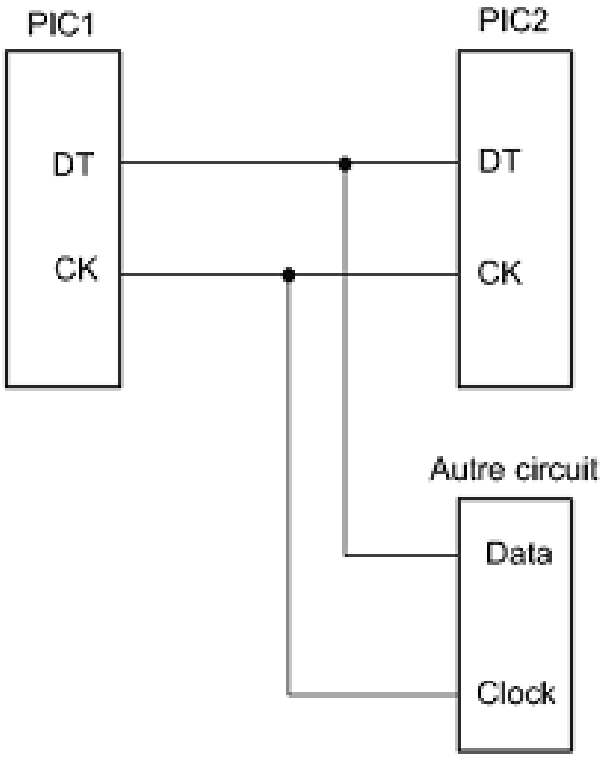

Le module USART de notre PIC® gère uniquement 2 pins, à savoir RC6/TX/CK et RC7/RX/DT. Comme vous savez qu'une liaison série synchrone nécessite une ligne dédicacée à l'horloge, il ne vous reste donc qu'une seule ligne pour transmettre les données. On en déduit que le PIC® ne pourra émettre et recevoir en même temps en utilisant l'USART en mode synchrone. On parlera donc de liaison « half-duplex ».

Par contre, le mode asynchrone n'a pas besoin de ligne d'horloge, il nous restera alors 2 lignes pour communiquer, chacune étant dédicacée à un sens de transfert. Nous pourrons donc envoyer et recevoir des données en même temps. On parlera de liaison « full-duplex ».

Comme, de plus, vous savez maintenant que l'horloge, en mode synchrone, est sous le contrôle du maître, notre USART pourra fonctionner dans les modes suivants :

1. Mode asynchrone full duplex : émission sur TX et réception sur RX.
2. Mode asynchrone half-duplex sur 2 lignes (TX et RX) ou sur une ligne (TX/RX reliées).
3. Mode synchrone maître : émission horloge sur CK et émission/réception données sur DT.
4. Mode synchrone esclave : réception horloge sur CK et émission/réception données sur DT.

## V.18.2. Mise en œuvre et protocoles.




Vous constatez que tous les intervenants peuvent être maîtres ou esclaves. Bien entendu, il ne peut y avoir qu'un maître en même temps, de même qu'il ne peut y avoir qu'un seul émetteur en même temps. C'est donc à vous de gérer ceci. Il existe différentes méthodes, par exemple :

1. Vous décidez que c'est toujours le même élément qui est le maître. C'est donc lui qui administre le bus. Il décide qui peut émettre, et quand. Ceci peut être réalisé, par exemple, en envoyant un octet particulier qui précise qui va répondre. Le maître interroge chaque esclave à tour de rôle en précisant dans son message le numéro de

l'esclave interrogé. Celui-ci répond sous le contrôle du maître. Cette méthode est couramment appelée « spooling ».

2. Le « tocken-ring », ou « anneau à jeton » fonctionne de la façon suivante : Le maître actuel parle à un esclave (il précise par un octet à qui il s'adresse). Il passe alors la parole à l'esclave, qui devient le nouveau maître du bus. Le maître actuel redevient esclave jusqu'à ce qu'un maître lui rende le droit de gestion (jeton).
3. Le « spooling avec request » mélange plusieurs techniques. Les esclaves et le maître sont interconnectés avec une ou plusieurs lignes de sélections supplémentaires (gérées par logiciel). Quand un esclave a quelque chose à dire, il force une ligne de « request ». Le maître sait alors que quelqu'un a quelque chose à dire, il va interroger les intervenants à tour de rôle pour savoir qui a positionné la ligne. Une fois l'esclave interrogé, celui-ci libère la ligne. Notez que cette méthode ressemble à celle utilisée en interne pour gérer les interruptions. On peut améliorer en ajoutant plusieurs lignes de « request » de priorités différentes, et vous en arrivez à la méthode utilisée par le processeur de votre PC pour communiquer avec certains périphériques (Interrupt-ReQuest ou IRQ).
4. Toute autre méthode par laquelle vous vous assurez qu'un seul et unique maître gère le bus à un instant donné, et qu'un seul et unique composant est en train d'émettre. Ceci inclus l'ajout de lignes spécifiques gérées comme des lignes de sélection.

Je vous donne un petit exemple de « tocken-ring ». Supposons qu'à la mise sous tension, le maître est le PIC®1. Vous décidez d'attribuer l'adresse 1 au PIC1, 2 au PIC2, et 3 au troisième composant. Vous devez ensuite vous fabriquer un protocole de communication.

Vous décidez que les trames envoyées (suite d'octets) seront de la forme :

L'octet 1 contient l'adresse du destinataire codée sur 5 bits, le bit 6 indiquant que le destinataire peut répondre, le bit 7 étant le jeton (tocken). Il sera suivi par 2 octets de données. Voici ce que pourrait donner un échange :

- ✓ Le PIC1 (qui est le maître) envoie : B'00000010', B'aaaaaaaa', B'bbbbbbbb'.
- ✓ Le PIC 2 (adresse = 2) sait que les octets lui sont destinés, mais il n'a pas droit de réponse.
- ✓ Le PIC1 envoie : B'01000010', B'cccccccc',B'dddddddd'.
- ✓ Le PIC2 sait que les octets lui sont destinés, le bit 6 du premier octet à « 1 » l'autorise à répondre. Il place sa réponse dans son registre d'émission, mais il reste en esclave.
- ✓ Le PIC1 provoque la lecture, il récupère la réponse du PIC®2 : B'00000001', B'eeeeeeee', B'ffffffff'.
- ✓ Le PIC1 envoie : B'10000011', B'gggggggg',B'hhhhhhhh'.
- ✓ Le PIC1 a donné le jeton (bit7 à 1) au PIC®3 tout en lui transmettant 2 octets.
- ✓ Le PIC1 devient esclave.
- ✓ Le PIC3 devient le maître : il peut continuer les transactions comme il l'entend.

Voici un exemple de ce que pourrait donner un « tocken-ring », comme ça vous pouvez imaginer des exemples pratiques d'application. Ceci pour vous dire que c'est à vous

d'imaginer un protocole de communication (ou de vous conformer à un protocole existant si vous avez besoin d'insérer votre PIC® dans un réseau spécifique).

## V.18.3. le registre TXSTA.

le registre «TRANSMITT STAtus and control register », comme son nom l'indique, contient des bits de status et de contrôle, la plupart concernant l'émission de données. Voici les bits utilisés en mode synchrone :

**TXSTA en mode synchrone**

1. b7 : CSRC : Clock SouRCe select bit (1 = master, 0 = slave).
2. b6 : TX9 : TRANSMITT 9 bits enable bit (1 = 9 bits, 0 = 8 bits).
3. b5 : TXEN : TRANSMITT ENable bit.
4. b4 : SYNC : SYNChronous mode select bit (1 = synchrone, 0 = asynchrone).
5. b3 : N.U. : Non Utilisé : lu comme « 0 ».
6. b2 : non : non utilisé en mode synchrone.
7. b1 : TRMT : TRansMiT shift register status bit (1 = TSR vide).
8. b0 : TX9D : TRANSMIT 9th bit Data.

Voyons ce que signifie tout ceci :

- ✓ ***CSRC*** détermine si votre PIC® va travailler en tant que maître ou en tant qu'esclave. Dans le premier cas, c'est lui qui décide du moment des transferts de données, et qui pilote l'horloge. Dans le second cas, il subira les événements.
- ✓ ***TX9*** indique si vous voulez envoyer des données codées sur 8 ou sur 9 bits. Un bit placé à 1 ici conduira à émettre des données codées sur 9 bits. Ceci peut vous permettre, par exemple, d'envoyer et de recevoir un bit de parité, mais peut également servir à un tout autre usage. Notez que le PIC® ne gère pas automatiquement le calcul de la parité. Si vous décidez de l'utiliser en tant que tel, il vous appartiendra de la calculer.
- ✓ ***TXEN*** permet de lancer l'émission. Comme vous ne pouvez travailler qu'en mode half-duplex, vous mettrez l'émission ou la réception en service, votre USART étant incapable d'effectuer les 2 opérations en même temps (au contraire de votre module MSSP en mode SPI pour qui ce mode était imposé).
- ✓ ***SYNC*** indique si vous travaillez en mode synchrone (1) ou asynchrone (0). Dans ce chapitre, nous traitons le mode synchrone, donc ce bit devra être positionné à « 1 ».
- ✓ ***TRMT*** indique quand le registre TSR est vide, c'est-à-dire quand l'émission de la dernière valeur présente est terminée.
- ✓ ***TX9D*** contient la valeur du 9ème bit à envoyer, quand vous avez décidé, via le positionnement de TX9, de réaliser des émissions de mots de 9 bits. Comme les registres ne peuvent en contenir que 8, il fallait bien caser ce dernier bit quelque part.

## V.18.4. Le registre RCSTA.

RCSTA, pour « ReCeive STAtus and control register », contient d'autres bits de contrôle et de status, axés principalement sur la réception des données. Voici les bits qui le composent.

**RCSTA en mode synchrone.**

1. b7 : SPEN : Serial Port ENable bit.
2. b6 : RX9 : RECEIVE 9 bits enable bit (1 = 9 bits, 0 = 8 bits).
3. b5 : SREN : Single Receive ENAble bit.
4. b4 : CREN : Continuous Receive ENable bit.
5. b3 : non : non utilisé en mode synchrone.
6. b2 : non : non utilisé en mode synchrone.
7. b1 : OERR : Overflow ERRor bit.
8. b0 : RX9D : RECEIVE 9th Data bit.

Quant au rôle de chacun de ces bits :

- ✓ ***SPEN*** met le module USART en service, permettant son utilisation.
- ✓ ***RX9*** permet de choisir une réception sur 8 ou sur 9 bits, exactement comme pour l'émission.
- ✓ ***SREN*** lance la réception d'un seul octet. La communication se terminera d'elle-même à la fin de la réception de l'octet.
- ✓ ***CREN*** lance la réception continue. Les octets seront reçus sans interruption jusqu'à ce que ce bit soit effacé par vos soins.
- ✓ ***OERR*** indique une erreur de type overflow. Cette erreur survient si vous n'avez pas traité de façon assez rapide les octets précédemment reçus.
- ✓ ***RX9D*** contient le 9ème bit de votre donnée reçue, pour autant, bien entendu, que vous ayez activé le bit RX9.

Vous voyez que vous avez le choix entre émission (bit TXEN), réception d'un octet (SREN), ou réception continue (CREN). Or, vous ne pouvez émettre et recevoir en même temps. Que se passe-t-il donc si vous activez plusieurs de ces bits en même temps ? En fait, Microchip® a déterminé une priorité dans l'ordre de ces bits, de façon à ce qu'un seul soit actif à un moment donné.

Les priorités sont les suivantes :

- ✓ Si CREN est activé, on aura réception continue, quel que soit l'état de SREN et TXEN.
- ✓ Si CREN n'est pas activé, et que SREN est activé, on aura réception simple, quel que soit l'état de TXEN.
- ✓ On n'aura donc émission que si TXEN est activé, alors que CREN et SREN sont désactivés.

## V.18.5. Le registre SPBRG.

Ce registre « Serial Port Baud Rate Generator » permet de définir la fréquence de l'horloge utilisée pour la transmission, et donc de fixer le débit de la communication.

Il est évident que SPBRG ne sert, dans le cas de la liaison synchrone, que pour le maître, puisque l'esclave reçoit son horloge de ce dernier, et n'a donc aucune raison de la calculer.

La formule qui donne le débit pour le mode synchrone est :

$$D = \frac{Fosc}{4 \times (SPBRG + 1)}$$

Comme vous aurez souvent besoin de calculer SPBRG en fonction du débit souhaité, voici la formule transformée :

$$SPBRG = \frac{Fosc}{4 \times D} - 1$$

Prenons un cas concret. Imaginons que vous vouliez réaliser une transmission à 19200 bauds avec un PIC® cadencé à 20MHz. Quelle valeur devons-nous placer dans SPBRG ?

SPBRG = (20.000.000 / (4 * 19200)-1 = 259,4.

Il va de soi que non seulement on doit arrondir, mais en plus, la valeur maximale pouvant être placée dans SPBRG est de 255 (décimal). Donc, nous prendrons SPBRG = 255, ce qui nous donne un débit réel de :

D = 20.000.000 / (4 * (255+1)) = 19531 bauds.

L'erreur sera de : Erreur = (19531 – 19200) / 19200 = 1,72% C'est une erreur tout à fait acceptable. N'oubliez pas qu'en mode synchrone, vous envoyez l'horloge en même temps que le signal, votre esclave va donc suivre l'erreur sans problème, à condition de rester dans ses possibilités électroniques.

Remarquez que plus on augmente SPBRG, plus on diminue le débit. Nous venons sans le vouloir de calculer que le débit minimum de notre PIC® maître cadencé à 20MHz sera de 19531 bauds. Si vous avez besoin d'un débit plus faible, il ne vous restera comme possibilité que de diminuer la fréquence de votre Quartz. De la même façon, si vous avez besoin d'un débit particulièrement précis, il vous appartiendra de choisir un quartz qui vous permettra d'obtenir une valeur entière de SPBRG dans votre calcul.

Calculons maintenant le débit maximum possible avec notre PIC® cadencée à 20MHz. Ce débit sera obtenu, en bonne logique, avec une valeur de SPBRG de « 0 ».

Dmax = 20000000 / (4 * (0+1)) = 5.000.000 = 5 MBauds.

Vous constatez une fois de plus que les liaisons synchrones sont prévues pour travailler avec de très grandes vitesses.

## V.18.6. L'initialisation.

Pour initialiser votre module en mode synchrone, il vous faudra.

- ✓ Choisir si vous travaillez en mode maître ou esclave.
- ✓ Décider si vous utilisez des émissions sur 8 ou sur 9 bits.
- ✓ Positionner votre bit SYNC pour le travail en mode synchrone.
- ✓ Décider si vous communiquez en 8 ou en 9 bits.

- ✓ Si vous travaillez en maître, initialiser la valeur de SPBRG.
- ✓ Mettre le module en service.

Par défaut, à la mise sous tension, les pins CK et DT sont configurées en entrée, il n'est donc pas nécessaire d'initialiser leur bit respectif dans TRISC, sauf si vous avez entre-temps modifié ce registre.

## V.18.7. L'émission en mode maître.

Le module est considéré comme configuré par la précédente routine. Vous validez la mise en service de l'émission en positionnant le bit TXEN. Si vous utilisez le format de donnée sur 9 bits, vous devez commencer par placer la valeur du 9ème bit dans le bit TX9D. Ensuite, vous placez la donnée à émettre dans le registre TXREG (TRANSMITT REGister). Cette donnée est transférée dès le cycle d'instruction suivant, ainsi que le bit TX9D, dans son registre TSR (Transmitt Shift Register). Ce registre n'est pas accessible directement par votre programme.

Il va effectuer l'envoi de vos bits de données sur la ligne DT en effectuant des décalages vers la droite. C'est donc le bit de poids faible (b0) qui sera envoyé en premier, au contraire du module MSSP qui effectuait ce décalage vers la gauche, et donc commençait là le bit de poids fort (b7).

Dès que l'octet est transféré dans TSR (et donc avant qu'il ne soit complètement transmis), le registre TXREG se retrouve donc vide. Ceci vous est signalé par le positionnement du flag d'interruption TXIF. Comme le transfert entre TXREG ne s'effectue que si la transmission du mot contenu dans TSR est terminée, ceci vous laisse la possibilité, sans écraser le contenu actuel de TSR, d'écrire une nouvelle valeur dans TXREG.

Le chargement d'une valeur dans TXREG s'accompagne de l'effacement automatique du flag TXIF. C'est d'ailleurs la seule façon de l'effacer.

Si TXREG ne contient plus aucun octet à envoyer, lorsque TSR aura terminé l'envoi de son dernier octet, le bit TRMT passera à « 1 », indiquant la fin effective de l'émission.

L'écriture d'une nouvelle valeur dans TXREG effacera de nouveau TRMT. Donc, en résumé, imaginons l'envoi de 2 octets :

- ✓ Vous placez le 9eme bit dans TX9D, puis l'octet à envoyer dans TXREG.
- ✓ Le bit TRMT passe à « 0 » (émission en cours).
- ✓ Le bit TXIF passe à « 0 » (registre TXREG plein).
- ✓ L'octet est transféré dans TSR, l'émission commence.
- ✓ Le registre TXREG est maintenant vide, le flag TXIF passe à 1.
- ✓ Vous chargez TX9D et le second octet à envoyer (le PIC® est toujours en train d'émettre l'octet précédent).
- ✓ Le bit TRMT est toujours à 0.
- ✓ Le bit TXIF passe à « 0 » (registre TXREG plein).
- ✓ A ce stade, TSR continue l'émission de l'octet 1, TXREG contient l'octet 2.
- ✓ TSR termine l'émission de l'octet 1, transfert de l'octet 2 de TXREG vers TSR.
- ✓ Le flag TXIF passe à « 1 » (TXREG vide).

- ✓ L'émission de l'octet 2 se poursuit.
- ✓ L'émission de l'octet 2 se termine, TRMT passe à « 1 » (émission terminée).

Bien évidemment, si vous travaillez avec des mots de 8 bits, vous n'avez pas à vous occuper du bit TX9D. Si vous avez tout compris, dans le cas où vous avez plusieurs octets à envoyer :

- ✓ Vous avez toujours un octet en cours de transmission dans TSR, le suivant étant déjà prêt dans TXREG.
- ✓ Si on a besoin de 9 bits, on place toujours TX9D avant de charger TXREG.
- ✓ Chaque fois que TXIF passe à 1 (avec interruption éventuelle), vous pouvez recharger TXREG avec l'octet suivant (tant que vous en avez).
- ✓ Quand la communication est terminée, TRMT passe à 1.

Notez que si vous utilisez les interruptions, le bit TXIF ne peut être effacé manuellement, il ne l'est que lorsque votre registre TXREG est rechargé. Ceci implique que lorsque vous placez votre dernier octet à envoyer, vous devez effacer TXIE avant de sortir pour interdire toute nouvelle interruption, le flag TXIF étant automatiquement repositionné à chaque fois que TXREG sera vide.

Pour qu'il y ait une émission, il faut donc que, le module étant initialisé et lancé :

- ✓ SREN et CREN soient désactivés.
- ✓ TXEN soit positionné.
- ✓ Le registre TXREG soit chargé.

La seconde et la troisième ligne peuvent être inversées, ce qui vous laisse 2 possibilités de démarrer une écriture.

- ✓ Soit vous validez TXEN, et vous chargez TXREG au moment où vous désirez lancer l'émission. Ceci est la procédure « normale ».
- ✓ Soit vous préparez TXREG avec la valeur à émettre, et vous positionnez TXEN. Cette dernière procédure permet de gagner un peu de temps, surtout sur les communications à faibles débits.

## V.18.8. L'émission en mode esclave.

Vous travaillez en mode esclave exactement comme en mode maître, excepté que vous n'avez pas à vous préoccuper du registre SPBRG.

Vous placerez donc vos données de la même façon, et vous serez prévenu que votre registre TXREG est vide par le positionnement de l'indicateur TXIF. La fin de l'émission se conclura également par le positionnement du bit TRMT. C'est bien le maître qui décide quand a effectivement lieu votre émission, mais vous aurez à gérer strictement les mêmes événements.

## V.18.9. La réception en mode maître.

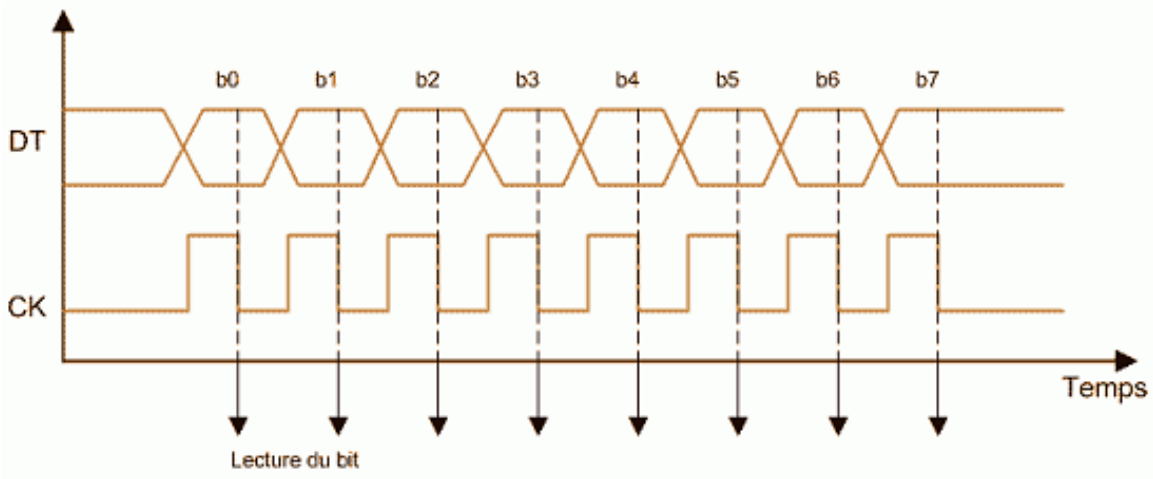
Comme il fallait s'y attendre, la réception des données met en œuvre 2 autres registres. RSR (Receive Shift Register) est utilisé pour la réception et TSR pour l'émission, il réalise la

réception des bits en effectuant un décalage vers la droite (souvenez-vous que pour l'USART, b0 est transmis en premier lieu).

Dès que RSR est plein, son contenu est transféré dans RCREG, qui est le seul des 2 registres accessible par votre programme. Il contient la donnée effectivement reçue, complétée éventuellement (si vous travaillez sur 9 bits) par le bit RX9D. La procédure est la suivante :

- ✓ Vous positionnez SREN ou CREN, ce qui a pour effet que votre PIC® commence à envoyer l'horloge. L'esclave qui a la parole s'est entre-temps occupé de préparer son émission.
- ✓ Quand les 8 impulsions d'horloge (ou 9 pour le mode 9 bits) ont été envoyés, l'octet a été reçu.
- ✓ L'octet est transféré dans le registre RCREG (le 9ème bit éventuel est transféré vers RX9D), et le flag RCIF est positionné.
- ✓ Vous lisez alors le bit RX9D éventuel puis le registre RCREG, ce qui provoque l'effacement du bit RCIF.

Si vous aviez choisi de positionner SREN, le cycle est terminé, l'horloge est stoppée. Par contre, si vous aviez préféré CREN, l'horloge continue d'envoyer ses impulsions, provoquant les lectures continues des autres mots à recevoir, jusqu'à ce que vous resettiez CREN. Notez que la lecture d'un bit s'effectue sur le flanc descendant de l'horloge.




Le fonctionnement est donc relativement simple, mais il importe de tenir compte de plusieurs points importants :

### a. La file FIFO de RCREG.

Le registre RCREG dispose d'une file FIFO de 2 emplacements. C'est un petit cadeau de Microchip®, qui vous permet de prendre un peu de retard dans la réaction aux événements entrants. La signalisation qu'il y a au moins un élément à traiter étant dévolu à RCIF. Comment cela fonctionne-t-il ? En fait, de façon totalement transparente pour l'utilisateur.

Imaginons que les octets arrivent, et que vous réagissiez avec retard :

- ✓ Le premier octet est en cours de réception dans RSR, RCREG est vide, donc RCIF est à « 0 ».
- ✓ Le premier octet est reçu et transféré dans la file de RCREG, RCIF est positionné, le second octet est en cours de réception dans RSR.
- ✓ Le second octet est reçu et transféré dans la file de RCREG, RCIF reste positionné, le troisième octet est en cours de réception dans RSR.

Comme on n'a que 2 emplacements, vous devrez réagir avant la fin de la réception du troisième octet, mais vous voyez que votre temps de réaction peut être allongé. Voici alors ce que vous allez faire :

- ✓ Vous lisez, si vous travaillez sur 9 bits, le 9ème bit, RX9D.
- ✓ Vous lisez RCREG, donc le premier octet reçu. Cette lecture ne provoque pas l'effacement de RCIF, puisque RCREG n'est pas vide. Par contre, le 9ème bit du second octet est transféré à ce moment dans RX9D, ce qui explique que vous deviez le lire en premier.
- ✓ Vous lisez alors, si vous travaillez sur 9 bits, le bit RX9D qui est le 9ème bit de votre second octet.
- ✓ Vous lisez donc encore RCREG, donc le second octet reçu. Le flag RCIF est effacé, le registre RCREG est vide.

Vous voyez que vous n'avez pas à vous occuper de ce FIFO. Tant que RCIF est positionné, vous lisez, c'est tout. Il faut seulement se souvenir qu'il faut impérativement lire RX9D avant de lire RCREG.

### b. L'erreur d'overflow.

Nous avons vu que le registre RCREG pouvait contenir 2 emplacements. Imaginons maintenant que vous réagissiez avec tellement de retard qu'un troisième octet soit reçu avant que vous ayez lu les 2 précédents mémorisés dans la file d'attente. Comme il n'y a plus de place dans la file, le mot reçu dans RSR sera tout simplement perdu.

Pour vous signaler ce fait, le bit OERR (Overflow ERRor) sera automatiquement positionné. Il vous appartiendra de remettre OERR à « 0 ». Cette remise à « 0 » se fait de façon un peu particulière, en effaçant le bit CREN pour mettre fin à la réception continue. Le bit OERR est en effet à lecture seule.

Attention, il est impératif de couper CREN (puis de le remettre le cas échéant). Dans le cas contraire, OERR ne serait pas remis à « 0 », ce qui bloquerait la réception des octets suivants. Votre réaction devrait donc être du style :

- ✓ Tant que RCIF est positionné.
- ✓ Je lis RX9D puis RCREG.
- ✓ RCIF n'est plus positionné, OERR est positionné ?
- ✓ Non, pas de problème.
- ✓ Oui, je sais que j'ai perdu un octet et je coupe CREN.

Evidemment, vous avez beaucoup moins de chance d'avoir une erreur d'overflow si vous travaillez via les interruptions, ce que je vous conseille fortement dans ce mode.

### V.18.10. La réception en mode esclave.

Tout comme pour l'émission, très peu de différences entre le maître et l'esclave. Les événements à gérer sont identiques et se traitent de la même façon. Notez cependant que le mode d'émission simple n'est pas utilisable dans le cas de l'esclave. Le bit SREN n'est donc pas géré si le PIC® est dans ce mode (CSRC = 0). Vous devrez alors utiliser uniquement CREN.

### V.18.11. Le mode sleep.

Une fois de plus, un simple raisonnement vous permettra de déduire le fonctionnement du PIC® en mode de sommeil. L'horloge CK est dérivée de l'horloge principale du PIC®. Comme cette horloge principale est stoppée durant le sommeil, il est clair que le PIC® configuré en maître ne peut ni émettre ni recevoir en mode « sleep ». Par contre, pour l'esclave, cette contrainte n'existe pas. Aussi, si une émission ou une réception est lancée avant la mise en sommeil, le PIC® sera réveillé si le bit de validation correspondant (RXIE et/ou TXIE) est positionné.

## V.19. Le module USART en mode asynchrone.

### V.19.1. Le mode série asynchrone.

Nous savons maintenant ce qu'est une liaison série synchrone. Une liaison série asynchrone, comme son nom l'indique, fonctionne de la même façon, en émettant les bits les uns à la suite des autres, mais sans fournir le signal d'horloge qui a permis de les générer. Ceci a forcément plusieurs conséquences, dont les suivantes :

- ✓ Comme le récepteur d'une donnée ne reçoit pas le signal d'horloge, il doit savoir à quelle vitesse l'émetteur a généré son transfert. C'est en effet la seule façon pour lui de savoir quand commence et quand fini un bit.
- ✓ Comme émetteur et récepteurs se mettent d'accord pour adopter une vitesse commune, et étant donné que chacun travaille avec sa propre horloge, de légères différences peuvent apparaître sur les 2 horloges, introduisant des dérives.

Il faudra gérer ceci :

- ✓ Il faut un mécanisme qui permette de détecter le début d'une donnée. En effet, imaginons que la ligne soit au niveau « 1 » et que l'émetteur décide d'envoyer un « 1 ». Comment savoir que la transmission a commencé, puisque rien ne bouge sur la ligne concernée ?
- ✓ De la même façon, il faut établir un mécanisme pour ramener la ligne à son état de repos en fin de transmission. Ceci étant indispensable pour permettre la détection de la donnée suivante.
- ✓ Notez enfin que dans les transmissions asynchrones, la grande majorité des liaisons procèdent en commençant par l'envoi du bit 0.

Ce sera notre cas. La liaison utilisera les 2 mêmes pins que pour la liaison synchrone, à savoir RC6/TX/CK et RC7/RX/DT. Les dénominations qui conviendront dans ce cas seront bien entendu TX pour l'émission et RX pour la réception. Ces pins devront être configurées en entrée via TRISC pour fonctionner en mode USART.

### a. Le start-bit.

Au repos, nous allons imaginer que notre ligne soit au niveau « 1 ». Je choisis ce niveau parce que c'est celui présent sur la ligne d'émission de votre PIC®. Rien n'empêche de travailler avec une électronique qui modifie ces niveaux.

Donc, notre ligne se trouve à « 1 ». Or, nous n'avons qu'une seule ligne dédicacée à un sens de transfert, donc nous disposons d'un seul et unique moyen de faire savoir au destinataire que la transmission a commencé, c'est de faire passer cette ligne à « 0 » (startbit). On peut donc simplement dire que le start-bit est :

- ✓ Le premier bit émis.
- ✓ Un bit de niveau toujours opposé au niveau de repos (donc 0 dans notre cas).
- ✓ Un bit d'une durée, par convention, la même que celle d'un bit de donnée.

### b. Les bits de donnée.

Nous avons envoyé notre start-bit, il est temps de commencer à envoyer nos bits de donnée. En général, les bits de donnée seront au nombre de 7 ou de 8 (cas les plus courants). La durée d'un bit est directement liée au débit choisi pour la connexion. En effet, le débit est exprimé en bauds, autrement dit en bits par seconde. Donc, pour connaître la durée d'un bit, il suffit de prendre l'inverse du débit.

$$Tb = 1 / Débit.$$

Par exemple, si on a affaire à une connexion à 9600 bauds, nous aurons :

$$Tb = 1 / 9600 \text{ bauds} = 104{,}16 \text{ µs}.$$

### c. La parité.

Directement après avoir envoyé nos 7 ou 8 bits de données, nous pouvons décider d'envoyer ou non un bit de parité. Ce bit permet d'imposer le nombre de bits à « 1 » émis (donnée + parité), soit comme étant pair (parité paire), soit comme étant impair (parité impaire). Le récepteur procède à la vérification de ce bit, et, si le nombre de bits à « 1 » ne correspond pas à la parité choisie, celui-ci saura que la réception ne s'est pas effectuée correctement.

La parité permet de détecter les erreurs simples, elle ne permet ni de les réparer, ni de détecter les erreurs doubles. Par contre, la détection fréquente d'erreurs de parité indique un problème dans l'échange des données. Voici par exemple, un cas de parité paire :

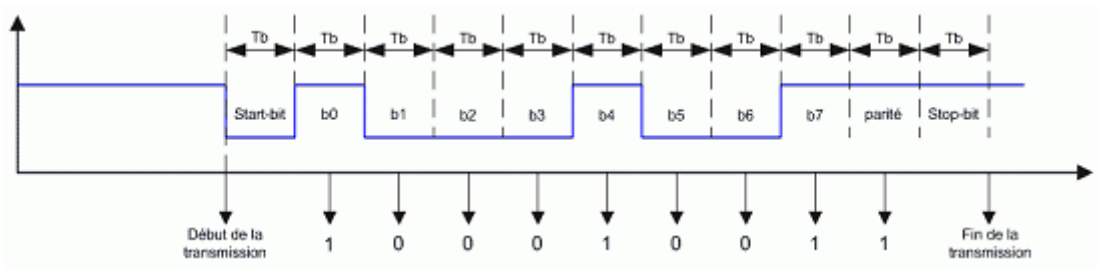
- ✓ La donnée vaut : B'00110100'.
- ✓ Nous avons 3 bits à « 1 ».
- ✓ Donc, pour avoir notre parité paire, notre bit de parité devra donc valoir « 1 », ce qui forcera le nombre total de bits « 1 » à valoir « 4 », ce qui est bien pair.

- ✓ On enverra donc : B'001101001' : 4 bits à « 1 » = parité paire.

### d. Le stop-bit.

Nous avons maintenant émis tous nos bits de donnée et notre parité éventuelle. Reste à permettre à notre ligne de transmission de revenir à l'état de repos. Ce passage est dénommé « stop-bit ». On peut dire à son sujet que :

- ✓ Il sera le dernier bit émis de l'octet.
- ✓ Son niveau est toujours celui du niveau de repos (donc 1 dans notre cas). - Sa durée sera par convention la même que celle d'un bit de donnée. Le stop-bit n'est pas comptabilisé (pas plus que le start-bit) pour le calcul du bit de parité. Notez qu'il est permis d'imposer l'utilisation de 2 « stop-bits ». Ceci est le cas, par exemple, de la norme ISO 7816. Ceci revient simplement à dire qu'il s'agit d'un stop-bit qui dure une durée équivalante à celle de 2 bits. Une autre façon de voir les choses, est de dire qu'on ne pourra envoyer l'octet suivant qu'après une durée Tb après l'émission du premier stop-bit. Ou, dit encore autrement, on doit ménager une pause d'une durée de 1 bit, entre l'émission de 2 octets consécutifs. Un mot concernant la lecture d'un bit par le récepteur. Il est évident que la lecture de chaque bit doit s'effectuer dans la zone présentant la plus grande probabilité de stabilité du bit. En général, les électroniques sont construites pour mesurer le bit au milieu de sa durée. Voici, pour illustrer tout ceci, l'émission d'une donnée codée sur 8 bits, avec parité paire et un stop-bit. Les lectures sont indiquées par les flèches inférieures :




Notez, pour information, que la lecture d'un bit par le PIC® ne s'effectue pas en une seule fois. En réalité, le PIC® effectuera 3 mesures consécutives centrées sur le milieu présumé du bit. Ces 3 mesures sont entrées dans une porte majoritaire, qui définit quel sera le niveau considéré comme exact. Ceci évite qu'un parasite n'engendre une erreur de lecture.

Une porte majoritaire, est un circuit qui fournit sur sa sortie l'état majoritairement présent sur sa ou ses entrées. Autrement dit, si votre PIC® a lu, pour un bit donné, 2 fois la valeur « 1 » et une fois la valeur « 0 », le bit sera considéré comme valant « 1 », puisque « 1 » est présent majoritairement (2 contre 1). Ceci est cependant transparent pour vous, vous n'avez pas à vous en préoccuper.

### e. Les modes compatibles.

Nous avons vu que le transfert d'une donnée sera composée de :

- ➢ 1 Start-bit ;
- ➢ 7 ou 8 bits de donnée ;

- 0 ou 1 bit de parité ;
- 1 ou 2 stop-bits.

Ceci nous donne des longueurs totales comprises entre 9 et 12 bits. Notre PIC® nous permet, lui, de gérer 8 ou 9 bits de données, la parité doit être gérée manuellement et intégrée dans les bits de données. Un seul stop-bit est émis. Ceci nous donne :

- 1 Start-bit ;
- 8 ou 9 bits de donnée ;
- 1 stop-bit.

Les longueurs possibles seront comprises entre 10 et 11 bits.

***Cas 1 :*** 1 start-bit, 7 bits de donnée, 0 bit de parité, 1 stop-bit.
Dans ce cas, la transmission s'effectue sur 9 bits, ce que ne permet pas le PIC®. Pour l'émission, on peut réaliser l'astuce suivante : On réalise une émission avec 8 bits de donnée, et on placera :

- Les 7 bits de donnée dans les bits 0 à 6 ;
- Le stop-bit dans le bit 7 (donc bit 7 toujours à « 1 »).

Ceci se traduira par la réception, vue par l'interlocuteur de votre PIC®, d'une donnée comportant 1 start-bit, 7 bits de donnée, 0 bit de parité, et 2 stop-bits (le faux, placé dans le bit 7 de la donnée, et le vrai, envoyé automatiquement par le module USART).

La réception est par contre impossible. En effet, si l'interlocuteur envoie ses données de façon jointive, le start-bit de la donnée suivante sera reçu au moment où le PIC® s'attend à recevoir le stop-bit de la donnée en cours. Donc, ceci générera une erreur de réception. Par chance, ce mode est extrêmement rare.

***Cas 2 :*** **1 start-bit, 7 bits de donnée, 0 bit de parité, 2 stop-bits.**
Nous voici avec une transmission sur 10 bits, ce qui ne devrait pas poser de problème. Forts de ce qui précède, nous en concluons que nous réaliserons une transmission avec 8 bits de données, pour laquelle :

- Les 7 bits de donnée seront placées dans les bits 0 à 6 ;
- Le premier stop-bit sera placé dans le bit 7 (donc toujours à « 1 »).

Emission et réception ne posent donc aucun problème.

***Cas 3 :*** 1 start-bit, 7 bits de donnée, 1 bit de parité, 1 stop-bit.
De nouveau, transmission sur 10 bits, donc sans problème avec une donnée codée sur 8 bits :

- Les 7 bits de donnée seront placés dans les bits 0 à 6 ;
- La parité sera placée dans le bit 7.

Cas 4 : 1 start-bit, 7 bits de donnée, 1 bit de parité, 2 stop-bits.
Nous voici avec une liaison sur 11 bits, donc nous choisirons une longueur de donnée de 9 bits pour notre PIC®. Ceci nous donne :

- ✓ Les 7 bits de donnée dans les bits 0 à 6 ;
- ✓ Le bit de parité dans le bit 7 ;
- ✓ Le premier bit de stop dans le bit 8 (9ème bit).

### Cas 5 : 1 start-bit, 8 bits de donnée, 0 bit de parité, 1 stop-bit.

Un cas classique, nous choisirons une longueur de donnée de 8 bits. La donnée ne nécessite aucun traitement. C'est le cas le plus simple. Les 8 bits de données sont placés dans les bits 0 à 7.

### Cas 6 : 1 start-bit, 8 bits de donnée, 0 bit de parité, 2 stop-bits.

Revoici une liaison sur 11 bits, donc nous choisirons une longueur de donné de 9 bits, ce qui nous donne :

- ✓ Les 8 bits de donnée dans les bits 0 à 7 ;
- ✓ Le premier stop-bit dans le bit 8 (9ème bit toujours à « 1 »).

De nouveau, c'est un cas très simple.

### Cas 7 : 1 start-bit, 8 bits de donnée, 1 bit de parité, 1 stop-bit.

Encore une liaison sur 11 bits, donc toujours une longueur de donnée de 9 bits. Ici, le bit de parité sera dans le 9ème bit, soit :

- ✓ Les 8 bits de donnée dans les bits 0 à 7 ;
- ✓ Le bit de parité dans le bit 8 (9ème bit).

C'est le cas le plus classique que vous rencontrerez souvent.

### Cas 8 : 1 start-bit, 8 bits de donnée, 1 bit de parité, 2 stop-bits.

Cette fois, nous avons une liaison sur 12 bits, soit hors possibilités de notre PIC®. Pour nous en servir, on procède comme suite:

**En réception** :

- ✓ Les 8 bits de données se retrouvent dans les bits 0 à 7 ;
- ✓ Le bit de parité se retrouve dans le bit 8 (9ème bit) ;
- ✓ Le second stop-bit est tout simplement considéré comme un temps mort, et est tout simplement ignoré par notre USART.

En résumé, en réception, vous n'avez pas à vous préoccuper du second stop-bit.

**En émission :**

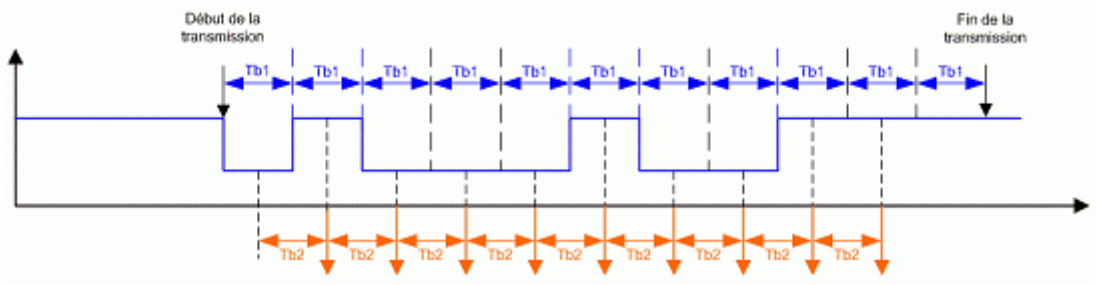
- ✓ On place les 8 bits de données dans les bits 0 à 7 ;
- ✓ On place la parité dans le bit 8 ;
- ✓ A la fin de l'émission, on attend un temps supplémentaire correspondant à 1 bit avant d'envoyer la donnée suivante.

Vous rencontrerez ce cas si vous réalisez des programmes pour la norme ISO 7816, par exemple. En résumé, en émission, vous créez le second stop-bit en introduisant un délai entre l'émission de 2 données.

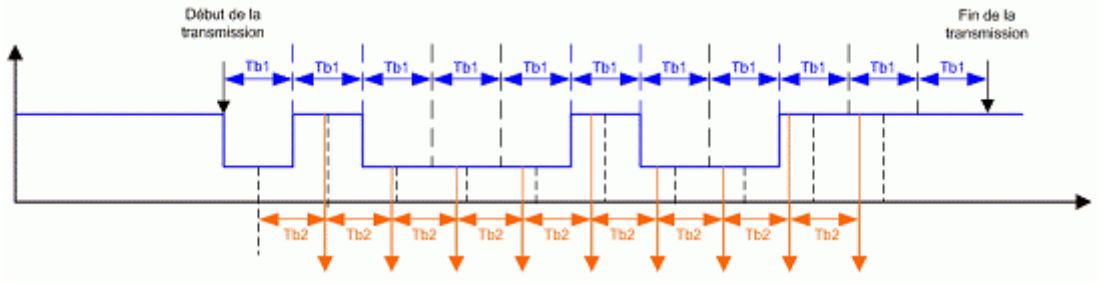
### f. Les erreurs de synchronisation.

En mode synchrone, émetteur et récepteur adoptent une vitesse d'horloge qui est sensée être identique sur les 2 composants. Seulement, ces horloges ne sont jamais exactement identiques, ce qui entraîne une erreur qui se cumule au fur et à mesure de la réception des bits. Bien entendu, la transmission sera resynchronisée au star-bit suivant. La dérive maximale est donc obtenue lors de la réception du dernier bit (stop-bit).

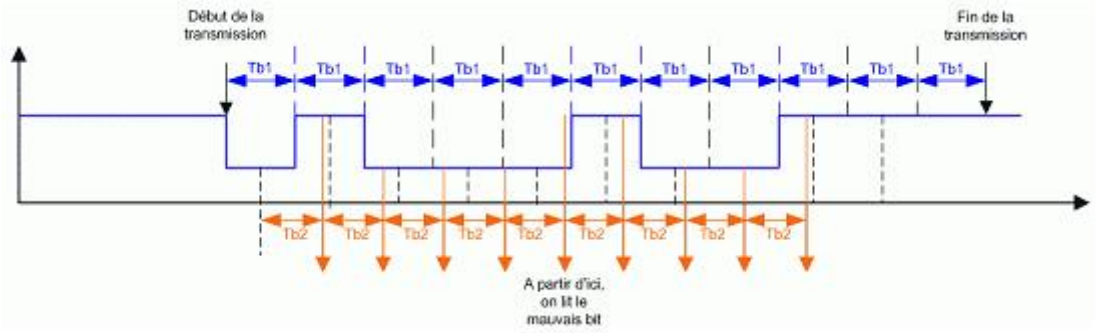
Prenons tout d'abord un cas idéal : Soit Tb1 le temps d'émission d'un bit par l'émetteur, et Tb2 le temps séparant 2 lectures par le récepteur. Dans ce cas idéal, les 2 horloges sont parfaitement égales, donc Tb1 = Tb2. Autrement dit, la lecture s'effectuera toujours au moment optimal. Voici ce que cela donne :




Maintenant, voyons le cas où l'horloge du récepteur n'est pas tout à fait identique. Imaginons qu'elle tourne légèrement plus vite, donc tb2 est inférieur à Tb1 :




On constate que le décalage entre le milieu réel du bit (en pointillé) et le moment de la lecture par le récepteur (en rouge) augmente au fur et à mesure de la réception des bits. Dans ce cas-ci, l'erreur est encore acceptable, puisque chaque lecture se fait durant le bit concerné. Cette erreur ne provoquera donc pas d'erreur de lecture. Que se passe-t-il si on augmente encore cette erreur ?




Et bien, on constate maintenant qu'à partir d'un certain endroit, la lecture est tellement décalée qu'elle s'effectue sur le mauvais bit. La réception est donc impossible. Quelle sera l'erreur maximale acceptable ? Il suffit pour cela de raisonner.

Lorsque nous lirons le dernier bit, il est évident que nous ne devons pas tomber à côté, l'erreur admissible maximale au bout des lectures de tous nos bits est donc inférieure à la durée de la moitié d'un bit. Comme l'erreur est cumulative, on peut dire que :

Au bout de « n » bits lus, l'erreur maximale doit être de moins de 50% du temps d'un bit. Autrement dit, l'erreur de l'horloge (qui intervient pour chaque bit) doit avoir une erreur maximale inférieure à 50% divisés par le nombre total de bits à lire.

Si nous prenons un exemple concret, en l'occurrence une transmission sur 10 bits, l'erreur maximale de l'horloge permise sera de :

$$\text{Erreur} < 50\% / 10.$$

$$\text{Erreur} < 5\%.$$

Autrement dit, vous devrez travailler avec une erreur inférieure à 5% pour la majorité de vos liaisons. En pratique, comme vous ne connaissez pas non plus la précision de votre correspondant, et que les flancs des signaux ne sont pas parfaits, la barre des 2% paraît réaliste.

Reste à définir comment calculer l'erreur théorique. C'est très simple, vous prenez la valeur absolue de la différence entre le débit réel obtenu et le débit idéal utilisé par votre interlocuteur, divisée par le débit idéal de votre interlocuteur. Autrement dit :

$$Erreur\ théorique = \frac{|Débit\ réel - Débit\ idéal|}{Débit\ idéal}$$

Si on considère, par exemple, que vous travaillez avec un débit réel de 9615 sur une ligne à 9600 bauds, vous aurez une erreur de :

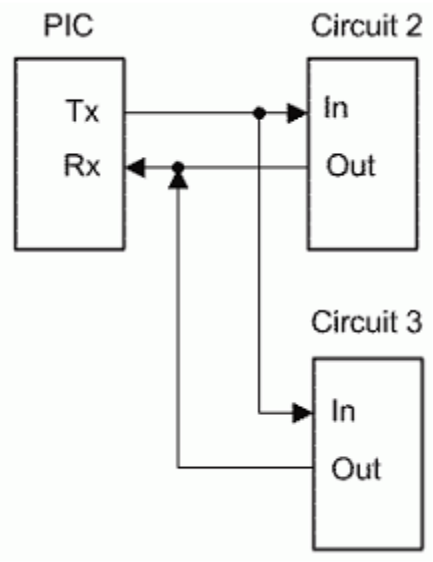
$$| 9615 - 9600 | / 9600 = 0{,}0015, \text{ soit } 0{,}15\%$$

A cette erreur, pour être précis, nous devrions ajouter la dérive de votre horloge et l'erreur de votre interlocuteur. Si vous utilisez un quartz, votre dérive est négligeable, quant à la dérive de votre interlocuteur, vous devrez consulter ses datasheets.

Cependant, en restant sous la barre des 2%, vous serez assurés, presqu'à coup sûr, de ne pas avoir de problème.

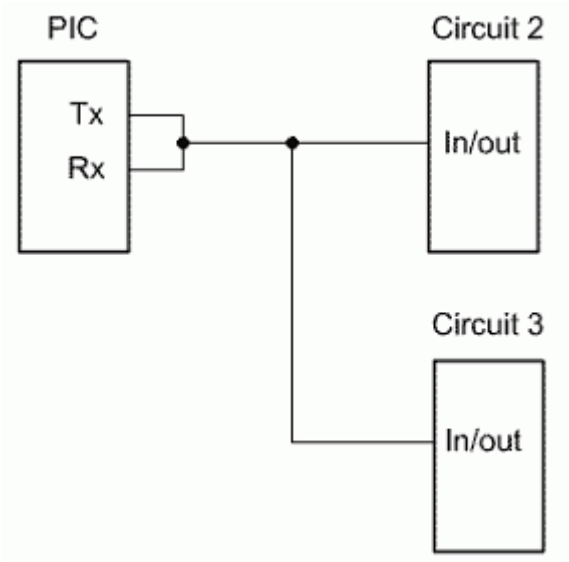
## V.19.2. Mise en œuvre.

Le PIC® est capable de travailler en mode full-duplex. Ceci vous laisse plusieurs possibilités de configuration. Voyons tout d'abord comment interconnecter les circuits dans ce mode :




Dans ce cas, notre PIC® est privilégié, puisqu'il peut parler avec le circuit 2 et 3, alors que ces circuits ne peuvent dialoguer entre eux. Vous voyez donc que le mode full-duplex procure un inconvénient. Soit on se contente de communiquer entre 2 composants, soit, si on a plus de 2 composants, un seul se trouve en situation privilégiée.

Voyons maintenant le cas de la liaison half-duplex :

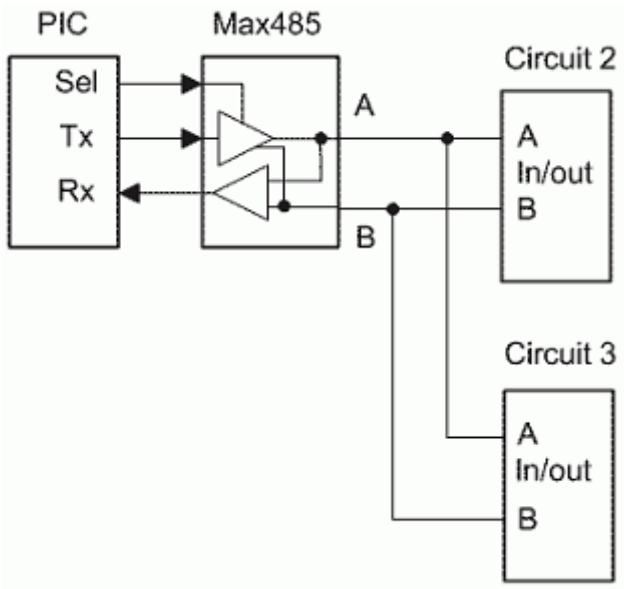



Dans ce cas, tous les composants peuvent parler entre eux. Bien entendu, il vous faudra alors définir des protocoles pour qu'un seul interlocuteur ne parle à la fois.

Le revers de la médaille, c'est que votre PIC® peut émettre et recevoir, mais pas les deux en même temps. Au repos, la ligne TX ne force pas réellement un niveau haut, en réalité elle se configure en entrée, ce qui permet ce mode de fonctionnement. Vous mettrez ce mode en action, généralement, via un bus particulier, et donc en utilisant un driver spécifique.

Prenons l'exemple de la norme RS485. Cette norme, utilisée en industrie, permet de communiquer sur de grandes distances (plus de 1Km), à des débits assez élevés, et en mode half-duplex. La ligne qui relie les différents périphériques est constituée de 2 fils torsadés qui véhiculent des tensions opposées. Selon qu'un ou l'autre fil se trouve porté à un potentiel supérieur à l'autre, nous aurons la présence d'un « 1 » ou d'un « 0 ».

Ne vous y trompez pas : il y a 2 fils, mais qui véhiculent la même information (mode différentiel), et non un fil affecté à chaque sens de transmission. Lorsque votre PIC® lit, il doit se retirer de l'émission, ce qui s'effectue en désélectionnant l'émetteur du convertisseur de bus RS485 (en l'occurrence, un MAX485). La ligne « Sel » est une ligne quelconque de votre PIC® configurée en sortie.




Notez le driver RS485, qui convertit les niveaux 0V/5V en 2 lignes différentielles nécessaires pour le RS485. La ligne Sel du PIC® (qui est une pin I/O quelconque), et que vous devez gérer vous-même, permet de placer le max485 en mode haute-impédance, et donc de permettre la lecture sans perturber le réseau. A un moment donné, un seul composant peut être en mode d'écriture, tous les autres écoutent. Il va de soi que les circuits 2 et 3 possèdent une électronique comparable.

## V.19.3. le registre TXSTA.

Venons-en à nos PIC®, et en particulier aux registres et mécanismes mis en œuvre dans le cadre de nos liaisons série asynchrones. Commençons par le registre TXSTA.

Je vais surtout ici vous montrer ses spécificités dans le mode asynchrone.

**TXSTA en mode asynchrone.**

1. b7 : non : non utilisé en mode asynchrone.
2. b6 : TX9 : TRANSMITT 9 bits enable bit (1 = 9 bits, 0 = 8 bits).
3. b5 : TXEN : TRANSMITT ENable bit.
4. b4 : SYNC : SYNChronous mode select bit (1 = synchrone, 0 = asynchrone).
5. b3 : N.U. : Non Utilisé : lu comme « 0 ».
6. b2 : BRGH : Baud Rate Generator High mode select bit.
7. b1 : TRMT : TRansMiT shift register status bit (1 = TSR vide).
8. b0 : TX9D : TRANSMIT 9th bit Data.

Rappelons ce que signifie tout ceci :

1. TX9 indique si vous voulez envoyer des données codées sur 8 ou sur 9 bits. Un bit placé à 1 ici conduira à émettre des données codées sur 9 bits.
2. TXEN permet de lancer l'émission. Cette fois, en mode asynchrone, votre USART pourra émettre et recevoir en même temps. Rien donc ne vous empêche de sélectionner émission et réception simultanée. Nous ne retrouverons donc pas ici la notion de priorité de commande.
3. SYNC indique si vous travaillez en mode synchrone (1) ou asynchrone (0). Dans ce paragraphe, nous traitons le mode asynchrone, donc ce bit devra être positionné à «0».
4. BRGH permet de choisir entre 2 prédiviseurs internes pour la génération des bits en fonction de SPBRG. Nous aurons un mode grande vitesse (BRGH = 1) et un mode basse vitesse (BRGH = 0). Nous en reparlerons au moment de l'étude de SPBRG.
5. TRMT indique quand le registre TSR est vide, c'est-à-dire quand l'émission de la dernière valeur présente est terminée.
6. TX9D contient la valeur du 9ème bit à envoyer, quand vous avez décidé, via le positionnement de TX9, de réaliser des émissions de mots de 9 bits.

## V.19.4. Le registre RCSTA.

### RCSTA en mode asynchrone.

1. b7 : SPEN : Serial Port ENable bit.
2. b6 : RX9 : RECEIVE 9 bits enable bit (1 = 9 bits, 0 = 8 bits).
3. b5 : non : non utilisé en mode asynchrone.
4. b4 : CREN : Continuous Receive ENable bit.
5. b3 : ADDEN : ADDress detect ENable bit.
6. b2 : FERR : Frame ERRor.
7. b1 : OERR : Overflow ERRor bit.
8. b0 : RX9D : RECEIVE 9th Data bit.

Quant au rôle de chacun de ces bits :

1. SPEN met le module USART en service.
2. RX9 permet de choisir une réception sur 8 ou sur 9 bits, exactement comme pour l'émission.

3. CREN lance la réception continue. Les octets seront reçus sans interruption jusqu'à ce que ce bit soit effacé par vos soins. Dans le mode asynchrone, il n'existe que ce mode de réception.
4. FERR indique une erreur de trame. Ceci se produit lorsqu'au moment où devrait apparaître le stop-bit en mode lecture, l'USART voit que la ligne de réception est à «0». Ce bit est chargé de la même façon que le bit RX9D. Autrement dit, ce n'est pas un flag à effacer, c'est un bit qui est lié à la donnée qu'on vient de lire. On doit donc lire ce bit, tout comme RX9D, avant de lire RCREG. Une fois ce dernier lu, le nouveau FERR écrase l'ancien. Si vous remarquvez, on a donc un bit FERR vu comme s'il était le 10ème bit du mot reçu.
5. OERR indique une erreur de type overflow. Il se retrouve positionné si vous n'avez pas lu RCREG suffisamment vite. L'effacement de ce bit nécessite le reset de CREN. Le non effacement de OERR positionné bloque toute nouvelle réception d'une donnée.
6. RX9D contient le 9ème bit de votre donnée reçue, pour autant, bien entendu, que vous ayez activé le bit RX9.

## V.19.5. Le registre SPBRG.

Ce registre, tout comme pour le mode synchrone, permet de définir le débit qui sera utilisé pour les transferts. C'est le même registre qui est utilisé pour émission et réception. De ce fait, un transfert full-duplex s'effectuera toujours avec une vitesse d'émission égale à celle de réception. Par contre, en mode half-duplex, rien ne vous empêche de modifier SPBRG à chaque transition émission/réception et inversément.

Nous avons 2 grandes différences pour ce registre par rapport au mode synchrone :

- ✓ Le registre SPBRG doit toujours être configuré, puisque tous les interlocuteurs doivent connaître le débit utilisé.
- ✓ La formule utilisée pour le calcul de SPBRG n'est pas identique à celle du mode synchrone. En réalité, nous avons 2 formules distinctes, selon que nous aurons positionné le bit BRGH à « 1 » ou à « 0 ».
  - **Si BRGH = 0 (basse vitesse)**.

$$\boldsymbol{Débit = \frac{Fosc}{(64 * (SPBRG + 1))}}$$

Ou encore.

$$\boldsymbol{SPBRG = \left(\frac{Fosc}{Débit * 64}\right) - 1}$$

Les limites d'utilisation sont (avec notre PIC®) à 20MHz :
Débit min = 20*106 / (64 * (255+1)) = 1221 bauds
Débit max = 20*106 / (64 * (0+1)) = 312.500 bauds

  - **Si BRGH = 1 (haute vitesse)**.

$$\boldsymbol{Débit = \frac{Fosc}{(16 * (SPBRG + 1))}}$$

Ou encore.

$$SPBRG = \left(\frac{Fosc}{Débit * 16}\right) - 1$$

Les limites d'utilisation sont (avec notre PIC®) à 20MHz :
Débit min = 20*106 / (16 * (255+1)) = 4883 bauds.
Débit max = 20*106 / (16 * (0+1)) = 1.250.000 bauds.

Vous voyez tout de suite que le mode basse vitesse permet d'obtenir des vitesses plus basses, tandis que le mode haute vitesse permet de monter plus haut en débit. Il existe une zone couverte pas les 2 modes. Dans ce cas, vous choisirez BRGH = 1 si cela vous permet d'obtenir le taux d'erreur le moins élevé.

## V.19.6. Emission, réception, et erreur d'overflow.

Les mécanismes sont exactement les mêmes que pour le mode synchrone. Les flags utilisés sont les mêmes. La seule différence est que maintenant vous pouvez émettre et recevoir en même temps.

Voici donc un petit résumé :

**Pour l'émission**.

- ✓ Valider la mise en service de l'émission en positionnant le bit TXEN.
- ✓ La validation de TXEN positionne le flag TXIF à « 1 » puisque le registre TXREG est vide.
- ✓ Dans cas d'une émission de donnée sur 9 bits, commencer par placer la valeur du 9ème bit dans le bit TX9D.
- ✓ Ensuite, placer la donnée à émettre dans le registre TXREG (TRANSMITT REGister).
- ✓ Cette donnée est transférée dès le cycle d'instruction suivant, ainsi que le bit TX9D, dans son registre TSR. TRMT passe à « 0 » (TSR plein).
- ✓ Recommencer éventuellement le chargement de TX9D + TXREG, le flag TXIF passe à « 0 » (TXREG plein).
- ✓ Dès que le premier octet est émis, le second est transféré dans TSR, le registre TXREG est vide, TXIF est positionné. Il est possible de charger le troisième octet, et ainsi de suite.
- ✓ Quand le dernier octet est transmis, TRMT passe à 1 (TSR vide, émission terminée).

**Avec toujours les mêmes remarques :**

- ✓ Vous avez toujours un octet en cours de transmission dans TSR, le suivant étant déjà prêt dans TXREG.
- ✓ Si on a besoin de 9 bits, on place toujours TX9D avant de charger TXREG.
- ✓ Chaque fois que TXIF passe à 1 (avec interruption éventuelle), vous pouvez recharger TXREG avec l'octet suivant (tant que vous en avez).
- ✓ Quand la communication est terminée, TRMT passe à 1.
- ✓ TXIF passe à « 1 » dès que TXEN est positionné.

Pour qu'il y ait une émission, il faut donc que, le module étant initialisé et lancé :

- ✓ TXEN soit positionné.
- ✓ Le registre TXREG soit chargé.

**Pour la réception.**

- ✓ Vous positionnez CREN, ce qui a pour effet que votre PIC® est prêt à recevoir une donnée.
- ✓ L'octet reçu est transféré dans le registre RCREG (le 9ème bit éventuel est transféré vers RX9D), et le flag RCIF est positionné.
- ✓ Vous lisez alors le bit RX9D éventuel puis le registre RCREG, ce qui provoque l'effacement du bit RCIF.

De nouveau, nous retrouvons notre file FIFO, qui nous permettra de recevoir 2 octets (plus un en cours de réception), avant d'obtenir une erreur d'overflow.

Pour rappel :

- ✓ Le premier octet est en cours de réception dans RSR, RCREG est vide, donc RCIF est à « 0 ».
- ✓ Le premier octet est reçu et transféré dans la file de RCREG, RCIF est positionné, le second octet est en cours de réception dans RSR.
- ✓ Si vous ne réagissez pas de suite, le second octet est reçu et transféré dans la file de RCREG, RCIF reste positionné, le troisième octet est en cours de réception dans RSR. Comme on n'a que 2 emplacements, vous devrez réagir avant la fin de la réception du troisième octet, mais vous voyez que votre temps de réaction peut être allongé.

Voici alors ce que vous allez faire dans ce cas :

- ✓ Vous lisez, si vous travaillez sur 9 bits, le 9ème bit, RX9D, puis RCREG, donc le premier octet reçu. Cette lecture ne provoque pas l'effacement de RCIF, puisque RCREG n'est pas vide. Par contre, le 9ème bit du second octet est transféré à ce moment dans RX9D, ce qui explique que vous deviez le lire en premier.
- ✓ Vous lisez alors, si vous travaillez sur 9 bits, le bit RX9D qui est le 9ème bit de votre second octet, puis RCREG, donc le second octet reçu. Le flag RCIF est effacé, le registre RCREG est vide. L'erreur d'overflow sera générée si un troisième octet est reçu alors que vous n'avez pas lu les deux qui se trouvent dans la file FIFO. Le positionnement empêche toute nouvelle réception. L'effacement de ce bit nécessite d'effacer CREN.

Votre réaction devrait donc être du style :

- ✓ Tant que RCIF est positionné.
- ✓ Je lis RX9D puis RCREG.
- ✓ RCIF n'est plus positionné, OERR est positionné ?.
- ✓ Non, pas de problème.
- ✓ Oui, je sais que j'ai perdu un octet et je coupe CRREN évidemment.

Vous avez beaucoup moins de chance d'avoir une erreur d'overflow si vous travaillez via les interruptions, ce que je vous conseille fortement dans ce mode.

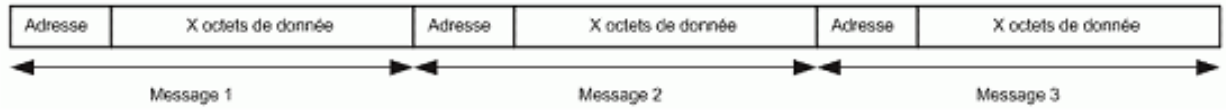
## V.19.7. L'erreur de frame.

L'erreur de frame survient si, au moment où l'USART s'attend à recevoir un stop-bit, il voit que le niveau de sa ligne de réception est positionné à « 0 ». Dans ce cas, quelque chose s'est mal déroulée. Le bit FERR sera positionné, mais, attention, ce bit fait partie de la file FIFO, tout comme le bit RX9D. Donc, vous lirez RX9D et FERR, et, ensuite, vous lirez RCREG. A ce moment, le nouveau bit FERR concernant l'octet suivant écrasera le FERR actuel, exactement comme le nouveau bit RX9D effacera l'actuel. Vous n'avez donc pas à effacer FERR, qui est, du reste, en lecture seule.

## V.19.8. La gestion automatique des adresses.

Nous avons vu que notre PIC® pouvait communiquer avec plusieurs circuits en mode asynchrone. Nous allons donc devoir « inventer » un mécanisme qui permette de savoir à qui on s'adresse. Ceci peut être réalisé en insérant dans les données envoyées et reçues, un octet qui représentera l'adresse du destinataire.

Imaginons, par exemple, que nous décidions d'envoyer l'adresse du destinataire, suivie par les octets de données. Dans ces octets de données pourra se trouver également la longueur du message, l'adresse de l'expéditeur, etc., mais tel ne sera pas le cas ici. Je m'intéresse à l'adresse du destinataire. Imaginons donc que notre PIC® reçoive des trames du type :




Vous voyez tout de suite que se pose un problème. En effet, il faut pouvoir détecter la séparation entre 2 messages, pour pouvoir localiser l'octet qui représente l'adresse du destinataire. On pourrait dire qu'il suffit de connaître la longueur du message, mais, s'il y avait la moindre interférence sur la ligne, tout serait définitivement décalé avec une impossibilité de resynchroniser les messages. On peut alors penser à 2 méthodes.

La première consiste à introduire un délai entre 2 messages, de façon à permettre leur séparation :


Si nous voulons implémenter la réception de ceci de façon logicielle, nous aurons quelque chose du style :

**Interruption réception caractère USART** :

- ✓ On resette le timer.
- ✓ Si le compteur de caractères = 0, il s'agit de l'adresse, on met à jour l'adresse reçue.
- ✓ On lit le caractère reçu (obligatoire pour les flags).

- ✓ Si la dernière adresse reçue correspond avec celle du PIC®,
- ✓ On sauve le caractère reçu.
- ✓ On incrémente le compteur de caractères.
- ✓ Fin de l'interruption.

**Interruption timer (programmé pour déborder si supérieur au temps d'arrêt) :**

- ✓ On resette le compteur de caractères.
- ✓ On signale que le message est complet et peut être traité.
- ✓ Fin de l'interruption.

On voit donc que cette méthode présente 3 inconvénients :

- ✓ On doit gérer un timer avec le temps d'arrêt.
- ✓ On doit lire tous les octets, même s'ils ne nous concernent pas.
- ✓ Le temps d'arrêt ralentit la vitesse de transfert des octets.

Imaginons alors une autre technique, qui consisterait à reconnaître qu'on a affaire à un octet d'adresse ou de donnée. A partir du moment où on n'a besoin que de 8 bits de données, par exemple, il nous reste le 9ème bit qui permettrait de décider si on a affaire à l'adresse du destinataire ou à une donnée. Voici ce que cela donnerait :


Plus besoin de temps mort, le début d'un message est caractérisé par le fait que le premier octet reçu est l'adresse du destinataire et contient son 9ème bit (bit8) positionné à « 1 ». Notre routine de réception serait alors du type :

**Interruption réception USART :**

- ✓ On lit le bit8 et l'octet reçu.
- ✓ Si bit8 = 1 (adresse),
- ✓ Mettre adresse reçue à jour.
- ✓ Si bit8 = 0 (donnée) et si adresse = adresse du PIC®,
- ✓ sauver octet reçu.
- ✓ Fin de l'interruption.

Quels sont maintenant les inconvénients (si on excepte que le bit8 est dédicacé à la détection adresse/donnée) ?

- ✓ On doit lire tous les octets, même s'ils ne nous concernent pas.
- ✓ Le bit 8 ralentit la vitesse de transfert des octets.

On a déjà éliminé l'inconvénient de l'utilisation du timer et le délai entre les 2 messages. Vous allez me dire que vous perdez le bit de parité (si vous comptiez utiliser le bit8 dans ce rôle). Ceci peut être compensé par d'autres mécanismes, comme le « checksum ». Pour

utiliser un checksum, vous faites, par exemple, la somme de tous les octets envoyés, somme dont vous n'utilisez que l'octet de poids faible (donc vous ignorez les débordements). Vous envoyez cette somme comme dernier octet du message, le destinataire effectue la même opération et vérifie le checksum reçu. De plus, c'est plus rapide à calculer que de calculer la parité pour chaque octet. Donc, vous utilisez un octet de plus dans les messages, mais vous gagnez en sécurité et en vitesse de traitement.

En effet, une parité ne permet que la vérification paire ou impaire, soit une chance sur deux de fonctionnement à chaque octet. Le checksum permet une probabilité d'erreur de 1/256 par message. Mais revenons à notre adresse…

Ce qui serait pratique, serait de ne pas devoir lire les octets qui ne nous concernent pas. Nous devrions donc pouvoir suspendre toute réception d'octet jusqu'à ce qu'on reçoive l'octet d'adresse suivant. Autrement dit :

- ✓ Je lis l'octet d'adresse.
- ✓ Si cette adresse est la mienne, je lis les octets suivants.
- ✓ Sinon, je ne lis plus rien et j'attends l'octet d'adresse suivant.

Et bien, bonne nouvelle, ce mécanisme peut être mis en service en positionnant le bit ADDEN du registre RCSTA. Dans ce cas, votre USART fonctionne de la façon suivante :

- ✓ Si ADDEN = 0, tous les octets sont réceptionnés normalement.
- ✓ Si ADDEN = 1, seuls les octets dont le bit8 vaut « 1 » sont réceptionnés. Les autres ne sont pas transférés de RSR vers RCREG, les flags ne sont donc pas affectés, l'interruption n'a pas lieu.

Donc, il vous suffit alors de travailler de la sorte :

**Interruption réception USART (ADDEN vaut « 1 » au démarrage) :**

- ✓ SI ADDEN vaut 1,
- ✓ Si l'octet reçu ne correspond pas à l'adresse du PIC®, fin de l'interruption.
- ✓ Sinon, on efface le bit ADDEN et fin de l'interruption.
- ✓ Si ADDEN vaut 0,
- ✓ On sauve l'octet reçu.
- ✓ Si c'était le dernier, on positionne ADDEN.
- ✓ Fin de l'interruption.

Vous voyez que vous ne passez dans votre routine d'interruption que si les données vous concernent ou que s'il s'agit d'une nouvelle adresse. Vous perdrez donc le minimum de temps. Il vous reste à détecter la fin du message, problème que vous rencontrez d'ailleurs pour les autres cas. La méthode utilisée sera généralement :

- ✓ Soit d'utiliser des messages de longeur fixe.
- ✓ Soit d'envoyer un octet supplémentaire qui indique la longueur du message en octets.

Dans les 2 cas, même en cas de perturbation, votre réception sera resynchronisée automatiquement grâce au bit8, et il vous suffit, pour détecter le moment de repositionnement de ADDEN, de compter les octets reçus. Rien de plus simple.