

LA PROGRAMMATION DES PICS

PAR BIGONOFF



SECONDE PARTIE – Révision 12

LA GAMME MID-RANGE PAR L'ETUDE DES 16F87X
(16F876-16F877)

1. INTRODUCTION.....	9
2. LES CARACTÉRISTIQUES DES 16F87X.....	11
2.1 CARACTÉRISTIQUES GÉNÉRALES DE LA FAMILLE 16F87X.....	11
2.2 ORGANISATION DE LA RAM	12
2.3 STRUCTURE DU 16F876	12
3. LES PARTICULARITÉS DE LA PROGRAMMATION DU 16F87X.....	15
3.1 LA DIRECTIVE « _CONFIG ».....	15
3.2 UTILISATION DE LA MÉMOIRE RAM	16
3.2.1 L'adressage direct.....	16
3.2.2 L'adressage indirect.....	18
3.3 L'UTILISATION DU REGISTRE PCLATH	18
3.3.1 PCLATH et les calculs d'adresse.....	19
3.3.2 PCLATH et les sauts directs.....	19
4. SAUTS DE PAGES ET PREMIER PROJET SUR MPLAB	23
4.1 STRUCTURE ET UTILISATION DU FICHIER « MAQUETTE ».....	24
4.2 CRÉATION D'UN PROGRAMME SANS SAUT DE PAGE.....	28
4.3 PREMIÈRES TENTATIVES DE SAUT INTER-PAGES.....	28
4.4 EVITONS LE PIÈGE	31
4.5 PREMIÈRE CORRECTION DE NOTRE EXERCICE.....	32
4.6 EVITONS LES WARNINGS INUTILES	33
4.7 DÉMONSTRATION PRATIQUE DE L'UTILITÉ DES MACROS	34
4.8 LES SOUS-PROGRAMMES INTER-PAGES	38
5. LES SOURCES D'INTERRUPTIONS.....	43
5.1 ENUMÉRATION.....	43
5.2 LE REGISTRE INTCON ET LES INTERRUPTIONS PÉRIPHÉRIQUES	44
5.2.1 Mise en service des interruptions primaires	44
5.2.2 Mise en service des interruptions périphériques.....	45
5.3 LES REGISTRES PIE1, PIE2, PIR1 ET PIR2	45
5.3 ETUDE DE LA ROUTINE D'INTERRUPTION DU FICHIER « MAQUETTE ».....	47
6. MISE EN ŒUVRE ET CONFIGURATION MINIMALE	57
6.1 LE MATÉRIEL NÉCESSAIRE	57
6.2 LE SCHÉMA MINIMUM	58
6.3 LES PARTICULARITÉS ÉLECTRIQUES.....	59
7. MIGRATION DU 16F84 VERS LE 16F876.....	63
7.1 SIMILITUDES ET DIFFÉRENCES AVEC LE 16F84.....	63
7.2 CONVERSION D'UN PROGRAMME ÉCRIT POUR LE 16F84 VERS LE 16F876.....	63
7.3 CAUSES DE NON FONCTIONNEMENT	64
7.3 CONVERSION D'UN EXEMPLE PRATIQUE.....	65
7.3.1 Réalisation du montage.....	65
7.3.2 Création du projet.....	66
7.3.3 La méthode conseillée	67
8. OPTIMISONS UN PEU	79
8.1 LES DEUX GRANDS TYPES D'OPTIMISATION.....	79
8.2 TRAVAIL SUR UN EXEMPLE CONCRET.....	79
8.3 LE CHOIX DU TYPE D'OPTIMISATION	82
8.4 APPLICATION PRATIQUE.....	82
8.4 OPTIMISATIONS PARTICULIÈRES	92
8.4.1 Optimisation des niveaux de sous-programmes.....	93
8.4.2 L'organisation des ressources.....	100
9. LES DIFFÉRENTS TYPES DE RESET.....	105

9.1 LES 6 TYPES DE RESET	105
9.2 LE RESET ET LES REGISTRES STATUS ET PCON	105
9.3 DÉTERMINATION DE L'ÉVÉNEMENT	106
10. LES PORTS ENTRÉE/SORTIE	111
10.1 LE PORTA	111
10.2 LE PORTB.....	112
10.3 LE PORTC.....	112
10.4 LE PORTD	112
10.5 LE PORTE.....	113
10.5.1 Le registre TRISE.....	113
11. LE PORTD EN MODE PSP	117
11.1 A QUOI SERT LE MODE PSP ?	117
11.1 COMMENT PASSER EN MODE PSP ?	117
11.2 CONNEXION DES PORTD ET PORTE EN MODE PSP	118
11.3 LE FONCTIONNEMENT LOGICIEL.....	118
11.4 LE FONCTIONNEMENT MATÉRIEL	120
12. LES ACCÈS À LA MÉMOIRE EEPROM	123
12.1 LE REGISTRE EECON1	123
12.2 L'ACCÈS EN LECTURE	124
12.3 L'ACCÈS EN ÉCRITURE	125
12.4 INITIALISATION D'UNE ZONE EEPROM	126
13. LES ACCÈS À LA MÉMOIRE PROGRAMME.....	127
13.1 GÉNÉRALITÉS	127
13.2 LES ACCÈS EN LECTURE	127
13.3 LES ACCÈS EN ÉCRITURE	129
13.4 PARTICULARITÉS ET MISE EN GARDE.....	130
13.5 INITIALISATION D'UNE ZONE EN MÉMOIRE PROGRAMME.....	131
13.6 LA TECHNIQUE DU « BOOTLOADER »	131
13.6 APPLICATION PRATIQUE : UN CHENILLARD	131
14. LE TIMER 0.....	149
14.1 GÉNÉRALITÉS	149
14.2 L'ÉCRITURE DANS TMR0	149
14.3 LE TIMING EN MODE COMPTEUR	150
14.4 MODIFICATION « AU VOL » DE L'ASSIGNATION DU PRÉDIVISEUR	152
14.4.1 Prédiviseur du timer 0 vers le watchdog.....	152
14.4.2 Prédiviseur du watchdog vers le timer 0.....	153
15. LE TIMER 1.....	155
15.1 CARACTÉRISTIQUES DU TIMER 1	155
15.2 LE TIMER 1 ET LES INTERRUPTIONS.....	156
15.3 LES DIFFÉRENTS MODES DE FONCTIONNEMENT DU TIMER1	156
15.4 LE REGISTRE T1CON.....	157
15.5 LE TIMER 1 EN MODE « TIMER ».....	159
15.6 LE TIMER 1 EN MODE COMPTEUR SYNCHRONES	159
15.7 LE TIMER 1 EN MODE COMPTEUR ASYNCHRONES.....	161
15.8 LE TIMER 1 ET TOSCEN.....	162
15.9 UTILISATION DU DÉBORDEMENT.....	164
15.10 UTILISATION D'UNE LECTURE	166
15.11 ÉCRITURE DU TIMER 1	170
15.12 EXERCICE PRATIQUE	172
15.12.1 Un peu de maths.....	173
15.12.2 Le programme	176
15.13 ERRATA : FONCTIONNEMENT NON CONFORME	181
16. LE DEBUGGAGE « PIN-STIMULUS ».....	185

17. LE TIMER 2.....	189
17.1 CARACTÉRISTIQUES DU TIMER 2	189
17.2 LE TIMER 2 ET LES INTERRUPTIONS.....	190
17.2 LE TIMER 2 ET LES REGISTRES PR2 ET T2CON.....	190
17.3 UTILISATION PRATIQUE DE NOTRE TIMER 2.....	193
18. RÉCAPITULATIF SUR LES TIMERS.....	199
18.1 LE CHOIX DU TIMER	199
18.1.1 Vous désirez mesurer un temps compris entre 256 et 65536 cycles.....	199
18.1.2 Vous désirez mesurer des temps allant jusque 524288	199
18.1.3 Vous désirez mesurer des temps quelconques avec une grande précision.....	199
18.1.4 Vous désirez compter des événements.....	200
18.2 LES BASES DE TEMPS MULTIPLES	200
18.3 POSSIBILITÉS NON ENCORE ABORDÉES.....	201
19. LE CONVERTISSEUR ANALOGIQUE/NUMÉRIQUE	203
19.1 PRÉAMBULE.....	203
19.2 NOMBRES NUMÉRIQUES, ANALOGIQUES ET CONVERSIONS.....	203
19.3 PRINCIPES DE CONVERSION SUR LES 16F87X	208
19.4 LE TEMPS D'ACQUISITION	209
19.5 LA CONVERSION.....	212
19.6 COMPROMIS VITESSE/PRÉCISION.....	215
19.7 LES VALEURS REPRÉSENTÉES.....	217
19.8 CONCLUSIONS POUR LA PARTIE THÉORIQUE.....	218
19.9 LA THÉORIE APPLIQUÉE AUX PICS : PINS ET CANAUX UTILISÉS	219
19.10 LES TENSIONS DE RÉFÉRENCE	220
19.11 MESURE D'UNE TENSION ALTERNATIVE.....	224
19.12 LES REGISTRES ADRESL ET ADRESH	227
19.13 LE REGISTRE ADCON1	227
19.14 LE REGISTRE ADCON0	230
19.15 LA CONVERSION ANALOGIQUE/NUMÉRIQUE ET LES INTERRUPTIONS.....	232
19.16 L'UTILISATION PRATIQUE DU CONVERTISSEUR	232
19.17 EXERCICE PRATIQUE SUR LE CONVERTISSEUR A/D	234
19.18 CONCLUSION.....	248
20. LES MODULES CCP1 ET CCP2	251
20.1 GÉNÉRALITÉS	251
20.2 RESSOURCES UTILISÉES ET INTERACTIONS.....	251
20.3 LES REGISTRES CCP1CON ET CCP2CON.....	252
20.4 LE MODE « CAPTURE ».....	253
20.4.1 Principe de fonctionnement.....	253
20.4.2 Champs d'application	255
20.4.3 Remarques et limites d'utilisation.....	255
20.4.4 Mode « sleep » et astuce d'utilisation.....	257
20.5 LE MODE « COMPARE ».....	257
20.5.1 Principe de fonctionnement.....	258
20.5.2 Champs d'application	262
20.5.3 Remarques et limites d'utilisation.....	263
20.5.4 Le mode « sleep ».....	263
20.5.5 Fonctionnement non conforme.....	264
20.6 LE MODE « PWM ».....	265
20.6.1 La théorie du « PWM »	265
20.6.2 La théorie appliquée aux PICS.....	267
20.6.3 Les registres utilisés.....	273
20.6.4 Champs d'application	275
20.6.5 Remarques et limites d'utilisation.....	275
20.6.6 Le mode « sleep ».....	275
20.7 EXERCICE PRATIQUE : COMMANDE D'UN SERVOMOTEUR PAR LE PWM.....	276
20.8 EXERCICE 2 : UNE MÉTHODE PLUS ADAPTÉE	291
20.9 CONCLUSION.....	306

21. LE MODULE MSSP EN MODE SPI.....	309
21.1 INTRODUCTION SUR LE MODULE MSSP	309
21.2 LES LIAISONS SÉRIE DE TYPE SYNCHRONES.....	309
21.3 LE MODE SPI	311
21.4 LES REGISTRES UTILISÉS	312
21.5 LE MODE SPI MASTER.....	316
21.5.1 Mise en œuvre	316
21.5.2 Le registre SSPSTAT.....	319
21.5.3 Le registre SSPCON.....	320
21.5.4 Choix et chronogrammes	321
21.5.5 Vitesses de transmission.....	325
21.5.6 Initialisation du mode SPI Master.....	326
21.5.7 Le mode sleep.....	327
21.6 LE MODE SPI SLAVE.....	327
21.6.1 Mise en œuvre	327
21.6.2 Le registre SSPSTAT.....	328
21.6.3 Le registre SSPCON.....	329
21.6.4 Les autres registres concernés	330
21.6.5 Choix et chronogrammes	331
21.6.6 Vitesses de transmission.....	332
21.6.7 Initialisation du mode SPI SLAVE	332
21.6.8 Le mode sleep.....	333
21.6.9 Sécurité de la transmission	333
21.7 EXERCICE PRATIQUE : COMMUNICATION SYNCHRONES ENTRE 2 PICS.....	334
21.8 EXERCICE 2 : ALTERNATIVE DE FONCTIONNEMENT.....	348
21.9 CONCLUSIONS.....	354
22. LE BUS I²C.....	357
22.1 INTRODUCTION	357
22.1 CARACTÉRISTIQUES FONDAMENTALES	357
22.2 LES DIFFÉRENTS TYPES DE SIGNAUX	361
22.2.1 Le bit « ordinaire »	361
22.2.2 Le start-condition.....	362
22.2.3 Le stop-condition.....	362
22.2.4 L'acknowledge	363
22.2.5 Le bit read/write.....	363
22.3 La notion d'adresse.....	364
22.3 Structure d'une trame I ² C	364
22.4 ÉVÉNEMENTS SPÉCIAUX.....	369
22.4.1 La libération du bus	369
22.4.2 Le repeated start-condition	369
22.4.3 La pause	369
22.5 ARBITRAGE DU BUS	370
22.6 LES EEPROM I ² C.....	373
22.6.1 Caractéristiques générales.....	374
22.6.2 Octet de contrôle et adresse esclave	375
22.6.3 Sélection d'une adresse interne.....	378
22.6.4 Écriture d'un octet	379
22.6.5 Écriture par page	380
22.6.6 La lecture aléatoire.....	381
22.6.7 La lecture de l'adresse courante.....	383
22.6.8 La lecture séquentielle	383
22.6.9 Le cas de la 24C16.....	384
22.6.10 Conclusions.....	385
23. LE MODULE MSSP EN MODE I²C	387
23.1 INTRODUCTION	387
23.2 LE REGISTRE SSPSTAT	388
23.3 LE REGISTRE SSPCON	389
23.4 LE REGISTRE SSPADD	390

23.5 LE REGISTRE SSPCON2	393
23.6 LES COLLISIONS	394
23.7 LE MODULE MSSP EN MODE I ² C MAÎTRE	394
23.7.1 La configuration du module	394
23.7.2 La vérification de la fin des opérations	395
23.7.3 La génération du start-condition	397
23.7.4 L'envoi de l'adresse de l'esclave	397
23.7.5 Le test de l'ACK	398
23.7.6 L'écriture d'un octet	399
23.7.7 L'envoi du repeated start-condition	399
23.7.8 L'envoi de l'adresse de l'esclave	401
23.7.9 La lecture de l'octet	402
23.7.10 L'envoi du NOACK	402
23.7.11 L'envoi du stop-condition	403
23.8 L'UTILISATION DES INTERRUPTIONS	404
23.8 LE MODULE MSSP EN MODE I ² C MULTI-MAÎTRE	405
23.8.1 L'arbitrage	405
23.8.2 La prise de contrôle du bus	405
23.8.3 La détection de la perte de contrôle	406
23.9 LE MODULE MSSP EN MODE I ² C ESCLAVE 7 BITS	406
23.9.1 L'adressage et l'initialisation	407
23.9.2 la réception du start-condition	407
23.9.3 La réception de l'adresse en mode écriture	408
23.9.4 La génération du « ACK »	409
23.9.5 La réception d'un octet de donnée	409
23.9.6 La réception de l'adresse en mode lecture	410
23.9.6 L'émission d'un octet	410
23.9.7 Le mode esclave 7 bits par les interruptions	411
23.10 Le module MSSP en mode I ² C esclave 10 bits	415
23.11 SYNTHÈSE	417
23.12 EXERCICE PRATIQUE : PILOTAGE D'UNE 24C64	418
23.13 LE MODULE I ² C EN MODE SLEEP	432
23.14 CONCLUSIONS	432
23.15 ANNEXE : ERRATA SUR LE I ² C	432
23.16 BUG EN MODE I ² C ESCLAVE	433
24. LE MODULE USART EN MODE SÉRIE SYNCHRONE	435
24.1 INTRODUCTION	435
24.2 MISE EN ŒUVRE ET PROTOCOLES	435
24.3 LE REGISTRE TXSTA	437
24.4 LE REGISTRE RCSTA	438
24.5 LE REGISTRE SPBRG	439
24.6 L'INITIALISATION	440
24.7 L'ÉMISSION EN MODE MAÎTRE	441
24.8 L'ÉMISSION EN MODE ESCLAVE	443
24.9 LA RÉCEPTION EN MODE MAÎTRE	444
24.9.1 La file FIFO de RCREG	445
24.9.2 L'erreur d'overflow	446
24.10 LA RÉCEPTION EN MODE ESCLAVE	447
24.11 LE MODE SLEEP	447
24.12 DIFFÉRENCES ENTRE MSSP ET USART	447
24.13 EXERCICE PRATIQUE	448
24.14 REMARQUE SUR LE MODE SLEEP	464
24.15 CONCLUSIONS	465
25. LE MODULE USART EN MODE ASYNCHRONE	467
25.1 LE MODE SÉRIE ASYNCHRONE	467
25.1.1 Le start-bit	467
25.1.2 Les bits de donnée	468
25.1.3 La parité	468

25.1.4 Le stop-bit	469
25.1.5 LES MODES COMPATIBLES	470
25.1.6 Les erreurs de synchronisation	473
25.2 MISE EN ŒUVRE	475
25.3 LE REGISTRE TXSTA	477
25.4 LE REGISTRE RCSTA	478
25.5 LE REGISTRE SPBRG	479
25.6 L'INITIALISATION	480
25.7 EMISSION, RÉCEPTION, ET ERREUR D'OVERFLOW	480
25.8 L'ERREUR DE FRAME	483
25.9 UTILISATION DU 9 ^{ÈME} BIT COMME BIT DE PARITÉ	483
25.10 LA GESTION AUTOMATIQUE DES ADRESSES	483
25.11 LE MODE SLEEP	486
25.12 EXEMPLE DE COMMUNICATION SÉRIE ASYNCHRONE	487
25.12.1 Utilisation de BSCP.	505
25.13 CONCLUSIONS	509
26. NORME ISO7816 ET PIC16F876	511
26.1 AVANT-PROPOS	511
26.2 L'ANCÊTRE «WAFER »	511
26.3 LA SUCCESSION « PICCARD2 »	513
26.4 LA BIGCARD 1	514
26.5 LA BIGCARD 2	515
26.6 CONCLUSIONS	517
27. LE MODE SLEEP	519
27.1 GÉNÉRALITÉS	519
27.2 LES MÉCANISMES	519
27.3 LES INTERRUPTIONS CONCERNÉES	520
27.4 CONCLUSION	521
28. LE RESTE DU DATASHEET	523
28.1 INTRODUCTION	523
28.2 LE MODE LVP	523
28.3 PWRTE ET BODEN	524
28.4 LES ADRESSES « RÉSERVÉES » 0x01 À 0x03	525
28.5 L'OVERCLOCKAGE	527
28.6 CE DONT JE N'AI PAS PARLÉ	528
ANNEXE1 : QUESTIONS FRÉQUEMMENT POSÉES (F.A.Q.)	529
A1.1 JE N'ARRIVE PAS À UTILISER MON PORTA	529
A1.2 LE CHOIX DU PROGRAMMATEUR	529
CONTRIBUTION SUR BASE VOLONTAIRE	531
B. UTILISATION DU PRÉSENT DOCUMENT	533

1. Introduction

Tout d'abord, un grand remerciement à tous ceux qui ont permis que cette aventure existe. Je parle ici de toutes les personnes qui m'ont envoyé leurs commentaires et corrections, les différents hébergeurs de la première partie de ce cours, ainsi que tous ceux qui m'ont envoyé remerciements et encouragements. Je remercie d'avance tous ceux qui feront de même pour cette seconde partie.

Merci également à mon épouse pour sa patience durant toutes ces soirées passées « planté derrière mon ordi ».

Cette seconde partie s'adresse aux personnes qui ont déjà lu et compris la première partie, dédiée à la programmation du 16F84. Donc, je ne reviendrai pas sur les explications de base concernant la programmation, et je n'expliquerai pas non plus les différentes fonctions avec autant de détail. Cet ouvrage est donc un complément, une évolution, et non un ouvrage destiné à être utilisé de manière complètement indépendante.

Le document indispensable pour aborder la programmation du processeur 16F87x est le datasheet 16F87x disponible sur le site <http://www.microchip.com>. Afin que tout le monde dispose du même document que l'auteur, j'ai intégré dans le répertoire « fichiers », la version du datasheet qui a servi pour élaborer ce cours.

Ceux qui veulent comprendre tous les détails du fonctionnement des PICs concernés peuvent également charger le « Pic Micro Mid-Range MCU Family Reference Manual », disponible à la même adresse. Attention, l'impression de cet ouvrage, disponible d'ailleurs par chapitres séparés, requiert plusieurs centaines de feuilles.

Cet ouvrage est plus technique que le précédent. Cette démarche m'a été imposée, d'une part par l'étendue des possibilités, et d'autre part par la nécessité d'une certaine part de théorie qui, seule, permettra de sortir le lecteur de toutes les situations non abordées de façon concrète.

En effet, ces composants possèdent une multitude de fonctions, elles-mêmes parfois scindées en plusieurs modes de fonctionnement. Décrire un exemple détaillé pour chacune de ses fonctions aurait nécessité un ouvrage trop conséquent pour que je puisse en venir à bout dans des délais raisonnables. Vous remarquerez cependant que je n'ai pas été avare d'exercices pratiques.

J'ai donc conservé le maximum d'exemples, mais j'ai opté principalement pour une démarche qui vise à vous fournir tous les outils et toutes les explications qui vous permettront d'utiliser toutes les fonctions de cette série dans vos applications particulières.

Ce faisant, j'ai dû aborder beaucoup de théorie, ce qui rend cet ouvrage un peu plus dur à lire que le précédent. Ce n'est pas une démarche élitiste, au contraire, mais une tentative pour vous ouvrir le maximum de portes pour le futur. De plus, vous êtes sensés, lors de la lecture de cet ouvrage, maîtriser parfaitement le 16F84, on ne peut donc plus parler d'ouvrage pour débutants.

J'espère qu'en procédant de la sorte, je ne me suis pas montré trop rébarbatif.

Je suis toujours à l'écoute de tous, je réponds toujours au courrier reçu, et si la demande se fait sentir concernant certains chapitres, il me sera toujours possible d'ajouter explications complémentaires et exemples supplémentaires.

N'hésitez donc pas à me faire part de vos remarques et suggestions, et jetez de temps en temps un œil sur mon site (<http://www.abcelectronique.com/bigonoff> ou www.bigonoff.org) pour voir si une nouvelle révision n'est pas disponible.

A tout ceux qui m'ont demandé ou qui se demandent pourquoi j'ai mis cet ouvrage gratuitement à la disposition de tous, je répondrai ceci :

« En ces temps où tout se marchande, où la mondialisation du commerce s'effectue au détriment du bien-être de l'Homme et de l'Humanité, il est primordial de se dresser contre cette mentalité du tout marchand.

Je ne suis pas le précurseur de cette démarche dans le domaine de la technique et de l'informatique. Bien d'autres ont compris ceci avant moi, comme par exemple les créateurs de logiciels freeware. D'autres suivront ces exemples, afin que le monde en général, et Internet en particulier, devienne ce qu'il aurait toujours dû être : un lieu d'échange de la connaissance et un partage des dons et des richesses de chacun .

Internet est un formidable outil de communication qui peut nous aider à atteindre ce but. Veillons attentivement à ce qu'il ne devienne pas un gigantesque lieu de e-business récupéré par les multinationales. »

Je vais cependant peut-être chercher un éditeur qui me permettra de distribuer en plus ces différents ouvrages par la voie classique. Ceci permettrait au plus grand nombre d'avoir accès aux informations, et, de plus, pourrait se révéler rentable pour l'utilisateur, le prix de l'impression d'un tel ouvrage sur une imprimante à jet d'encre pouvant se révéler aussi onéreux que l'acquisition d'un livre. Mais le cours restera à disposition sur internet aussi longtemps que j'en aurai la possibilité matérielle et financière.

La demande de contribution sur base volontaire n'est pas une entorse à cette philosophie. Le cours reste gratuit, m'aide qui le veut, et comme il le veut.

J'ai construit ce livre en pensant à son format papier. Il est prévu pour être imprimé en recto-verso. Comme il est amené à être corrigé, et afin de ne pas vous imposer la réimpression complète en cas de modifications, j'ai laissé un peu de place libre entre chaque chapitre. Ainsi, une insertion ne nécessitera pas la renumérotation des pages.

Je vous souhaite beaucoup de plaisir à la lecture de ce petit ouvrage sans prétention, et je vous le conseille : expérimentez, expérimentez, et expérimentez encore !

... Et vive l'Internet libre !

BIGONOFF

2. Les caractéristiques des 16F87x

2.1 Caractéristiques générales de la famille 16F87x

Le 16F876 et le 16F877 (et d'autres) font partie de la sous-famille des 16F87x. Cette branche fait partie intégrante de la grande famille des PICs Mid-Range, au même titre que le 16F84 dont je parle dans le précédent ouvrage. On peut considérer que le 16F84 constitue le circuit d'entrée de gamme de cette famille, alors que le 16F877 représente la couche supérieure. De nouveaux circuits ne devraient probablement pas tarder à améliorer encore les performances.

De ce fait, beaucoup de similitudes sont à prévoir entre ces 2 composants. Nous verrons dans le chapitre suivant quels sont ces points communs.

La famille 16F87x comprend toute une série de composants, voici les différents types existant au moment de l'écriture de cet ouvrage. Notez que les composants sont en constante évolution. Il vous appartiendra donc de vous tenir au courant de ces évolutions.

PIC	FLASH	RAM	EEPROM	I/O	A/D	Port //	Port série
16F870	2K	128	64	22	5	NON	USART
16F871	2K	128	64	33	8	PSP	USART
16F872	2K	128	64	22	5	NON	MSSP
16F873	4K	192	128	22	5	NON	USART/MSSP
16F874	4K	192	128	33	8	PSP	USART/MSSP
16F876	8K	368	256	22	5	NON	USART/MSSP
16F877	8K	368	256	33	8	PSP	USART/MSSP

Tous ces composants sont identiques, aux exceptions citées dans le tableau précédent. Les différences fondamentales entre ces PICs sont donc les quantités de mémoires disponibles, le nombre d'entrées/sorties, le nombre de convertisseurs de type « analogique/digital » (dire « analogique/numérique », et le nombre et le type des ports intégrés.

A l'heure actuelle, ces composants existent dans divers types de boîtiers. Ces types sont représentés page 1 et 2 du datasheet. Les boîtiers 40 broches sont utilisés par les composants qui disposent d'un port //, comme le 16F877. Le boîtier qui nous intéresse plus particulièrement est celui dont le brochage est disponible en haut et à gauche de la page 2. Ce boîtier est disponible en version standard (PDIP 28 broches) ou en version « composant de surface », ou « CMS » (SOIC).

Comme pour le 16F84, le numéro peut être suivi d'un « A », et d'un « -xx » qui donne la fréquence d'horloge maximum du composant. A l'heure où j'écris cet ouvrage, la version la plus courante est la version -20. Donc, la fréquence que nous utiliserons en standard dans cet ouvrage sera de 20MHz. De même, les exercices seront réalisés sur un PIC16F876, moins cher que le 16F877. Nous n'utiliserons ce dernier composant que lorsque ses spécificités seront indispensables à la réalisation de l'exercice.

Nous nous intéresserons donc dans cet ouvrage principalement au 16F876, qui est le plus utilisé à l'heure actuelle. La transposition vers un autre type ne demande qu'un minimum d'adaptation. Nous parlerons également du 16F877 lorsque nous aborderons le port parallèle. Ainsi, nous aurons vu l'intégralité des fonctions de cette famille.

2.2 Organisation de la RAM

Nous voyons donc que la mémoire RAM disponible du 16F876 est de 368 octets. Elle est répartie de la manière suivante et donnée page 13 du datasheet :

- 1) 80 octets en banque 0, adresses 0x20 à 0x6F
- 2) 80 octets en banque 1, adresses 0xA0 à 0XEF
- 3) 96 octets en banque 2, adresses 0x110 à 0x16F
- 4) 96 octets en banque 3, adresses 0x190 à 0x1EF
- 5) 16 octets communs aux 4 banques, soit 0x70 à 0x7F = 0xF0 à 0xFF = 0x170 à 0x17F = 0x1F0 à 0x1FF

Que signifient ces octets communs ? Tout simplement que si vous accédez au registre (adresse mémoire RAM) 0x70 ou au registre 0XF0, et bien vous accédez en réalité au même emplacement. Ceci à l'avantage de permettre d'utiliser ces emplacements sans devoir connaître l'état de RP0, RPI, et IRP.

2.3 Structure du 16F876

Allons ensemble regarder la structure interne du 16F876, schématisée figure 1.1 page 5 du datasheet.

Que constatons-nous au premier abord ?

En premier lieu, les largeurs de bus internes sont les mêmes que pour le 16F84, c'est à dire que nous devrons faire face aux mêmes contraintes pour les accès aux différentes banques et registres.

Ensuite, nous constatons la présence de plus de ports, ce qui augmente d'autant le nombre d'entrées/sorties disponibles.

Viennent ensuite les timers, au nombre de 3 au lieu d'un seul pour le 16F84. A côté de ces timers on remarquera la présence d'un convertisseur analogique de 10 bits.

Au vu de ce schéma-bloc et des indications précédentes, on peut donc dire, pour dégrossir le sujet de manière approximative, qu'un 16F876, c'est un 16F84 doté en supplément :

- De plus de mémoire RAM (répartie sur 4 banques), Flash, et eeprom
- De plus de ports d'entrée/sortie
- De plus de timers
- De nouvelles fonctionnalités, comme les gestions de ports « série »
- D'un convertisseur A/D (analogique/numérique) à plusieurs canaux d'une résolution de 10 bits.

Nous voyons donc que l'étude du 16F876 peut se résumer à l'apprentissage des différences par rapport au 16F84. Ces différences introduisent cependant une part non négligeable de théorie, ce qui m'a convaincu d'écrire cette seconde partie.

Il faut dire qu'écrire un livre de plus de 500 pages rien que sur les différences avec la précédente version, à propos de laquelle l'ouvrage comportait 200 pages, indique que les dites différences justifiaient qu'on s'y attarde.

Il existe de plus des fonctions qui nécessitent une étude plus poussée (debuggage sur circuit, techniques du bootloader), mais j'aborderai ceci dans le livre suivant (3^{ème} partie : Les secrets des 16F87x). En effet, afin de limiter la taille de cet ouvrage, et d'accélérer sa sortie, j'ai scindé la version initialement prévue en plusieurs volumes.

En bonne logique, commençons donc par voir ce qui caractérise le 16F87x

Notes :

3. Les particularités de la programmation du 16F87x

3.1 La directive « CONFIG »

Tout comme nous l'avons vu dans la première partie, cette directive détermine le fonctionnement du PIC. La valeur est inscrite au moment de la programmation dans un registre spécial, situé en mémoire programme à l'adresse 0x2007, et ne peut plus être modifié en cours d'exécution du programme.

Notez que puisque la mémoire programme des PICs vierges contient des « 1 », les niveaux actifs de ces bits sera le niveau « 0 ».

Ce registre de 14 bits (puisque en mémoire programme) dispose d'une organisation différente par rapport à celui du 16F84. Voici donc ses fonctions, reprises **page 122 du datasheet** :

- **CP1/CP0** : bits 13/12 et 5/4 : Détermine quelle zone du 16F876 sera protégée contre la lecture. Vous pouvez donc choisir de protéger la totalité du PIC, ou seulement une partie. Les différentes zones pouvant être protégées sont les suivantes :

CP1 CP0

1	1	Aucune protection (CP OFF)
1	0	Protection de la zone 0x1F00 à 0x1FFF (CP_UPPER_256)
0	1	Protection de la zone 0x1000 à 0x1FFF (CP_HALF)
0	0	Protection de l'intégralité de la mémoire (CP_ALL)

- **DEBUG** : bit 11 : Debuggage sur circuit. Permet de dédicacer RB7 et RB6 à la communication avec un debugger.
 - 1 : RB6 et RB7 sont des I/O ordinaires (**DEBUG_OFF**)
 - 0 : RB6 et RB7 sont utilisés pour le debuggage sur circuit (**DEBUG_ON**)
- Bit 10 : non utilisé
- **WRT** : bit 9 : Autorisation d'écriture en flash
 - 1 : Le programme peut écrire dans les zones **non protégées par les bits CP1/CP0** (**WRT_ENABLE_ON**)
 - 0 : Le programme ne peut pas écrire en mémoire flash (**WRT_ENABLE_OFF**)
- **CPD** : bit 8 : Protection en lecture de la mémoire eeprom.
 - 1 : mémoire eeprom non protégée (**CPD_OFF**)
 - 0 : mémoire eeprom protégée (**CPD_ON**)
- **LVP** : bit 7 : Utilisation de la pin RB3/PGM comme broche de programmation
 - 1 : La pin RB3 permet la programmation du circuit sous tension de 5V (**LVP_ON**)
 - 0 : La pin RB3 est utilisée comme I/O standard (**LVP_OFF**)
- **BODEN** : bit 6 : provoque le reset du PIC en cas de chute de tension (surveillance de la tension d'alimentation)
 - 1 : En service (**BODEN_ON**)

- 0 : hors service (**BODEN_OFF**)
- **PWRTE** : bit 3 : Délai de démarrage à la mise en service. Attention, est automatiquement mis en service si le bit BODEN est positionné.
 - 1 : délai hors service (sauf si **BODEN = 1**) (**PWRTE_OFF**)
 - 0 : délai en service (**PWRTE_ON**)
- **WDTE** : bit 2 : Watchdog timer
 - 1 : WDT en service (**WDT_ON**)
 - 0 : WDT hors service (**WDT_OFF**)
- **FOSC1/FOSC0** : bits 1/0 : sélection du type d'oscillateur
 - 11 : Oscillateur de type RC (**RC_OSC**)
 - 10 : Oscillateur haute vitesse (**HS_OSC**)
 - 01 : Oscillateur basse vitesse (**XT_OSC**)
 - 00 : Oscillateur faible consommation (**LP_OSC**)

N'oubliez pas d'utiliser ces valeurs dans la directive **CONFIG**. Les différentes valeurs, prédéfinies dans le fichier « **P16F876.INC** » doivent être séparées par des directives « **&** ».

Voici un exemple d'utilisation :

```
_CONFIG _CP_OFF & _DEBUG_OFF & _WRT_ENABLE_OFF & _CPD_OFF &
_LVP_OFF & _BODEN_OFF & _PWRTE_ON & _WDT_OFF & _HS_OSC
```

Je fournis dans les fichiers joints à ce cours, une maquette pour le 16F876 et une pour le 16F877, dans lesquelles le corps de programme et les explications des directives ont déjà été élaborés par mes soins. Ces fichiers « **m16f876.asm** » et « **m16f877.asm** » vous épargneront sans aucun doute bien du travail peu valorisant.

3.2 Utilisation de la mémoire RAM

Sur ce processeur, nous avons donc **4 banques de mémoire RAM**. Nous voyons que les adresses s'échelonnent entre **0x00 et 0x1FF**.

Si vous convertissez 0x1FF en binaire, à l'aide de la calculatrice de Windows, vous constaterez qu'il faut **9 bits** pour coder cette valeur. La largeur du **bus de données** étant **de 8 bits**, et la largeur du **bus d'adressage direct** de **7** (voir la première partie), il va nous falloir trouver des bits supplémentaires.

3.2.1 L'adressage direct

Pour ce type d'adressage, il nous **manque** donc **2 bits**. Tout comme dans le cas du 16F84, nous allons trouver ces 2 bits dans le registre **STATUS**. La combinaison des bits **RP0** et **RP1** de ce registre, permet donc d'accéder en adressage direct à l'intégralité de la mémoire RAM. L'adresse finale est donc composée de **RP1/RP0 comme bits 8 et 7**, complétés des 7 bits de l'adresse directe utilisée dans l'expression.

Les 4 possibilités concernant **RP1/RP0** donnent donc :

- 00 : Accès à la RAM 0x00 à 0x7F
- 01 : Accès à la RAM 0x80 à 0xFF
- 10 : Accès à la RAM 0x100 à 0x17F
- 11 : Accès à la RAM 0x180 à 0x1FF

Notez que la RAM située de 0x70 à 0x7F est accessible quel que soit l'état de **RP0** et **RP1**. En effet, les zones correspondantes dans les autres banques sont en fait des images de cette zone

Voici des emplacements de mémoire RAM. Pour rappel, les « () » dans les commentaires signifient « le contenu de » :

```
variable1 EQU 0x25 ; adresse de la variable 1
variable2 EQU 0x7E ; adresse de la variable 2
variable3 EQU 0xA1 ; adresse de la variable 3

movlw 0x50 ; charger 0x50 dans W
movwf variable2 ; dans la case mémoire 0x7E, car indépendant de
; RPx
movlw 0x30 ; charger 0x30 dans W
bcf STATUS , RP0 ; passage en banque 0
bcf STATUS , RP1
movwf variable1 ; placer dans la case mémoire 0x25
movwf variable3 ; placer dans 0x21, car seuls 7 bits sont pris en
; compte. Piège.
bsf STATUS , RP1 ; passer en banque 2
movf variable2 , W ; (0x17E) dans W, donc (0x7E), donc 0x50
movwf variable1 ; 0x50 dans la case mémoire 0x125
bsf STATUS , RP0 ; passage en banque 3
movwf variable1 ; 0x50 dans la case mémoire 0x1A5
```

Vous pouvez facilement voir les correspondances des adresses en utilisant le **tableau 2.3 page 13 du datasheet**, ou avec la calculatrice de Windows. Si vous ne comprenez pas bien ou que vous doutez, procédez à la méthode suivante pour obtenir l'adresse concernée :

- 1) Convertissez l'adresse en binaire avec la calculatrice Windows
- 2) Entrez en binaire les 2 bits **RP0** et **RP1**
- 3) Complétez avec les 7 bits à gauche de l'adresse binaire.

Par exemple, pour la dernière ligne du précédent exemple :

- 1) **RP0/RP1** = 11
- 2) 25 en hexadécimal donne 100101 en binaire (attention, il n'y a que 6 chiffres, je complète donc par un 0 à gauche : 0100101
- 3) Ceci donne B'110100101', soit 0x1A5

3.2.2 L'adressage indirect

L'adressage indirect utilise le registre **FSR/INDF**. Hors, ce registre à une largeur de 8 bits. Donc, tel quel, il lui est impossible d'adresser plus de 2 banques. Il va donc falloir trouver une fois de plus un bit supplémentaire.

Ce bit est le bit **IRP** du registre **STATUS**. Ce bit est donc utilisé comme bit de poids fort (bit 8) complété par les 8 bits contenus dans le registre **FSR**.

En fonction de la valeur de **IRP**, nous accéderons donc :

IRP = 0 : Banques 0 et 1, soit registres de 0x00 à 0xFF

IRP = 1 : Banques 2 et 3, soit registres de 0x100 à 0x1FF

Donc, faites attention : les bits **RP0** et **RP1** n'influencent en aucune façon l'adressage indirect. De même, **IRP** n'influence en aucune façon l'adressage direct.

Voici un exemple d'utilisation de l'adressage indirect. On présume que cet exemple est la suite du précédent :

```
bcf    STATUS,IRP    ; pointer banques 0 et 1 indirect
movlw  variable1      ; charger 0x25, donc adresse de variable1
movwf  FSR            ; mettre 0x25 dans le pointeur
movf   INDF,w         ; charger (0x25), donc 0x30
bsf    STATUS,IRP    ; pointer banque 2 et 3 indirect
movwf  INDF           ; mettre 0x30 dans le registre 0x125
movlw  variable3      ; charger adresse variable3
movwf  FSR            ; dans pointeur
bcf    STATUS,RP0     ; pointer banque0 en adressage direct
bcf    STATUS,RP1
movf   variable3,w    ; charger (0x21) car 0xA1 en 7 bits , donc 0x30
movwf  INDF           ; placer 0x30 en 0x1A1. En effet, 8 bits de
                      ; l'adresse 0xA1 + bit8 à 1 (IRP)
```

Pour résumer :

- Vous devez toujours vérifier **RP0** et **RP1** avant toute utilisation de l'adressage direct.
- Vous devez toujours vérifier **IRP** avant toute utilisation de l'adressage indirect.

Sachez déjà que, suivant mes observations, plus de la moitié des erreurs de programmation persistantes sont dues à des erreurs de sélection de banques.

3.3 L'utilisation du registre PCLATH

Nous avons déjà vu ce registre dans l'étude du 16F84 : en effet, lors d'une opération mathématique sur le registre **PCL** (dans le cadre de l'utilisation d'un tableau), le résultat donnait une valeur sur 8 bits.

Notre 16F84 permettant un adressage de plus de 8 bits, les bits manquants étaient également empruntés à **PCLATH**.

3.3.1 PCLATH et les calculs d'adresse

Cette limitation existe ici aussi de la même manière. Donc, nous devrons donc utiliser notre **PCLATH** en suivant les recommandations suivantes :

- En cas d'opération sur **PCL**, la valeur sur 8 bits obtenue est complétée par **PCLATH** pour former l'adresse de destination effective. Supposons donc que nous fassions une addition de **w** par rapport à **PCL**. Où va pointer alors le **pointeur de programme (PC)** ?

Voici le contenu de PC après l'addition :

b7 à b0 seront en fait le contenu effectif de **PCL**
b12 à b8 seront les bits b4 à b0 de **PCLATH**.

Exemple d'opération qui s'applique dans ce cas :

```
movlw B'10010' ; charger valeur
movwf PCLATH   ; initialiser PCLATH
movlw 0x22     ; charger valeur
addwf PCL, f   ; sauter plus loin
```

PCLATH (5 bits)					PCL (8 bits)							
B4	B3	B2	B1	B0	B7	B6	B5	B4	B3	B2	B1	B0
PC(13 bits) = adresse de saut												
B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0

Ce fonctionnement est strictement identique à celui expliqué dans le cadre du 16F84.

Si on prend comme exemple concret le résultat suivant :

- **PCL** vaut B'11011010' après l'addition
- **PCLATH** a été initialisé à B'00010010' par l'utilisateur. Notez que b7 à b5 sont inutilisés.
- Le saut final s'effectuera donc à l'adresse B'1001011011010' après l'addition, soit 0x12DA.

PCLATH (5 bits)					PCL (8 bits)							
1	0	0	1	0	1	1	0	1	1	0	1	0
PC(13 bits) = adresse de saut												
1	0	0	1	0	1	1	0	1	1	0	1	0

3.3.2 PCLATH et les sauts directs

Et oui, avec ce processeur survient un autre problème, problème que nous n'avions pas rencontré cette fois avec le 16F84.

Si vous examinez le jeu d'instructions du 16F876 dans le détail, vous allez vous apercevoir qu'il est strictement identique à celui du 16F84, et, de ce fait, présente les mêmes limitations.

Une de celle-ci, la plus gênante, est le nombre de bits qui accompagne les instructions de saut, comme le « goto ». Rappelez-vous que les instructions de saut sont de la forme : Op-code + 11 bits de destination.

Or, les sauts sur 11 bits ne permettent « que » des sauts à l'intérieur d'une page de 2K Mots. Notre processeur utilise un maximum de 8K mots de programme, et nécessite donc 2 bits supplémentaires pour accéder à l'intégralité de la mémoire programme. Le schéma-bloc donnant bien une largeur de bus d'adressage de 13 bits.

Ces bits, nous allons donc aller les chercher également dans le registre PCLATH. Comme il ne nous manque que 2 bits, nous n'utiliserons donc que les 2 bits b4 et b3 de PCLATH.

Pour résumer, le saut s'effectuera à l'adresse donnée en argument complétée en poids fort par les 2 bits b4 et b3 de PCLATH. Les bits 2 à 0 de PCLATH sont inutilisés ici.

PCLATH		ARGUMENT DU SAUT (11 bits)										
B4	B3	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
PC(13 bits) = adresse de saut												
B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0

Petit exemple pratique : soit les instructions suivantes :

```
movlw    0x13      ; charger 0x13, soit B'10011', donc b4=1, b3=0
movwf    PCLATH     ; dans PCLATH
goto     0x112      ; sauter théoriquement en 0x112 (B'00100010010')
```

Voici ce que ça donne :

PCLATH		ARGUMENT DU SAUT (11 bits)										
1	0	0	0	1	0	0	0	1	0	0	1	0
PC(13 bits) = adresse de saut												
1	0	0	0	1	0	0	0	1	0	0	1	0

Donc, vous constaterez que le programme saute en réalité à l'adresse 0x1112, et non à l'adresse 0x112.

Mais ne vous inquiétez pas trop : tout d'abord nous allons voir tout ça avec un exemple pratique, et ensuite, les sauts dans les espaces supérieurs à 2K ne devraient pas vous perturber avant un certain temps.

N'oubliez pas en effet que vous ne rencontrerez ce problème que si votre programme dépasse les 2K mots de code. Donc, il faudra que vous ayez écrit plus de 2000 lignes de code dans votre programme.

Nul doute que quand vous en serez là, vous maîtriserez parfaitement ce type de limitation.

De plus, je vais vous venir en aide, en vous expliquant et en créant des macros pour vous faciliter le traitement de ce problème le jour où vous le rencontrerez.

Notez qu'il est rare d'écrire un ligne du type

```
goto      0x112
```

car, en général, vous utiliserez plutôt un saut vers une étiquette, du style :

```
goto  etiquette
```

Dans ce cas, vous devrez raisonner de façon inverse. Il vous faudra savoir dans quelle page de 2K mots se trouve « etiquette » et ensuite vous positionnerez **PCLATH** en conséquence.

Rassurez-vous cependant, MPASM ne manquera pas de vous avertir, par un warning, qu'il y a un risque à ce niveau.

Mais, pour être certain que vous compreniez bien tout, nous allons mettre tout ceci en pratique dans le chapitre suivant.

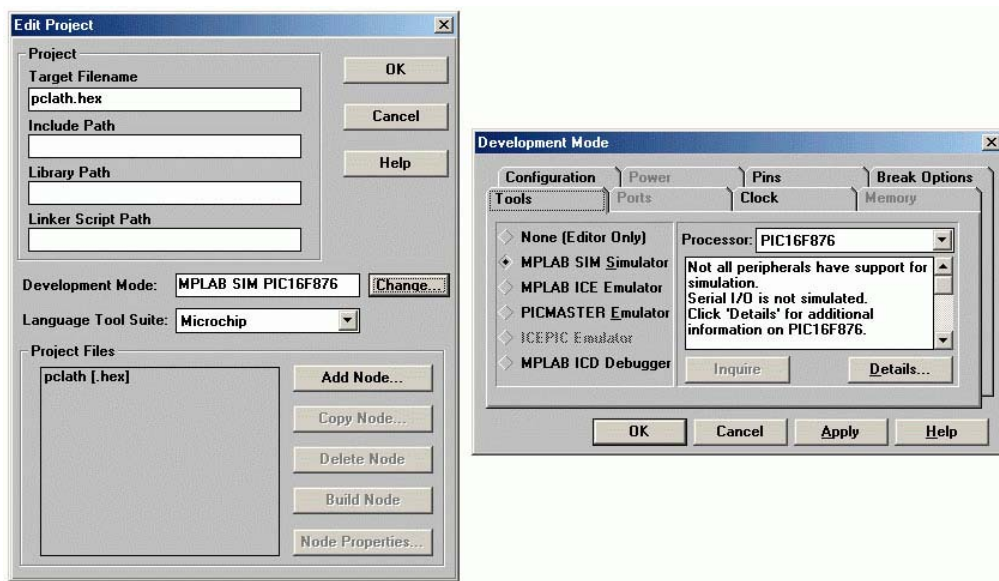
NOTES : ...

4. Sauts de pages et premier projet sur MPLAB

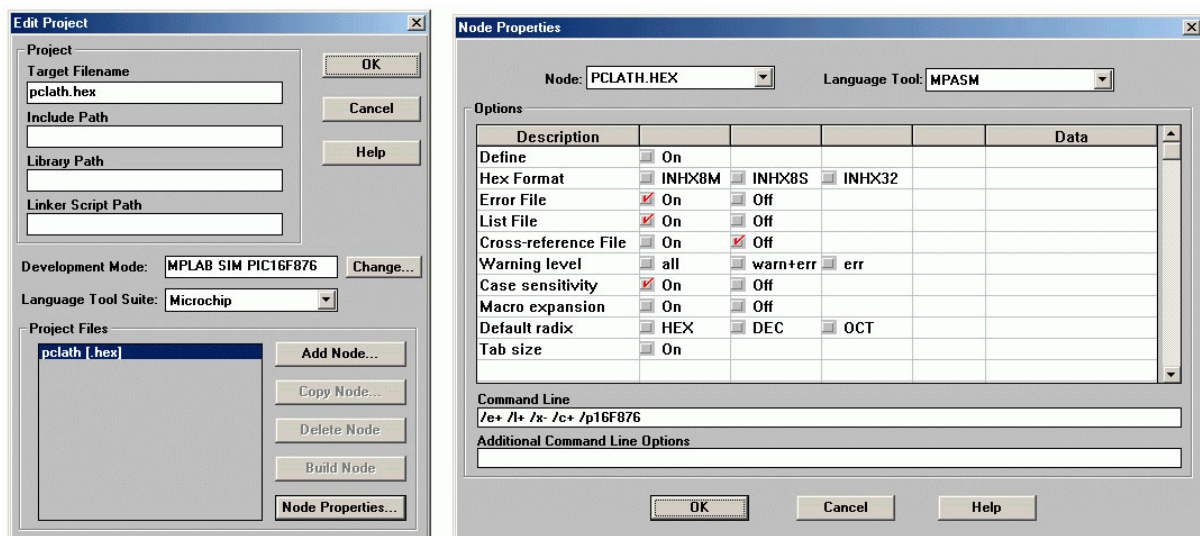
Pour clarifier la situation, nous allons donc mettre ceci en pratique. Commencez donc par faire une copie de votre fichier « m16f876.asm » et renommez cette copie en « pclath.asm ».

Lancez MPLAB et créez un nouveau projet « pclath.pjt ».

Je n'entrerai plus dans les détails, vu que j'ai tout expliqué dans la première partie du cours. Dans la fenêtre « edit project » qui s'ouvre alors, sélectionnez « change » à côté de la fenêtre « development mode », puis choisissez la mise en service du simulateur pour le 16F876. Ceci est donné pour l'ancienne version 5.x de MPLAB. Pour la 6.x, voyez le cours-part1



Cliquez ensuite sur la ligne « pclath [.hex] », le bouton « Node Properties » devient actif. Cliquez dessus et configurez la fenêtre qui s'ouvre de manière identique à celle de la figure suivante :



Cliquez « OK » dans la fenêtre « Node Properties », puis revenez à la fenêtre « Edit project ». Créez un nouveau nœud avec <Add Node> et sélectionnez votre fichier « pclath.asm ». Cliquez ensuite <OK> dans la fenêtre « Edit project ».

Dans le menu « file », ouvrez enfin votre fichier « pclath.asm ». Vous êtes enfin prêts à travailler sur votre premier programme en 16F876.

Si vous n'avez pas tout suivi, retournez jeter un œil dans le cours sur le 16F84.

4.1 Structure et utilisation du fichier « maquette »

Commencez donc, comme toujours, par remplir votre cadre de commentaires, pour expliquer ce que contiendra votre nouveau projet. Tout comme dans la première partie du cours, j'insiste de nouveau pour que vous documentiez au maximum vos programmes. C'est le seul et unique moyen de vous permettre une maintenance aisée de vos réalisations dans le temps. Ce qui vous semble évident aujourd'hui ne le sera peut-être plus demain. Sachez que moi-même je documente mes programmes exactement de la façon que je décris dans ce cours, car ce qui est bon pour vous l'est forcément pour moi aussi.

J'ouvre ici une parenthèse, pour me fâcher un peu (une fois n'est pas coutume, vous ne m'y reprendrez plus).

Malgré tous ces avertissements, je reçois régulièrement des programmes exécutés par certains d'entre-vous, et vierges de tout commentaire. D'une part, je ne lirai plus les programmes sans commentaire, et d'autre part, si ces personnes me les envoient, c'est qu'ils ne fonctionnent pas. Comment alors admettre que ces personnes n'ont pas besoin de conseils pour réaliser leur programme, mais en ont besoin pour les debugger lorsqu'ils ne fonctionnent pas ?

De même, je reçois des programmes de personnes qui ont décidé de n'utiliser aucun symbole. Essayez donc de lire des programmes du style :

```
bsf    03,7
movlw  0x3C
movwf  04
movf   0,0
```

Et bien, moi, j'y renonce définitivement. Et dire que leurs auteurs s'étonnent qu'ils sont difficiles à debugger. C'est quand même plus clair comme ceci, non ? (c'est le même programme) :

```
bsf    STATUS,IRP      ; pointer sur banques 2 et 3
movlw  mavariable      ; charger adresse de ma variable
movwf  FSR              ; dans pointeur indirect
movf   INDF,w          ; charger mavariable dans w
```

Autrement dit, si vous vous obstinez à utiliser la première méthode (qui vous fera très vite perdre le temps gagné), alors c'est que vous avez décidé de vous passer de mes conseils. Gardez donc dans ce cas vos problèmes de debuggage pour vous.

Ben oui, cette fois je râle un peu, mais il faut me comprendre, non ? J'estime donc que si vous poursuivez, c'est que vous êtes pleins de bonne volonté, et que vous comprenez que c'est dans votre propre intérêt d'utiliser les commentaires et adressages symboliques.

Je referme la parenthèse.

Voici donc un exemple de zone de commentaires pour cet exercice :

```
;*****
; Exercice sur les sauts inter-pages pour montrer l'utilisation de *
; PCLATH. *
; *
;*****
; *
; NOM: PCLATH *
; Date: 04/07/2001 *
; *
; Version: 1.0 *
; *
; Circuit: aucun *
; Auteur: Bigonoff *
; *
; *
;*****
; *
; Fichier requis: P16F876.inc *
; *
; *
;*****
```

Viennent ensuite, comme toujours, la déclaration de processeur destinée à **MPASM**, et la référence au fichier d'inclusion, qui devra être présent dans votre répertoire des fichiers.

```
LIST p=16F876 ; Définition de processeur
#include <p16F876.inc> ; fichier include
```

Ensuite, vous trouvez la fameuse directive **CONFIG** qui définit le fonctionnement de base du processeur. Rien de spécial ici, vu que ce n'est qu'un exercice statique:

```
CONFIG _CP_OFF & _DEBUG_OFF & _WRT_ENABLE_OFF & _CPD_OFF & _LVP_OFF &
_BODEN_OFF & _PWRTE_ON & _WDT_OFF & _HS_OSC
```

Plus bas se trouvent les **assignations système**. Ceci dans le but de vous éviter de chercher dans le programme après les registres concernés. Il suffit de modifier ici les valeurs de certains registres. Ces valeurs seront chargées lors de l'initialisation par la routine « init ».

```
;*****
; ASSIGNATIONS SYSTEME *
;*****
; REGISTRE OPTION_REG (configuration)
; -----
OPTIONVAL EQU B'00000000'
; RBPUR b7 : 1= Résistance rappel +5V hors service
; INTEDG b6 : 1= Interrupt sur flanc montant de RB0
; 0= Interrupt sur flanc descend. de RB0
; TOCS b5 : 1= source clock = transition sur RA4
```

```

;
; TOSE      b4 :      0= horloge interne
;              1= Sélection flanc montant RA4 (si B5=1)
;
; PSA       b3 :      0= Sélection flanc descendant RA4
;              1= Assignation prédiviseur sur Watchdog
;
; PS2/PS0   b2/b0 :   0= Assignation prédiviseur sur Tmr0
;                      valeur du prédiviseur
;                      000 = 1/1 (watchdog) ou 1/2 (tmr0)
;                      001 = 1/2              1/4
;                      010 = 1/4              1/8
;                      011 = 1/8              1/16
;                      100 = 1/16             1/32
;                      101 = 1/32             1/64
;                      110 = 1/64             1/128
;                      111 = 1/128            1/256
;

; REGISTRE INTCON (contrôle interruptions standard)
; -----
INTCONVAL EQU B'00000000'
; GIE  b7 : masque autorisation générale interrupt
;          ne pas mettre ce bit à 1 ici
;          sera mis en temps utile
; PEIE b6 : masque autorisation générale périphériques
; T0IE b5 : masque interruption tmr0
; INTE b4 : masque interruption RB0/Int
; RBIE b3 : masque interruption RB4/RB7
; T0IF b2 : flag tmr0
; INTF b1 : flag RB0/Int
; RBIF b0 : flag interruption RB4/RB7

; REGISTRE PIE1 (contrôle interruptions périphériques)
; -----
PIE1VAL EQU B'00000000'
; PSPIE b7 : Toujours 0 sur PIC 16F786
; ADIE  b6 : masque interrupt convertisseur A/D
; RCIE  b5 : masque interrupt réception USART
; TXIE  b4 : masque interrupt transmission USART
; SSPIE b3 : masque interrupt port série synchrone
; CCP1IE b2 : masque interrupt CCP1
; TMR2IE b1 : masque interrupt TMR2 = PR2
; TMR1IE b0 : masque interrupt débordement tmr1

; REGISTRE PIE2 (contrôle interruptions particulières)
; -----
PIE2VAL EQU B'00000000'
; UNUSED b7 : inutilisé, laisser à 0
; RESERVED b6 : réservé, laisser à 0
; UNUSED b5 : inutilisé, laisser à 0
; EEIE  b4 : masque interrupt écriture EEPROM
; BCLIE b3 : masque interrupt collision bus
; UNUSED b2 : inutilisé, laisser à 0
; UNUSED b1 : inutilisé, laisser à 0
; CCP2IE b0 : masque interrupt CCP2

```

Ne vous occupez pas de ces registres ici. Nous verrons ceux que vous ne connaissez pas encore plus tard. Vous trouvez également deux zones prévues pour placer vos propres définitions et assignations, le tout avec chaque fois un exemple.

Il va de soi que lorsque vous maîtriserez parfaitement les PICs, vous pourrez vous passer des assignations dont vous n'avez pas usage. Mais ceci vous permet d'être certains de ne rien oublier.

Maintenant, vous vous trouvez dans la zone des macros, où quelques macros existent déjà. Par exemple, les macros de passage de banques. Nous n'aurons pas besoin des autres dans le cadre de cet exercice

```
; *****
;
;                               MACRO
; *****

BANK0    macro                ; passer en banque0
    bcf    STATUS,RP0
    bcf    STATUS,RP1
endm

BANK1    macro                ; passer en banque1
    bsf    STATUS,RP0
    bcf    STATUS,RP1
endm

BANK2    macro                ; passer en banque2
    bcf    STATUS,RP0
    bsf    STATUS,RP1
endm

BANK3    macro                ; passer en banque3
    bsf    STATUS,RP0
    bsf    STATUS,RP1
endm
```

Ces macros permettent de sélectionner une banque particulière en positionnant les 2 bits concernés. Il est évident que si vous savez dans quelle banque vous vous trouvez, **il n'est pas toujours nécessaire de repositionner les 2 bits**. Il se peut en effet que l'un des 2 soit déjà dans le bon état. Néanmoins, **l'utilisation de cette macro ne vous coûtera qu'une ligne supplémentaire, et limitera fortement les risques d'erreurs**. Vous verrez d'autres macros dans cette maquette, n'en tenez pas compte pour l'instant.

De plus, certaines macros que je développe dans mes exercices et dans les fichiers maquettes ont leur équivalent (ou presque) déjà intégré dans MPASM. Cependant, dans ce cas, leur contenu n'est pas visible, c'est donc une démarche moins didactique (n'oubliez pas qu'il s'agit d'un cours). De toute façon, je vous parlerai de ces macros particulières.

Ensuite vous trouvez les différentes zones d'emplacement RAM pour chacune des banques, ainsi que l'emplacement des **16 octets de la zone commune**. Je rappelle que ces 16 octets **sont accessibles indépendamment de la valeur de RP0 et RP1**.

```
; *****
;
;                               VARIABLES ZONE COMMUNE
; *****
; Zone de 16 bytes
; -----

CBLOCK 0x70                ; Début de la zone (0x70 à 0x7F)
```

```

w_temp : 1      ; Sauvegarde registre W
status_temp : 1 ; sauvegarde registre STATUS
FSR_temp : 1    ; sauvegarde FSR (si indirect en interrupt)
PCLATH_temp : 1 ; sauvegarde PCLATH (si prog>2K)
ENDC

```

Dans cette zone, vous trouvez les variables de sauvegarde temporaire **nécessaires lorsque vous utilisez les interruptions**. Les explications concernant ces variables sont données dans le chapitre 5 sur les interruptions.

Pour notre exemple, supprimez ces variables, nous n'utiliserons pas les interruptions. Supprimez enfin tout ce qui suit les lignes suivantes, excepté la directive « END » :

```

; *****
;                                     DEMARRAGE SUR RESET
; *****

org 0x000      ; Adresse de départ après reset

```

4.2 Création d'un programme sans saut de page

Nous allons maintenant compléter notre petit exercice. N'oubliez pas de toujours laisser la directive « **END** » après la dernière ligne de votre programme.

Ajoutez maintenant les lignes suivantes, puis compilez avec **<F10>**. Tout devrait se passer sans problème.

```

saut1
nop      ; ne rien faire
nop      ; ne rien faire
goto    saut1 ; sauter

END

```

Tapez ensuite **<F6>**, le curseur du simulateur vient se placer sur la ligne à l'adresse **0x00**. Pressez ensuite plusieurs fois **<F7>** pour constater que votre boucle fonctionne sans problème.

4.3 Premières tentatives de saut inter-pages

Nous allons maintenant provoquer un **saut entre 2 pages distinctes**. Tout d'abord, **quelle est la longueur d'une page ?**

Et bien, rappelez-vous : elle est de 2K mots, qui est la limite des 11 bits imposée par l'architecture des instructions de saut. Donc, on **débordera de la page** en passant de **B'1111111111** à **B'100000000000**, autrement dit quand on a modification du bit 12 ou du bit 13 (bit 12 dans ce cas). Cette nouvelle adresse est l'**adresse 0x800**. Voici donc les zones des différentes pages :

```

Page 0 : de 0x0000 à 0x07FF
Page 1 : de 0x0800 à 0x0FFF

```

Page 2 : de 0x1000 à 0x17FF

Page 3 : de 0x1800 à 0x1FFF

Modifions donc notre petit programme de la façon suivante (n'oubliez pas la directive « **END** », je ne l'indiquerai plus) :

```
org 0x000      ; Adresse de départ après reset
nop           ; ne rien faire
saut1
nop           ; ne rien faire
nop           ; ne rien faire
goto saut2    ; sauter

org 0x800      ; adresse de la suite du programme
saut2
nop           ; ne rien faire
nop           ; ne rien faire
goto saut1    ; sauter
```

Lancez la compilation avec <F10>. Dans la fenêtre des résultats, vous voyez apparaître 2 « warnings ». Les numéros de lignes peuvent être différents en fonction des espaces entre les lignes :

```
Message[306] D:\DOCUME~1\LESSONS\PICS-P~2\FICHIERS\PCLATH.ASM 214 : Crossing
page boundary -- ensure page bits are set.
Message[306] D:\DOCUME~1\LESSONS\PICS-P~2\FICHIERS\PCLATH.ASM 221 : Crossing
page boundary -- ensure page bits are set.
```

Si vous vous reportez aux lignes incriminées (dans mon cas 214 et 221), vous constaterez que ces lignes correspondent aux deux instructions de saut.

En fait **MPASM** attire votre attention sur le point suivant : les sauts incriminés traversent une limite de page, il vous demande de vérifier si les bits correspondants de **PCLATH** sont bien positionnés. Ceci est logique, car la directive « org0x800 » place le « saut2 » en page 1, alors que le saut 1 est situé en page 0.

Ne tenons pas compte de ces avertissements dans un premier temps. Tapez donc ensuite <F6> puis une succession de <F7> pour voir le déroulement du programme.

Que se passe-t-il ? Au lieu que l'instruction « goto saut2 » nous envoie vers « saut2 », nous nous retrouvons en fait à la première ligne du programme.

Examinons ce qui s'est passé :

- L'instruction « goto saut2 » est traduite par l'assembleur en « goto 0x800 »
- Or, 0x800 comporte 12 bits, donc est tronquée à 11 bits, ce qui donne « goto 0x00 ». En effet $0x800 = B'100000000000$. Si vous enlevez le bit de poids fort, il reste $B'000000000000$
- La gestion interne du PIC ajoute à l'adresse les bits 3 et 4 de **PCLATH**, qui sont à « 0 ». L'adresse finale est donc 0x00. En effet, l'étude des 2 dernières colonnes du tableau 2-1

de la page 15 du datasheet nous apprend que ce registre est toujours remis à « 0 » après un reset ou une mise sous tension.

- Que trouvons-nous à l'adresse 0x00 ? Tout simplement la ligne qui suit la directive « org 0x00 ».

Modifions encore notre programme pour voir si vous avez tout compris :

Remplaçons pour cela la ligne

```
org 0x800 ; adresse de la suite du programme
```

Par

```
org 0x850 ; adresse de la suite du programme
```

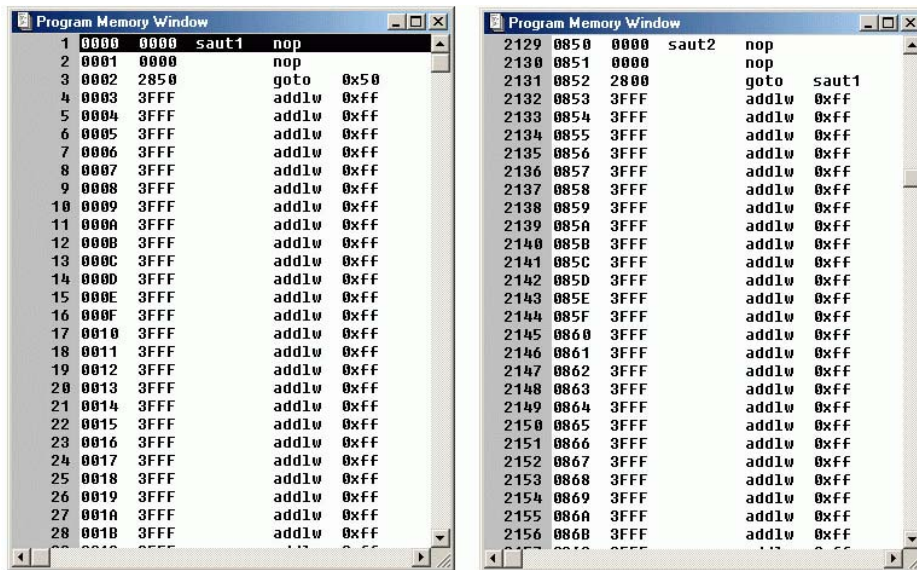
Profitez-en pour ouvrir la fenêtre « Special Function Registers » du menu « Window ». Ceci vous permettra de suivre l'évolution du registre PCL, à titre d'information.

Lancez la compilation, puis faites avancer votre programme lentement en pas à pas. Arrivé la ligne du saut vers « saut2 », pressez encore une fois <F7>. Que se passe-t-il ? Une nouvelle fenêtre s'ouvre alors, intitulée « Program Memory Window ». Cette fenêtre représente le contenu de votre mémoire programme.

En effet, où donc avez-vous sauté ? Appliquons de nouveau la même méthode. Le saut a donc eu lieu vers l'adresse 0x050, au lieu de 0x850 prévue. donc hors de notre programme. MPLAB ne peut donc pas pointer dans votre fichier source, il est contraint d'ouvrir cette nouvelle fenêtre.

Si vous ne pointez pas sur la bonne adresse, pas de panique, MPLAB semble ne pas apprécier ce type de manœuvre. Faites défiler la fenêtre jusqu'à l'adresse 0x00. Vous verrez alors la première partie de votre programme.

Remarquez que l'assembleur a bien traduit le saut en « goto 0x050 ». Défilez maintenant jusque l'adresse 0x850. vous voyez alors la seconde partie de notre programme.



4.4 Evitons le piège

Nous allons encore une fois modifier notre programme en remplaçant la ligne

```
org 0x850 ; adresse de la suite du programme
```

par

```
org 0x7FF ; adresse de la suite du programme
```

Nous allons donc sauter juste avant la limite de la page, ce qui fait que la première instruction suivant l'étiquette « saut2 » sera en page 0, tandis que la suite du programme sera en page 1. Nous nous trouvons donc au lieu de débordement de page d'un programme.

Recompilons notre programme et analysons la fenêtre des résultats. Nous constatons alors qu'il n'y a plus qu'un seul warning, correspondant au second saut. Ceci est logique, car l'étiquette « saut2 » est maintenant située dans la même page (0) que le premier « goto ».

Faisons donc tourner notre programme en pas-à-pas. Il semble maintenant fonctionner tout à fait correctement. Quel est donc ce mystère ?

En fait, comme toujours, ce n'est que pure logique. Il y a 3 phases à distinguer dans le fonctionnement de ce petit bout de code.

- Le premier saut (goto saut1) s'effectue sans problème, puisque l'adresse de destination (0x7FF) est située dans la page 0, les bits 3 et 4 de PCLATH étant à '0', l'adresse de destination obtenue est bien 0x7FF
- Ensuite, en avançant en pas à pas, on passe à la page2. Notez en passant 2 choses. PCL est remis à 0, ce qui est logique, et PCLATH voit son contenu inchangé. C'est ici qu'il ne faut pas tomber dans le piège :

LE REGISTRE PCLATH N'EST JAMAIS MODIFIE AUTOMATIQUEMENT PAR LE PROGRAMME.

C'est à l'utilisateur seul de procéder à sa modification. Mais PCLATH n'est utilisé que pour les sauts directs, PAS pour le déroulement du programme. Si vous regardez en effet le schéma-bloc du 16F876, vous constatez que le compteur de programme a une capacité de 13 bits, donc ne pose aucun problème pour le déroulement séquentiel du programme.

- Ensuite, vous trouvez le saut « goto saut1 ». De nouveau on s'attend à un problème, et il n'en est rien. Pourquoi ? Et bien tout simplement, malgré le warning de MPASM qui demande de vérifier PCLATH, ce dernier est tout simplement bien configuré.

En effet, nous n'avons pas jusqu'à présent modifié PCLATH, donc il pointe toujours sur la page 0. Donc notre saut s'effectuera bien en page0. Donc,

CE N'EST PAS PARCE QUE VOTRE PROGRAMME S'EXECUTE DANS UNE PAGE QUE PCLATH POINTE VERS CETTE MEME PAGE. IL VOUS INCOMBE DE GERER VOUS-MEME PCLATH.

Si vous avez compris ce qui précède, vous allez tout comprendre dans le fonctionnement de PCLATH. Rassurez-vous, je vous le rappelle, ceci ne vous sera utile que pour des programmes d'une taille supérieure à 2K mots, donc vous avez le temps de bien assimiler. Je commence cependant par cette partie, car je reçois énormément de courrier de personnes qui désirent modifier un code-source écrit pour un 16F876. Ces modifications entraînent parfois un débordement de page, ce qui empêche donc le programme modifié de fonctionner.

4.5 Première correction de notre exercice

Remplaçons encore une fois notre ligne

```
org 0x7FF ; adresse de la suite du programme
par
org 0x850 ; adresse de la suite du programme
```

Mais, cette fois, nous allons tenter d'effectuer correctement nos sauts. Pour ce faire, occupons-nous d'abord du premier. Nous devons donc sauter de la page 0 vers la page1.

Mais avant tout, voici un petit tableau destiné à vous éviter l'usage de la calculatrice pour établir les valeurs des bits 3 et 4 de PCLATH.

Adresses de destination	Page	B4	B3
0x0000 à 0x07FF	0	0	0
0x0800 à 0x0FFF	1	0	1
0x1000 à 0x17FF	2	1	0
0x1800 à 0x1FFF	3	1	1

Nous voyons d'après ce tableau que nous devons positionner le bit3 de PCLATH à '1' pour pouvoir sauter en page1, 0x850 étant bien situé dans cette page.

Donc, avant la ligne de saut, nous devons ajouter la mise à 1 de ce bit. Notez que l'analyse du tableau des registres du datasheet nous montre que **PCLATH** est présent dans toutes les banques. Donc, inutile de positionner correctement **RP0** et **RP1**. Et oui, j'en vois qui avaient déjà purement ignoré ce détail (rappelez-vous, plus de la moitié des erreurs de programmation)...

```
bsf    PCLATH , 3    ; préparer saut en page 1
goto   saut2         ; sauter
```

Maintenant, occupons-nous de notre second saut. En effet, l'adresse de destination va dépendre également, COMME TOUS LES SAUTS, du contenu de **PCLATH**. Si nous laissons **PCLATH** dans cet état, le saut2 s'effectuera en réalité vers l'adresse 0x800. Aussi, avant le saut, devons-nous remettre **PCLATH** en configuration « page 0 ».

```
bcf    PCLATH , 3    ; préparer saut en page 0
goto   saut1         ; sauter
```

Compilons le programme : toujours les 2 warnings d'avertissement de franchissement de pages. Exécutons le programme : il tourne maintenant parfaitement. Vous voyez, rien de sorcier, juste de la réflexion.

4.6 Evitons les warnings inutiles

Nous avons vu que les warnings sont bien pratiques dans ce cas précis pour nous indiquer les sauts inter-page. Le problème, c'est qu'une fois la solution trouvée, le warning persiste. Or, dans le cas d'un tout gros programme, puisqu'il s'agit ici de programmes dépassant 2K mots, vous risquez de vous retrouver avec des dizaines de warnings de ce type.

Une fois que vous aurez réglé chaque cas séparément, si vous modifiez votre programme, et que cette modification, par exemple une insertion, provoque des modifications dans les emplacements de sauts de pages, vous serez dans l'incapacité de le vérifier sans tout reprendre depuis le début.

Autre inconvénient, si votre correction est mauvaise, vous n'aurez pour le vérifier aucun autre choix que de tout contrôler depuis le début. J'ai rencontré également ce problème, et j'ai trouvé une astuce qui me met à l'abri de ce type d'inconvénient, car je transfère alors le contrôle des sauts de page directement à **MPASM**.

Je vais vous montrer quelle a été ma démarche et quelle solution pratique j'ai trouvée (de nouveau, certaines macros intégrées à MPLAB effectuent le même style d'opérations).

MPASM génère un warning de saut de page quand la page de destination est différente de la page dans laquelle l'instruction de saut est exécutée. L'astuce consiste à faire croire à **MPASM** qu'il se trouve dans la même page. Comment ? Tout simplement en modifiant les adresses de destination. Je commence par expliquer directement dans le source du programme.

Voyons notre premier saut : nous nous trouvons en page0 et nous désirons sauter à l'adresse « saut2 » située en page1. **MPASM** remplace « saut2 » par sa valeur et voit que cette valeur n'est pas en page0. Nous allons donc le bluffer.

Pour lui faire croire que le saut s'effectue en page0, il suffit de supprimer le bit 11 de l'adresse, afin de ne garder que les 11 bits utilisés en réalité. Cette opération n'aura aucun impact, puisque l'assemblage ne conservera de toute façon que les 11 bits de poids faible. Par contre, MPASM n'aura plus aucune raison de générer un warning.

Modifions donc simplement notre ligne

```
goto    saut2                ; sauter
en
goto    (saut2 & 0x7FF) ; sauter
```

Compilez le tout : le premier warning a disparu, occupons-nous du second. Ici, c'est exactement le contraire. MPASM constate que nous sommes en page 1 et que nous sautons en page 0. Il suffit donc de lui faire croire que nous sautons en page1. Comment ? Tout simplement en forçant le bit 11 de l'adresse à '1'. Ceci n'aura aucune conséquence, puisque à l'assemblage, seuls les 11 bits de poids faibles seront conservés.

Modifions donc simplement notre ligne

```
goto    (saut1 )            ; sauter
en
goto    (saut1 | 0x800) ; sauter
```

Le symbole « | » signifie pour l'assembleur « OR » (voir première partie, opérations booléennes). Il est obtenu en maintenant la touche <alt> de droite enfoncée et en tapant la touche <1> pour le clavier belge, et <6> pour le clavier français (pas sur le pavé numérique).

Compilons tout ça, plus de warning, exécutons le programme : tout fonctionne bien. Mais tout ceci n'est guère pratique, ni facile à mettre en place. De plus le risque d'erreur devient très grand au fur et à mesure de l'augmentation de taille de notre programme.

En effet, si différentes insertions déplacent par exemple notre saut2 vers la page2, et bien l'instruction de saut placée en page1 ne générera plus aucun warning. Ceci est logique, car on aura définitivement fait croire à MPASM que notre appel s'effectue à l'intérieur de la page0. Nous n'avons donc résolu notre problème que de manière temporaire.

Faites l'essai si vous n'êtes pas convaincu. Modifiez « org 0x850 » en « org 1000 », et vous constaterez que vous n'avez pas de warning pour l'instruction « goto saut2 », alors que le saut est erroné.

4.7 Démonstration pratique de l'utilité des macros

Pour que ce soit pratique, reste à implémenter ceci dans des macros. Je vous ai donc mis au point un petit macro qui va faire tout le travail pour vous.

```
GOTOX macro ADRESSE                ; saut inter-page
local BIT4 = (ADRESSE & 0x1000) ; voir bit 12
local BIT3 = (ADRESSE & 0x0800) ; voir bit 11
local ICI                                     ; adresse courante
ICI
local PICI = (ICI+2 & 0x1800) ; page du saut
```

```

IF BIT3                                ; si page 1 ou 3
    bsf PCLATH , 3                    ; b3 de PCLATH = 1
ELSE                                    ; sinon
    bcf PCLATH , 3                    ; b3 de PCLATH = 0
ENDIF
IF BIT4                                ; si page 2 ou 3
    bsf PCLATH , 4                    ; b4 de PCLATH = 1
ELSE                                    ; sinon
    bcf PCLATH , 4                    ; b4 de PCLATH = 0
ENDIF
    goto (ADRESSE & 0x7FF | PIC1); adresse simulée
endm

```

Houlà ! J'en vois qui paniquent. Restez cool, j'explique ligne par ligne, c'est tout simple. Vous allez commencer à voir le véritable intérêt des macros. Plus qu'un simple traitement de texte, c'est, cette fois, un véritable instrument anti-erreurs.

```
GOTOX macro ADRESSE ; saut inter-page
```

Ici, tout simplement on déclare la macro. Cette macro se nomme donc GOTOX, et elle reçoit en argument ADRESSE, qui est l'adresse de destination du saut. Donc, pour un saut inter-page vers l'adresse « saut2 », on utilisera donc : « GOTOX saut2 »

```
local BIT4 = (ADRESSE & 0x1000) ; voir bit 12
```

Dans cette ligne, nous trouvons d'abord la directive « LOCAL », qui signifie que la valeur sera recalculée à chaque appel de la macro. La valeur « BIT4 » sera donc différente à chaque fois que vous appellerez « GOTOX ». C'est logique.

Vient derrière le calcul de « BIT4 ». On pourra dire que BIT4 est égal à l'adresse de destination AND 0x1000, soit B'10000000000000. Donc, BIT4 vaudra 0 si l'adresse de destination tient sur 12 bits, donc inférieure à 0x1000, donc en page 0 ou 1. C'est à dire si le bit 4 de PCLATH devra être mis à « 0 » pour sauter. L'adresse de destination a dans ce cas son bit 12 à « 0 ».

```
local BIT3 = (ADRESSE & 0x0800) ; voir bit 11
```

Strictement identique pour une adresse qui comporte son bit 11 à 0. Donc, ceci nous permet de savoir si l'adresse de destination se situe ou non en page 1 et 3. Reportez-vous au tableau chapitre 4.5

```
local ICI ; adresse courante
ICI
```

Ici, nous déclarons simplement une étiquette locale. Cette déclaration est suivie par l'étiquette en elle-même, en première colonne, comme il se doit. Ceci nous permet de savoir où, et plus précisément dans quelle page, la macro « GOTOX » a été appelée. La directive « local » est indispensable, car si on appelle plusieurs fois la macro « GOTOX », « ICI » sera à chaque fois à un endroit différent.

```
local PIC1 = (ICI+2 & 0x1800) ; page du saut
```

Bon, ici que va-t-on faire ? Et bien tout simplement **déterminer la page dans laquelle se trouve notre macro**. En effet, nous avons vu que pour berner les warnings de **MPASM**, il faut connaître la page dans laquelle on se trouve au moment de l'appel.

Donc, **on commence par ajouter 2 à l'adresse courante**. Pourquoi ? Tout simplement parce que ce qui importe, c'est l'adresse du « **goto** ». Or, notre macro va s'occuper également automatiquement des bits 3 et 4 de **PCLATH**, et ceci, bien évidemment, avant le « **goto** » en question. Donc, l'adresse de notre « **goto** » est l'adresse actuelle incrémentée de 2, **puisque 2 instructions le précéderont**.

Ensuite, on trouve un « **AND** » avec 0x1800, ce qui a pour effet **d'éliminer l'adresse en cours et de conserver son adresse de page**. Donc, à l'issue de ceci, « **PICI** » vaudra 0x000, 0x800, 0x1000, ou 0x1800.

```
IF BIT3          ; si page 1 ou 3
```

Ceci signifie tout simplement que la ou **les lignes suivantes seront assemblées SI la valeur de BIT3 existe**, donc si « **BIT3** » est différent de « 0 », donc si on doit sauter en page 1 ou en page 3.

```
bsf PCLATH , 3    ; b3 de PCLATH = 1
```

Dans ce cas, MPASM créera automatiquement cette ligne, c'est à dire que le bit3 de **PCLATH** sera mis automatiquement à 1.

```
ELSE              ; sinon
    bcf PCLATH , 3 ; b3 de PCLATH = 0
```

Tout simplement : **sinon** (donc si **BIT3** = 0), **MPASM écrira automatiquement cette dernière ligne**, donc le bit3 de **PCLATH** sera mis automatiquement à 0.

```
ENDIF
```

Ceci indique la fin du test précédent.

```
IF BIT4          ; si page 2 ou 3
    bsf PCLATH , 4 ; b4 de PCLATH = 1
ELSE              ; sinon
    bcf PCLATH , 4 ; b4 de PCLATH = 0
ENDIF
```

Et bien, ici, c'est **strictement identique, mais concernera le bit4 de PCLATH**. Cette macro vous permet donc de sauter automatiquement de et vers n'importe laquelle des 4 pages sans plus vous soucier de rien. Avouez que c'est rudement pratique.

```
goto (ADRESSE & 0x7FF | PICI) ; adresse simulée
```

Ceci, c'est la fameuse ligne, un peu améliorée, que vous avez utilisée dans l'exemple précédent. **C'est cette ligne qui va « berner » MPASM** en lui faisant croire que vous sautez dans la page actuelle.

La première opération (`ADRESSE & 0x7FF`) permet de conserver l'adresse codée sur 11 bits, tandis que la seconde (`PIC1`) ajoute en fait l'adresse de base de la page courante.

Il ne reste plus qu'à modifier notre programme principal.

```
org 0x000      ; Adresse de départ après reset
saut1
nop            ; ne rien faire
nop            ; ne rien faire
GOTOX saut2    ; sauter

org 0x850      ; adresse de la suite du programme
saut2
nop            ; ne rien faire
nop            ; ne rien faire
GOTOX saut1    ; sauter en page 0
END            ; directive fin de programme
```

Compilez le tout : plus de warning. Exécutez : tout ce passe sans problème. Allez jeter un œil sur le code obtenu à l'aide de cette macro dans la fenêtre « `window-> program memory` ». Amusez-vous également à modifier les directives « `org` » pour essayer toutes sortes de combinaisons de pages. Vous verrez qu'à chaque fois `MPASM` compile correctement votre programme. Plus de warning, plus de soucis.

Il ne vous reste plus qu'une chose à retenir : en cas de warning concernant les sauts de page, remplacez l'instruction « `goto` » par la macro « `GOTOX` » pour les lignes concernées, et le problème sera automatiquement résolu. N'est-ce pas utile, les macros ?

Attention, ne tombez cependant pas dans le piège grossier suivant. Par exemple, ne remplacez pas :

```
btfss STATUS , Z
goto plusloin
par
btfss STATUS , Z
GOTOX plusloin
```

Pourquoi ? Tout simplement parce que `btfss` permet de sauter l'instruction suivante, et que `GOTOX` est une macro qui contient 3 instructions. Donc, le test ne fonctionnerait pas.

En effet, ce ne serait pas `goto` qui suivrait `btfss`, mais `bsf` ou `bcf PCLATH`, 3. Il vous faudra donc, soit exécuter votre test d'une autre façon, soit placer les modifications de `PCLATH` avant le test.

Je vous ai écrits 2 macros séparées directement utilisables avec ce genre de configuration. : « `PCLAX` » (modifie `PCLATH`) et « `GOTSX` » (saut simple, sans modification de `PCLATH`). En fait, `PCLAX` suivi de `GOTSX` est l'équivalent de `GOTOX`.

L'avantage est de séparer le saut de l'initialisation de `PCLATH`.

```
PCLAX macro ADRESSE      ; positionne PCLATH pour
                        ; les sauts sans le saut
local BIT4 = (ADRESSE & 0x1000) ; voir bit 12
local BIT3 = (ADRESSE & 0x0800) ; voir bit 11
```

```

IF BIT3                                ; si page 1 ou 3
    bsf PCLATH , 3                     ; b3 de PCLATH = 1
ELSE                                    ; sinon
    bcf PCLATH , 3                     ; b3 de PCLATH = 0
ENDIF
IF BIT4                                ; si page 2 ou 3
    bsf PCLATH , 4                     ; b4 de PCLATH = 1
ELSE                                    ; sinon
    bcf PCLATH , 4                     ; b4 de PCLATH = 0
ENDIF
Endm

```

```

GOTSX macro ADRESSE                    ; saut inter-page sans
                                        ; sélection de PCLATH
    local ICI                          ; adresse courante
    local PICI = (ICI & 0x1800)        ; page du saut
ICI
    goto (ADRESSE & 0x7FF | PICI)     ; adresse simulée
endm

```

En fait, il s'agit tout simplement de la **macro « GOTOX » scindée** pour pouvoir être utilisée facilement avec les tests. Si on reprend notre exemple, on pourra remplacer

```

btfss STATUS , Z
goto plusloin

```

par

```

PCLAX pluloain                        ; configure le registre PCLATH pour le saut
btfss STATUS , Z                      ; effectue le test (n'importe lequel)
GOTSX plusloin                        ; saute avec leurre de warning (une seule
                                        ; instruction)

```

Pour information, la macro intégrée PAGESEL est équivalente à notre macro PCLAX.

```

PAGESEL pluloain                     ; configure le registre PCLATH pour le saut
btfss STATUS , Z                      ; effectue le test (n'importe lequel)
GOTSX plusloin                        ; saute avec leurre de warning (une seule
                                        ; instruction)

```

4.8 Les sous-programmes inter-pages

Maintenant, vous allez me dire. Les sauts de type « **goto** », c'est bien joli, mais que se passe-t-il pour les sous-routines ?

Et bien, identiquement **la même chose**, à un détail près. Une sous-routine se termine par une instruction de retour (return, retlw). Si vous regardez le schéma-bloc, vous verrez que la totalité du compteur de programme, c'est à dire les 13 bits, a été conservée dans la pile.

Donc, vous devez positionner PCLATH pour l'appel de sous-routine, exactement comme pour un saut, par contre, vous ne devez pas vous en préoccuper concernant le retour. Ceci est heureux, car sinon il serait difficile d'appeler la même sous-routine depuis 2 pages différentes.

Par contre, comme votre programme revient à sa page d'appel après le return, vous devrez repositionner correctement **PCLATH** pour éviter qu'un saut « ordinaire » ne prenne en compte votre modification de **PCLATH**.

Le retour s'effectuera donc toujours correctement quelles que soient les pages d'appel et d'emplacement de la sous-routine. Reste seulement à établir pour les « **call** », les macros correspondant aux « **goto** ». Les voici :

```
CALLX macro ADRESSE                                ; call inter-page
    local BIT4 = (ADRESSE & 0x1000)                ; voir bit 12
    local BIT3 = (ADRESSE & 0x0800)                ; voir bit 11
    local ICI                                       ; adresse courante
ICI
    local PIC1 = (ICI+2 & 0x1800)                   ; page du saut
    IF BIT3                                         ; si page 1 ou 3
        bsf PCLATH , 3                             ; b3 de PCLATH = 1
    ELSE                                           ; sinon
        bcf PCLATH , 3                             ; b3 de PCLATH = 0
    ENDIF
    IF BIT4                                         ; si page 2 ou 3
        bsf PCLATH , 4                             ; b4 de PCLATH = 1
    ELSE                                           ; sinon
        bcf PCLATH , 4                             ; b4 de PCLATH = 0
    ENDIF
    call (ADRESSE & 0x7FF | PIC1)                   ; adresse simulée
    local BIT4 = ((ICI+5) & 0x1000)                ; voir bit 12
    local BIT3 = ((ICI+5) & 0x0800)                ; voir bit 11
    IF BIT3                                         ; si page 1 ou 3
        bsf PCLATH , 3                             ; b3 de PCLATH = 1
    ELSE                                           ; sinon
        bcf PCLATH , 3                             ; b3 de PCLATH = 0
    ENDIF
    IF BIT4                                         ; si page 2 ou 3
        bsf PCLATH , 4                             ; b4 de PCLATH = 1
    ELSE                                           ; sinon
        bcf PCLATH , 4                             ; b4 de PCLATH = 0
    ENDIF
endm

CALLSX macro ADRESSE                                ; sous-routine inter-page sans
                                                    ; sélection de PCLATH
    local ICI                                       ; adresse courante
    local PIC1 = (ICI+2 & 0x1800)                   ; page du saut
ICI
    call (ADRESSE & 0x7FF | PIC1)                   ; adresse simulée
endm
```

La macro **PCLAX** étant identique, inutile de l'écrire deux fois.

Le calcul (ICI+5) se justifie par le raisonnement suivant :

Il importe de repositionner **PCLATH** pour qu'il pointe dans la page courante, afin d'éviter qu'un simple « goto » ou un « call » à l'intérieur de la page ne provoque en réalité un saut dans la page précédemment pointée par **PCLATH** modifié par la macro. Tout ceci est réalisé automatiquement par notre macro **CALLX**

La macro va générer **5 lignes de code**, et donc après son exécution, le programme continuera à l'adresse du début de la macro (ICI) incrémentée de 5.

Vous n'avez donc plus qu'à placer votre macro à la place de votre call habituel.

```
Etiquette_pagex          ; on est en page x
    CALLX  etiquette_pagey ; appel de sous-routine en page y
    Suite du programme principal ; poursuite, avec PCLATH pointant
                                ; automatiquement sur la page x

etiquette_pagey          ; sous-routine en page y
    ici commence la sous-routine ; traitement normal
    return                ; retour normal de sous-routine
```

Vous pouvez également pointer sur la bonne page en utilisant **PCLAX**, et en indiquant l'adresse courante comme adresse de destination.

```
    PCLAX  etiquette_pagey ; pointer sur page y
    CALLSX etiquette_pagey ; appel de sous-routine en page y
ici      ; étiquette pour déterminer pagecourante
    PCLAX  ici              ; repointer sur page courante
    Suite du programme
```

Ou, enfin, en utilisant la macro intégrée de MPASM.

```
    PAGESEL etiquette_pagey ; pointer sur page y
    CALLSX  etiquette_pagey ; appel de sous-routine en page y
ici      ; étiquette pour déterminer pagecourante
    PAGESEL ici              ; repointer sur page courante
    Suite du programme
```

Vous voyez que CALLX est tout de même plus pratique, avec bien moins de risques d'oubli.

J'ai été particulièrement long dans ce chapitre, mais cette notion était assez délicate à aborder, et il me semblait dommageable de vouloir utiliser un processeur plus puissant et se retrouver bloqué le jour on désire créer un programme conséquent.

A la fin de ce chapitre, je place dans le répertoire « fichiers », le programme « **pclath.asm** », qui contient l'exercice terminé.

Notes : ...

Notes : ...

5. Les sources d'interruptions

J'ai choisi d'énumérer les différentes sources d'interruption du 16F876, car nous allons en avoir besoin dans les chapitres suivants. De plus, nous allons pouvoir mettre en évidence « quelques » différences avec le 16F84, ce qui place ce chapitre dans la suite logique du précédent.

5.1 Enumération

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 sus. 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

Quelques mots sur ce tableau.

En jaune vous avez les 3 interruptions dont le traitement est identique à celui du 16F84 et que nous avons vues dans la première partie.

En vert, vous voyez l'interruption d'écriture eeprom dont le traitement a été modifié. Cette interruption était déjà présente sur le 16F84, mais son traitement est ici légèrement différent.

En bleu ce sont les nouvelles interruptions utilisables sur le 16F876.

La ligne grise se réfère à l'interruption supplémentaire du 16F877 ou 16F874.

L'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 banque0, avec **INTCON** présent dans les 4 banques)
- **PEIE** : Indique si le positionnement de **PEIE** du registre **INTCON** joue un rôle. C'est à cause du **OUI** dans la ligne **EEPROM** que le traitement de cette interruption est ici légèrement différente du 16F84.
- **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)

5.2 Le registre INTCON et les interruptions périphériques

Commençons donc par ce que nous connaissons, et pour faire simple, comparons le contenu de ce registre avec celui du 16F84, histoire d'avoir une idée des modifications.

INTCON pour le 16F84							
GIE	EEIE	TOIE	INTE	RBIE	TOIF	INTF	RBIF
INTCON pour le 16F87x							
GIE	PEIE	TOIE	INTE	RBIE	TOIF	INTF	RBIF

En fait, nous constatons que tout est resté identique au 16F84, excepté **EEIE**, devenu **PEIE**. Nous allons expliquer pourquoi.

En fait, Microchip a choisi une méthode un peu particulière, et a scindé les interruptions entre ce que nous pourrions appeler les **interruptions primaires** et les interruptions secondaires, appelées ici **interruptions périphériques**.

Pour réaliser ceci, on a donc **ajouté au registre INTCON un second bit de validation** « générale » des interruptions qui ne concerne que les interruptions périphériques.

Comme il n'y avait pas la place dans ce registre, le bit **EEIE** concernant l'écriture en eeprom, et considéré comme « secondaire », a été déplacé vers le registre **PIR2**.

5.2.1 Mise en service des interruptions primaires

Des constatations précédentes, nous pouvons dire que nous disposons de **3 interruptions primaires**. C'est à dire utilisables exactement de la même manière que pour le 16F84. Pour mettre en service une de ces interruptions, vous devez donc :

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

Donc, à ce niveau vous ne devriez avoir aucun problème, rien de changé.

5.2.2 Mise en service des interruptions périphériques

Nous avons vu que 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.

5.3 Les registres **PIE1**, **PIE2**, **PIR1** et **PIR2**

Vous avez vu que ces composants disposent de plus de sources d'interruptions que ne peut en gérer le registre **INTCON**. Donc, les autorisations d'interruptions vont se trouver dans d'autres registres. Ces registres sont **PIE1** et **PIE2**. Les flags correspondants se trouvent quant à eux dans les registres **PIR1** et **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	NI	Réservé	NI	EEIE	BCLIE	NI	NI	CCP2IE
PIR2	0x0D	NI	Réservé	NI	EEIF	BCLIF	NI	NI	CCP2IF

Vous remarquerez que les 2 registres d'autorisations (**PIE1** et **PIE2**) se trouvent en banque 1, tandis que les registres de flags (**PIR1** et **PIR2**) se trouvent en banque 0.

Remarquez aussi 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

PSPIE	Lecture/écriture sur le port PSP // (concerne uniquement les 16F877/16F874)
ADIE	Conversion analogique/digitale
RCIE	Réception d'un caractère sur le port série USART
TXIE	Emission d'un caractère sur le port série USART
SSPIE	Communication sur le port série synchrone SSP
CCP1IE	Événement sur compare/capture registre 1
TMR2IE	Correspondance de valeurs pour le timer TMR2
TMR1IE	Débordement du timer TMR1

PIE2

EEIE	Écriture dans l'EEPROM
BCLIE	Collision de bus pour le port série synchrone I2C
CCP2IE	Événement sur compare/capture registre 2

Et maintenant, les flags correspondants. Attention, certains flags nécessitent une explication assez poussée, explications qui seront reprises dans les chapitres correspondant aux événements concernés.

PIR1

PSPIF	Lecture ou écriture terminée sur le port // du 16F877
ADIF	Fin de la conversion analogique/digitale
RCIF	Le buffer de réception de l'USART est plein (lecture seule)
TXIF	Le buffer d'émission de l'USART est vide (lecture seule)
SSPIF	Fin de l'événement dépendant du mode de fonctionnement comme suit : <ul style="list-style-type: none"> Mode SPI Un caractère a été envoyé ou reçu Mode I2C esclave Un caractère a été envoyé ou reçu Mode I2C 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)
CCP1IF	Événement compare/capture 1 détecté suivant mode : <ul style="list-style-type: none"> Mode capture : capture de la valeur TMR1 réalisée Mode compare : La valeur de TMR1 atteint la valeur programmée
TMR2IF	La valeur de TMR2 atteint la valeur programmée
TMR1IF	Débordement du timer TMR1

PIR2

EEIF	Fin d'écriture de la valeur en EEPROM
BCLIF	Collision de bus, quand le SSP est configuré en maître I2C
CCP2IF	Événement compare/capture 2 détecté suivant le mode : <ul style="list-style-type: none"> Mode capture : capture du TMR1 réalisée

Mode compare : La valeur de **TMR1** atteint la valeur programmée

5.3 Etude de la routine d'interruption du fichier « maquette ».

Voici le contenu de la routine d'interruption principale contenue dans le fichier « **m16f876.asm** ». Cette routine contient tous les tests qui permettent d'aiguiller les interruptions vers la bonne sous-routine. Dans la majorité des cas, vous n'aurez plus qu'à vous occuper de remplir correctement les sous-routines concernées.

Vous pourrez également supprimer les tests qui ne vous sont pas nécessaires, et, éventuellement inverser leur ordre afin de définir des priorités. **Soyez dans ce cas attentifs au bon usage du bit PEIE.**

```
;*****  
;  
; ROUTINE INTERRUPTION  
;*****  
;sauvegarder registres  
;-----  
org 0x004 ; adresse d'interruption  
movwf w_temp ; sauver registre W  
swapf STATUS , w ; swap status avec résultat dans w  
movwf status_temp ; sauver status swappé  
movf FSR , w ; charger FSR  
movwf FSR_temp ; sauvegarder FSR  
movf PCLATH , w ; charger PCLATH  
movwf PCLATH_temp ; le sauver  
clrf PCLATH ; on est en page 0  
BANK0 ; passer en banque0
```

Cette première partie effectue la sauvegarde des différents registres. Nous allons décrire ici leur utilité et leurs impératifs.

« **w_temp** » est la sauvegarde du registre de travail « **W** ». Ce registre va être utilisé pour la sauvegarde des autres registres. De fait, **c'est le premier qui doit être sauvé.**

Hors, si vous sauvez le registre « **W** » avant d'être autorisé à modifier « **STATUS** », qui lui n'est pas encore sauvé, vous ne pouvez donc pas changer de banque. En effet, le changement de banque nécessite la modification de **RP0** et **RP1** du registre « **STATUS** ».

Donc, si vous suivez toujours, au moment de l'interruption, vous ne pouvez pas savoir comment sont configurés **RP0** et **RP1**, donc vous ne savez pas vers quelle banque vous pointez. **Il est donc de ce fait impératif de sauvegarder « W » dans la zone RAM commune.**

REMARQUE IMPORTANTE

Les 16F873 et 16F874 ne disposent pas d'une zone de banque commune. Comme vous ne pouvez pas prévoir dans quelle banque vous allez sauver W, vous devez réserver la même adresse relative pour w_temp à la fois dans la banque 0 et dans la banque 1. Ensuite, vous effacerez le registre STATUS pour sauver les autres registres dans la banque 0 (par exemple)

```
movwf w_temp ; Sauver W dans la banque 0 ou dans la 1  
swapf STATUS,W ; charger STATUS swappé
```

```

clrf STATUS          ; pointer sur la banque 0
movwf status_temp    ; sauver STATUS en banque 0

```

La suite étant identique, excepté qu'il est inutile d'écrire la macro « BANK0 »

La déclaration sera du type :

```

CBLOCK    0x20        ; Début de la zone (0x20 à 0x7F)
    w_temp : 1        ; sauvegarde de W en banque 0
    ..
    ..
ENDC

CBLOCK    0xA0        ; Début de la zone (0xA0 à 0xFF)
    w_temp2 : 1       ; sauvegarde de w en banque 1 (même position impérative
    ..                ; que pour w_temp. Dans ce cas, première position
    ..                ; l'étiquette « w_temp2 » ne sera pas utilisée, mais
    ..                ; l'emplacement est réservé en pratiquant de la sorte.
ENDC

```

Revenons à notre 16F876 : Pour « **status_temp** » vous pourriez par contre commencer par charger status dans w, puis procéder à un changement de banque avant de le sauver. Sa sauvegarde en banque 0 n'est donc pas impérative. Vous pouvez donc placer status_temp dans n'importe quelle banque. Ceci est également vrai pour les registres suivants

« **FSR_temp** » sauvegarde le registre « **FSR** » d'adressage indirect. Ce registre ne devra être impérativement sauvé que si la routine d'interruption risque de perturber à ce niveau le programme principal. Il faut donc 2 conditions à ceci :

- Il faut que le programme principal utilise l'adressage indirect, sinon, aucune raison de sauvegarder **FSR**.
- ET il faut que la routine d'interruption utilise l'adressage indirect, car sinon le registre **FSR** ne serait pas modifié et il n'y aurait donc aucune raison de le sauvegarder.

Notez que rien ne vous empêche de sauvegarder FSR dans tous les cas pour des raisons de facilité et de précaution. Il ne vous en coûtera que 2 mots de programme et quelques nanosecondes de temps d'exécution. A vous de voir si ces surplus sont nuisibles ou non à votre programme. La même remarque est valable pour la sauvegarde de **PCLATH**.

Pour terminer, voyons donc la sauvegarde du registre **PCLATH** dans « **PCLATH_temp** » : cette sauvegarde n'a de sens bien entendu que si la routine d'interruption modifie **PCLATH**.

En général, cette modification risque d'intervenir si votre programme est d'une taille supérieure à 2K mots. Dans ce cas, en effet, le programme principal a de grandes chances de modifier **PCLATH** pour effectuer des sauts corrects, et vous serez alors contraint de remettre les bits 3 et 4 de **PCLATH** à 0.

Ceci pour éviter qu'un saut dans votre routine d'interruption ne sorte de celle-ci. Notez que si vous supprimez la sauvegarde de **PCLATH** dans votre routine d'interruption et que vous utilisez **PCLATH** pour un autre usage (tableau), vous devrez alors également supprimer la ligne « **clrf PCLATH** » qui modifie ce registre dans la routine d'interruption. Dans ce cas

particulier, vous n'êtes pas obligé de sauver PCLATH, mais vous ne pouvez pas le modifier non plus.

On termine ce début par la **macro BANK0**, qui permet de pointer sur la banque 0 en cas d'adressage direct.

Voyons maintenant la suite de notre routine :

```
    ; Interruption TMR0
    ; -----

    btfsc INTCON , T0IE      ; tester si interrupt timer autorisée
    btfss INTCON , T0IF      ; oui, tester si interrupt timer en cours
    goto  intsw1             ; non test suivant
    call  inttmr0            ; oui, traiter interrupt tmr0
    bcf   INTCON , T0IF      ; effacer flag interrupt tmr0
    goto  restorereg        ; et fin d'interruption
```

Il s'agit bien évidemment ici de vérifier si c'est le timer0 qui a déclenché l'interruption.

On commence par tester si l'interruption a été autorisée, en vérifiant le positionnement de **T0IE**. Vous allez me dire : drôle de procédure, quel est l'intérêt ?

Et bien, vous ne devez jamais perdre de vue que les flags sont positionnés quel que soit l'état du bit de validation correspondant. Vous avez donc **3 cas possibles** :

- 1) Vous n'utilisez pas les interruptions tmr0 dans votre programme : dans ce cas, vous pouvez tout simplement effacer ces 6 lignes de code.
- 2) Vous utilisez les interruptions tmr0 dans votre programme. Cependant, le bit **T0IE** reste en service tout au long de votre programme. Il est donc inutile de le tester. Vous pouvez donc effacer la première ligne.
- 3) Votre programme utilise les interruptions du timer0, mais pas tout le temps, de plus vous utilisez d'autres sources d'interruption. Donc, dans votre programme principal, vous coupez par moment **T0IE** afin de ne pas être interrompu par le débordement du timer0 pendant certaines périodes. Dans ce cas, si votre timer0 déborde, le flag **T0IF** sera positionné, mais l'interruption correspondante ne sera pas générée. Si par malheur se produit alors une interruption due à un autre événement que vous aviez autorisé, et que vous ne testiez pas **T0IE**, alors vous seriez induit en erreur lors du test de **T0IF**. Celui-ci étant positionné, bien que l'interruption ne soit pas due au débordement de tmr0. Vous devez donc laisser le test de **T0IE**.

La seconde ligne, qui vérifie si le bit **T0IF** est positionné, ne sera donc exécuté que si la ligne précédente a défini que **T0IE** est bien positionné. Donc, Si **T0IF** et **T0IE** sont tout deux positionnés, on saute à la ligne « **call inttmr0** ». Dans le cas contraire, on saute, via « **goto intsw1** », au test suivant pour continuer la recherche de la source de l'interruption.

La ligne « **call inttmr0** » permet d'appeler la sous-routine « **inttmr0** », dans laquelle vous pourrez placer le code de traitement de votre routine d'interruption timer0.

IMPORTANT : la routine d'interruption, telle qu'elle est écrite ici, utilise des sous-routines. Donc, si vous utilisez les interruptions, et sachant que vous disposez de 8 niveaux sur la pile (voir première partie), il vous restera donc : $8 - 1$ (pour les interruptions) $- 1$ (pour l'appel de sous-routine dans la routine d'interruption) = 6 imbrications de sous-routines possibles.

Si vous avez besoin d'un niveau supplémentaire, il vous suffira tout simplement de remplacer le « `call inttmr0` » par un « `goto inttmr0` ». Dans l'ancienne sous-routine « `inttmr0` », vous devrez donc remplacer le « `return` » par un « `goto restorerreg` », après avoir ajouté la ligne « `bcf INTCON, T0IF` » juste avant. Vous devrez bien entendu procéder de la même façon pour chaque interruption utilisée. Ceci vous économise donc un niveau de pile, par l'absence de sous-routines dans la routine d'interruption. Mais il est quant même peu fréquent dans un programme bien structuré de dépasser cette limite. Ne vous inquiétez donc pas de ceci pour l'instant.

La ligne jaune, « `bcf INTCON, T0IF` » permet d'effacer le flag après avoir traité la routine. N'oubliez pas que ces flags, pour la plupart (nous verrons les exceptions), ne s'effacent pas automatiquement. Si vous oubliez de les effacer, dès que le programme sortira de l'interruption, il y entrera de nouveau, et ce indéfiniment (à moins que le watchdog ne soit en service). Mais guère besoin de vous inquiéter, cette routine prend tout ceci en charge pour vous. Ce qu'il importe, c'est d'avoir compris, nul besoin de « réinventer la roue ».

La ligne verte, « `goto restorerreg` », quant à elle, permet, une fois le traitement de votre interruption effectué, de sortir directement de la routine d'interruption, après avoir restauré les différents registres.

Notez que la suppression de cette ligne permet de conserver les tests suivants, donc de traiter en une seule fois plusieurs interruptions « simultanées ». C'est à vous que le choix appartient en fonction du cas particulier de votre programme.

Il vous est également possible d'inverser l'ordre des différents tests. Ceci vous amène à établir une hiérarchie de priorités dans le cas d'interruptions multiples « simultanées ». La première testée sera dans ce cas la première traitée.

N'oubliez pas dans ce cas de bien prendre en compte le test de `PEIE`, ainsi que de gérer correctement les étiquettes.

Ensuite, nous trouvons :

```
                                ; Interruption RB0/INT
                                ; -----
intsw1
    btfsc INTCON , INTE        ; tester si interrupt RB0 autorisée
    btfss INTCON , INTF        ; oui, tester si interrupt RB0 en cours
    goto  intsw2               ; non sauter au test suivant
    call  intrb0               ; oui, traiter interrupt RB0
    bcf   INTCON , INTF        ; effacer flag interrupt RB0
    goto  restorerreg          ; et fin d'interruption

                                ; interruption RB4/RB7
                                ; -----
intsw2
    btfsc INTCON , RBIE        ; tester si interrupt RB4/7 autorisée
    btfss INTCON , RBIF        ; oui, tester si interrupt RB4/7 en cours
```

```

goto intsw3          ; non sauter
call intrb4          ; oui, traiter interrupt RB4/7
bcf  INTCON , RBIF   ; effacer flag interrupt RB4/7
goto restorereg      ; et fin d'interrupt

```

Ces deux tests n'amènent aucun commentaire particulier, et fonctionnent de façon identique au premier cas cité.

Vient ensuite :

```

intsw3
  btfss INTCON , PEIE      ; tester interruption périphérique autorisée
  goto restorereg         ; non, fin d'interruption

```

Ceci amène un petit commentaire. En effet, vous avez deux façons d'arriver à ce point de votre routine d'interruption :

- 1) Vous avez conservé les lignes « goto restorereg ». Dans ce cas, puisque vous arrivez à ce point du programme, c'est que forcément vous n'êtes pas passé par une des interruptions précédentes. En effet, après exécution d'une interruption précédente, cette partie aurait été sautée via la ligne en question.

Donc, dans ce cas, les événements précédents n'ont pas été les déclencheurs de l'interruption. Il s'agit donc forcément d'une interruption périphérique, donc, il est inutile de tester PEIE, car il est forcément à « 1 ».

- 2) Vous avez supprimé les lignes « goto restorereg ». Dans ce cas, il est possible qu'une interruption précédente ait eu lieu. Dans ce cas, il vous reste deux sous-solutions :
 - Si le bit PEIE reste inchangé durant toute l'exécution du programme, vous pouvez supprimer ce test.
 - Si vous modifiez le bit PEIE durant l'exécution de votre programme, vous devez laisser cette ligne, pour les mêmes raisons que celles évoquées pour le test de TOIE.

Naturellement, si vous n'utilisez dans votre programme aucune interruption périphérique, inutile de laisser ces lignes.

Notez que si vous ne touchez à rien, tout fonctionnera très bien dans tous les cas (sauf certains programmes spécifiques). Je vous conseille donc, dans le doute, de laisser cette routine telle quelle, si vous n'êtes pas à quelques microsecondes près.

Ensuite, nous trouvons les tests des interruptions périphériques

```

          ; Interruption convertisseur A/D
          ; -----

bsf  STATUS , RP0      ; sélectionner banque1
btfss PIE1 , ADIE      ; tester si interrupt autorisée
goto intsw4            ; non sauter
bcf  STATUS , RP0      ; oui, sélectionner banque0
btfss PIR1 , ADIF       ; tester si interrupt en cours
goto intsw4            ; non sauter
call intad              ; oui, traiter interrupt

```

```

bcf    PIR1 , ADIF      ; effacer flag interrupt
goto   restorereg      ; et fin d'interrupt

; Interruption réception USART
; -----
intsw4
    bsf    STATUS , RP0      ; sélectionner banque1
    btfss  PIE1 , RCIE      ; tester si interrupt autorisée
    goto   intsw5          ; non sauter
    bcf    STATUS , RP0      ; oui, sélectionner banque0
    btfss  PIR1 , RCIF      ; oui, tester si interrupt en cours
    goto   intsw5          ; non sauter
    call   intrc            ; oui, traiter interrupt
                                ; LE FLAG NE DOIT PAS ETRE REMIS A 0
    goto   restorereg      ; et fin d'interrupt

; Interruption transmission USART
; -----
intsw5
    bsf    STATUS , RP0      ; sélectionner banque1
    btfss  PIE1 , TXIE      ; tester si interrupt autorisée
    goto   intsw6          ; non sauter
    bcf    STATUS , RP0      ; oui, sélectionner banque0
    btfss  PIR1 , TXIF      ; oui, tester si interrupt en cours
    goto   intsw6          ; non sauter
    call   inttx           ; oui, traiter interrupt
                                ; LE FLAG NE DOIT PAS ETRE REMIS A 0
    goto   restorereg      ; et fin d'interrupt

; Interruption SSP
; -----
intsw6
    bsf    STATUS , RP0      ; sélectionner banque1
    btfss  PIE1 , SSPIE     ; tester si interrupt autorisée
    goto   intsw7          ; non sauter
    bcf    STATUS , RP0      ; oui, sélectionner banque0
    btfss  PIR1 , SSPIF     ; oui, tester si interrupt en cours
    goto   intsw7          ; non sauter
    call   intssp          ; oui, traiter interrupt
    bcf    PIR1 , SSPIF     ; effacer flag interrupt
    goto   restorereg      ; et fin d'interrupt

; Interruption CCP1
; -----
intsw7
    bsf    STATUS , RP0      ; sélectionner banque1
    btfss  PIE1 , CCP1IE    ; tester si interrupt autorisée
    goto   intsw8          ; non sauter
    bcf    STATUS , RP0      ; oui, sélectionner banque0
    btfss  PIR1 , CCP1IF    ; oui, tester si interrupt en cours
    goto   intsw8          ; non sauter
    call   intccp1         ; oui, traiter interrupt
    bcf    PIR1 , CCP1IF    ; effacer flag interrupt
    goto   restorereg      ; et fin d'interrupt

; Interruption TMR2
; -----
intsw8
    bsf    STATUS , RP0      ; sélectionner banque1
    btfss  PIE1 , TMR2IE    ; tester si interrupt autorisée
    goto   intsw9          ; non sauter
    bcf    STATUS , RP0      ; oui, sélectionner banque0

```

```

    btfss PIR1 , TMR2IF    ; oui, tester si interrupt en cours
    goto  intsw9           ; non sauter
    call  inttmr2          ; oui, traiter interrupt
    bcf   PIR1 , TMR2IF    ; effacer flag interrupt
    goto  restorereg       ; et fin d'interrupt

                                ; Interruption TMR1
                                ; -----

intsw9
    bsf   STATUS , RP0     ; sélectionner banque1
    btfss PIE1 , TMR1IE    ; tester si interrupt autorisée
    goto  intswA           ; non sauter
    bcf   STATUS , RP0     ; oui, sélectionner banque0
    btfss PIR1 , TMR1IF    ; oui, tester si interrupt en cours
    goto  intswA           ; non sauter
    call  inttmr1          ; oui, traiter interrupt
    bcf   PIR1 , TMR1IF    ; effacer flag interrupt
    goto  restorereg       ; et fin d'interrupt

                                ; Interruption EEPROM
                                ; -----

intswA
    bsf   STATUS , RP0     ; sélectionner banque1
    btfss PIE2 , EEIE      ; tester si interrupt autorisée
    goto  intswB           ; non sauter
    bcf   STATUS , RP0     ; oui, sélectionner banque0
    btfss PIR2 , EEIF      ; oui, tester si interrupt en cours
    goto  intswB           ; non sauter
    call  inteprom         ; oui, traiter interrupt
    bcf   PIR2 , EEIF      ; effacer flag interrupt
    goto  restorereg       ; et fin d'interrupt

                                ; Interruption COLLISION
                                ; -----

intswB
    bsf   STATUS , RP0     ; sélectionner banque1
    btfss PIE2 , BCLIE     ; tester si interrupt autorisée
    goto  intswC           ; non sauter
    bcf   STATUS , RP0     ; oui, sélectionner banque0
    btfss PIR2 , BCLIF     ; oui, tester si interrupt en cours
    goto  intswC           ; non sauter
    call  intbc            ; oui, traiter interrupt
    bcf   PIR2 , BCLIF     ; effacer flag interrupt
    goto  restorereg       ; et fin d'interrupt

                                ; interruption CCP2
                                ; -----

intswC
    bcf   STATUS , RP0     ; sélectionner banque0
    call  intccp2          ; traiter interrupt
    bcf   PIR2 , CCP2IF    ; effacer flag interrupt

```

Trois remarques :

- Comme c'est précisé dans le fichier source, les flags RCIF et TXIF ne doivent pas être effacés manuellement. Vous ne le pouvez d'ailleurs pas, ils sont en lecture seule. Nous verrons plus tard que ces flags sont en fait remis à 0 automatiquement lors de la lecture du contenu du registre de réception ou de l'écriture du registre d'émission.

- Si vous conservez les lignes « goto restorerreg » dans votre routine d'interruption, alors vous n'arriverez à votre dernier test d'interruption que si tous les autres ont échoué. Dans ce cas, inutile donc de tester la dernière interruption, puisque c'est forcément elle qui a provoqué l'événement. Par contre, si vous décidez de supprimer ces lignes, il vous faudra alors ajouter un test supplémentaire sur la dernière interruption, sans quoi elle serait exécutée à chaque fois.
- Il est inutile de conserver les tests des interruptions que vous n'utilisez pas dans votre programme. Donc, de fait, inutile également de conserver des interruptions qui ne concernent pas votre type de processeur. C'est pour ça que cette routine n'intègre pas l'interruption du port parallèle. Si vous utilisez un 16F877, par exemple, libre à vous de l'ajouter. Je l'ai d'ailleurs fait pour vous dans le fichier « m16F877.asm ».

Pour terminer, nous aurons :

```

                ;restaurer registres
                ;-----
restorerreg
    movf  PCLATH_temp , w      ; recharger ancien PCLATH
    movwf PCLATH              ; le restaurer
    movf  FSR_temp , w        ; charger FSR sauvé
    movwf FSR                 ; restaurer FSR
    swapf status_temp , w     ; swap ancien status, résultat dans w
    movwf STATUS              ; restaurer status
    swapf w_temp , f          ; Inversion L et H de l'ancien W
                                ; sans modifier Z
    swapf  w_temp , w          ; Réinversion de L et H dans W
                                ; W restauré sans modifier status

    retfie                    ; return from interrupt

```

Ce bout de code restaure les registres sauvegardés précédemment. Aussi, si vous avez supprimé des sauvegardes, n'oubliez pas de supprimer également les restaurations.

La dernière ligne sort de la routine d'interruption, et replace le bit GIE à '1'.

Notes : ...

Notes : ...

6. Mise en œuvre et configuration minimale

Et oui, la théorie, c'est bien beau et indispensable, mais si vous désirez programmer des 16F876, c'est bien entendu pour réaliser des montages pratiques.

Nous allons donc commencer ici la partie pratique de l'utilisation de ce nouveau PIC pour vous. Et tout d'abord :

6.1 Le matériel nécessaire

J'ai tout d'abord pensé à créer une platine d'expérimentation universelle, avec programmeur intégré. J'ai cependant renoncé au dernier moment pour les raisons suivantes :

- Cette seconde partie s'adresse à ceux qui ont déjà expérimenté le 16F84, et donc qui ne désirent pas forcément utiliser toutes les possibilités du 16F876, mais désirent utiliser une application particulière.
- Il m'est apparu très difficile de créer une carte qui permette d'exploiter l'intégralité des fonctions des 16F87x, sauf à utiliser jumpers, switches mécaniques et électroniques, qui auraient rendu la carte très lourde à mettre en œuvre, et très onéreuse.
- Tous les magasins spécialisés proposent des platines d'expérimentation d'usage général. Chacun pourra donc trouver la solution qui lui convient le mieux.
- Tout le monde n'a pas envie de se lancer dans la réalisation d'une carte assez complexe, nécessitant la réalisation d'un circuit imprimé

Pour toutes ces raisons, et étant donné que cette seconde partie peut être utilisée comme un manuel de référence, j'ai choisi d'utiliser de petites platines d'expérimentation simples pouvant être réalisées sur une platine d'essais. Ceci permet de disposer de fonctions autonomes claires, et permet au lecteur de réaliser une petite platine centrée sur le sujet qui l'intéresse.

Le matériel minimum pour réaliser la carte de base est donc :

- 1 PIC 16F876 –20 en boîtier DIP
- 1 alimentation de 5V continu. Une simple pile plate de 4.5V peut faire l'affaire.
- 2 supports 28 broches, largeur 0.3'' « tulipe », ou 2 supports 14 pins « tulipe »
- 1 Quartz 20 MHz
- 2 condensateurs non polarisés 15 pF
- 1 platine d'expérimentation à trous, comme pour la première partie du cours

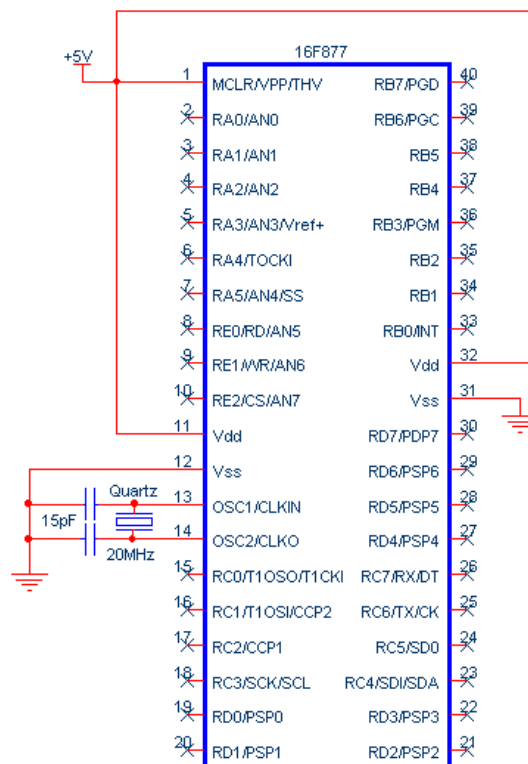
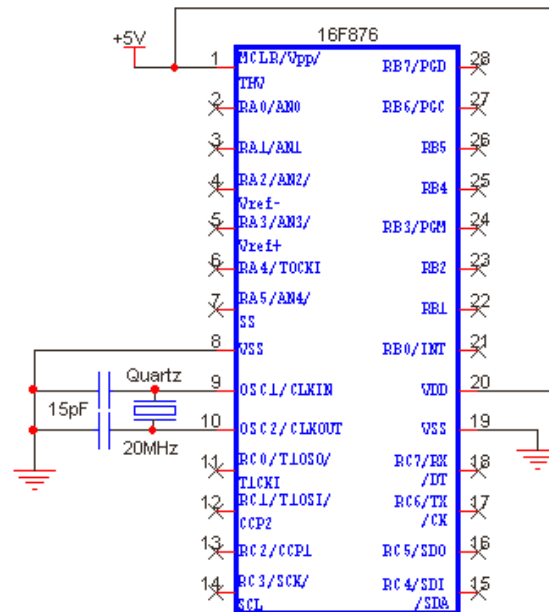
A ceci viendront s'ajouter les différents composants nécessaires à la réalisation des différentes applications.

Au sujet des 2 supports, je vous conseille en effet d'insérer votre PIC dans un des supports, et de le laisser insérer tout au long de vos expérimentations. Il vous suffira

d'insérer ce support dans le support de la platine d'expérimentation, comme si PIC + support se comportaient comme un composant unique. Ainsi, vous épargnerez les pins de votre PIC.

6.2 Le schéma minimum

Voici les schémas minimum permettant de faire fonctionner notre 16F876 ou notre 16F877



Vous pouvez constater que la mise en œuvre du 16F876 (et du 16F877) est similaire à celle du 16F84.

6.3 Les particularités électriques

Vous constatez que sur le schéma concernant le 16F876, vous avez 2 connexions « VSS » qui sont reliées à la masse. En fait, en interne, ces pins sont interconnectées. La présence de ces 2 pins s'explique pour une raison de dissipation thermique. Les courants véhiculés dans le pic sont loin d'être négligeables du fait des nombreuses entrées/sorties disponibles.

Le constructeur a donc décidé de répartir les courants en plaçant 2 pins pour l'alimentation VSS, bien évidemment, pour les mêmes raisons, ces pins sont situées de part et d'autre du PIC, et en positions relativement centrales.

Sur le 16F877, cette procédure a été également étendue à VDD, car encore plus d'entrées/sorties sont disponibles.

Notez que ces PICs fonctionneront si vous décidez de ne connecter qu'une seule de chacune des pins, mais dans ce cas vous vous exposerez à une destruction des composants lors des fortes charges, suite au non respect de la répartition des courants internes.

Pour le reste nous noterons l'habituelle connexion de MCLR au +5V, cette pin étant utilisée pour effectuer un reset du composant en cas de connexion à la masse.

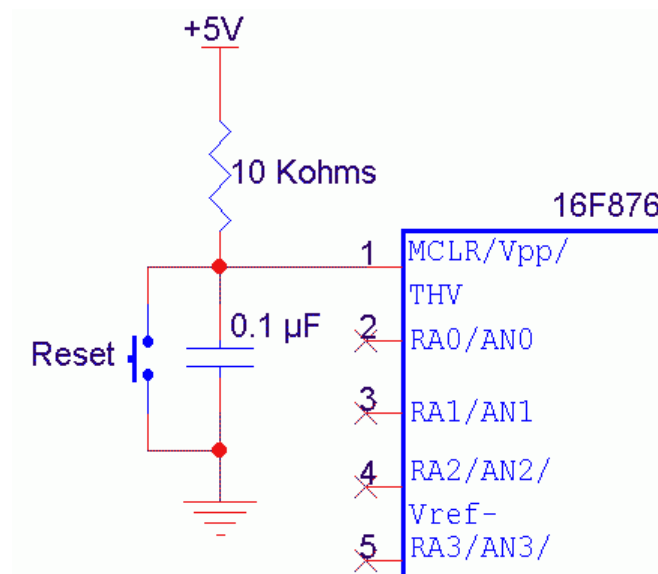
Nous trouvons également le quartz, qui pourra être remplacé par un résonateur ou par un simple réseau RC, de la même manière que pour le 16F84.

Les condensateurs de découplage, du fait de la fréquence plus importante du quartz utilisé ici (20MHz) seront de valeur inférieure à celle utilisée pour le 16F84 (à 4Mhz), soit 15pF.

La tolérance sur ces composants permet d'utiliser d'autres valeurs, mais mon expérience m'a montré que c'est cette valeur qui permet le fonctionnement le plus fiable à cette fréquence. Certains composants ont en effet refusé de fonctionner avec des valeurs de 27pF avec les quartz qui étaient en ma possession. Aucun n'a refusé de fonctionner avec la valeur de 15pF.

Il se peut cependant que vous rencontriez un comportement différent (peu probable), du fait de la provenance de votre propre quartz. En cas de doute, demandez la documentation de celui-ci afin d'adapter les valeurs, ou essayez d'autres valeurs de façon expérimentale.

Si vous avez besoin d'un reset « hardware », le montage suivant est le plus simple et le plus pratique. Le condensateur est facultatif, sauf en environnement perturbé.



Notes : ...

Notes : ...

7. Migration du 16F84 vers le 16F876

7.1 Similitudes et différences avec le 16F84

Nous allons directement préciser quelles sont les similitudes entre le 16F84 et le 16F876, afin de vous montrer à quels chapitres de la première partie vous pouvez vous rapporter sans problème.

En fait, c'est très simple : **pratiquement tout** ce que vous avez vu sur le 16F84 **est identique** pour le 16F876. Ceci inclus :

- le jeu d'instructions
- les modes d'adressage
- L'utilisation des sous-programmes
- le timer 0
- Le mécanisme général des interruptions
- Les niveaux d'imbrication des sous-programmes
- Les limitations dues aux largeurs de bus internes (utilisation de RP0 par exemple)
- L'utilisation du watchdog

En contrepartie, et en sus des fonctions supplémentaires, vous noterez les **différences suivantes** :

- La séparation des registres en 4 banque implique la modification des adresses de ceux-ci
- L'utilisation de 4 banques implique l'utilisation du bit RP1 en plus du bit RP0
- L'utilisation de 4 banques implique l'utilisation du bit IRP pour l'adressage indirect
- L'utilisation d'un programme de plus de 2K implique l'utilisation de **PCLATH**
- L'augmentation des possibilités modifie l'organisation des registres d'interruption
- L'initialisation préalable du registre ADCON1 est nécessaire pour l'utilisation du PORTA
- Les variables (zones RAM disponibles) ne se trouvent pas aux mêmes emplacements

7.2 Conversion d'un programme écrit pour le 16F84 vers le 16F876

En conséquence des points précédents, il vous apparaît donc que les programmes écrits pour 16F84 peuvent être assez facilement portés vers le 16F876.

Mais attention, comme les adresses ne sont plus identiques, **vous devrez impérativement recompiler le fichier source dans MPLAB**. Je vais vous expliquer un peu plus loin comment procéder en partant d'un exemple pratique.

La première chose importante à retenir est donc que :

ON NE PEUT PAS PLACER TEL QUEL DANS UN 16F876 UN FICHER « .HEX » ASSEMBLE POUR UN 16F84.

7.3 Causes de non fonctionnement

Si le programme transféré ne fonctionne plus, il faudra l'examiner avec attention. En effet, il se peut que vous rencontriez un des cas suivants, surtout si le programme a été réalisé sans tenir compte des recommandations que je fais dans la première partie.

1) Vérifiez si des adresses ne sont pas utilisées directement dans le programme

Exemple :

```
movf 0x0C , w ; charger W avec la variable adresse 0x0C
```

Solution : il vous faut déclarer la variable dans la zone des variables, et utiliser son étiquette dans le programme

```
movf mvariable,w ; la variable « mvariable » est déclarée en zone 0x20
```

2) Vérifiez les assignations et définitions d'adresses

Exemple :

```
mvariable EQU 0x0C
```

Solution : modifiez l'assignation ou la définition de façon à la rendre compatible avec le 16F876

Exemple :

```
mvariable EQU 0x020
```

Ou mieux, déclarez la variable de façon classique dans la zone des variables.

3) Vérifiez si des bits non utilisés dans le 16F84 n'ont pas été utilisés en tant que bits libres pour le programmeur

Exemple :

```
bsf PCLATH , 4 ; mettre flag perso à 1
```

Solution : déclarez une variable supplémentaire contenant le flag et utilisez celui-ci dans le programme

Exemple :

```
bsf mesflags , 1 ; mettre flag perso à 1
```

4) Vérifiez si le programme n'utilise pas une astuce concernant les adresses .

Exemple :

```
movlw    0x10    ; pointer sur 0x10
movwf    FSR      ; placer adresse dans pointeur
movf     INDF , w ; charger une variable
bsf      FSR , 2  ; pointer sur 0x14
movwf    INDF     ; copier la valeur
```

Solution : ceci est un peu plus délicat. La procédure utilisée ci-dessus utilise une astuce en manipulant directement les bits du pointeur FSR. Il est évident que si vous changez les adresses des variables, ceci va entraîner un dysfonctionnement du programme.

Dans ce cas vous devrez réorganiser vos variables en conséquence, ou modifier le corps du programme pour tenir compte des nouvelles dispositions.

5) Si les entrées/sorties sur PORTA fonctionnent sur simulateur mais pas en pratique

Vous avez probablement omis d'initialiser le registre **ADCON1**. Notez que le fichier maquette inclus cette gestion. Vous n'avez donc pas à vous en inquiéter pour l'instant.

Evidemment, on considère que les fréquences de fonctionnement des 2 PICs sont identiques.

7.3 Conversion d'un exemple pratique

Afin de mettre ceci en pratique, nous allons « migrer » notre programme « **led** **Tmr1** » pour le porter sur 16F876. Afin de vous faciliter la vie, le fichier original, créé pour le 16F84, est fourni sous la référence « **Led16f84.asm** ».

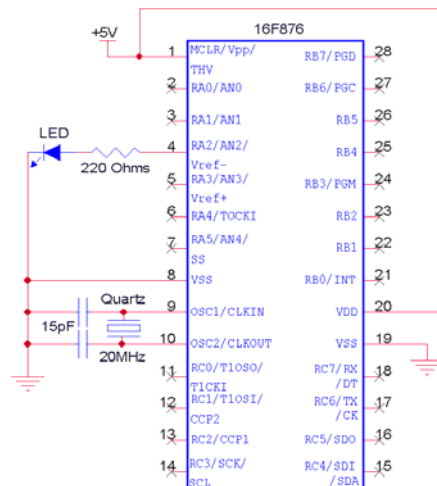
Il existe plusieurs méthodes pour réaliser cette opération. La première qui vient à l'esprit est de créer un nouveau projet et de modifier le fichier original. Cette méthode présente cependant un très fort taux de probabilité d'erreur, surtout pour les gros projets.

Je vais donc vous présenter une méthode plus fiable.

7.3.1 Réalisation du montage

Afin de pouvoir tester notre exercice, nous allons réaliser la platine d'expérimentation correspondante.

Il suffit dans ce cas de réaliser le schéma de base présenté au chapitre 6 et d'y connecter une LED sur RA2. Ceci ne devrait vous poser aucun problème.



7.3.2 Création du projet

Nous allons maintenant créer notre premier vrai projet 16F876. Pour ceci, commencez par dupliquer le fichier « m16F876.asm », et renommez-le « Led_tmr0.asm ».

Ensuite exécutez les étapes suivantes :

- Lancez MPLAB
- Cliquez « new project »
- Choisir le nom « Led_tmr0.pjt » en pointant dans le répertoire des fichiers d'exemple.
- Dans la fenêtre qui s'ouvre, cliquez « change » à côté de « development mode »
- Choisissez « PIC16F876 » dans le menu déroulant « processor »
- Cochez la case « MPLAB SIM Simulator »
- Cliquez « OK »
- Une fenêtre vous indique qu'il n'y a pas de fichier « hex ». Cliquez « OK »
- Confirmez par « OK » les différentes fenêtres d'avertissement.
- Dans la fenêtre « edit project » restée ouverte, cliquez sur « add node »
- Choisissez votre fichier « Led_tmr0.asm »
- Cliquez « OK », la fenêtre « project edit » se ferme, l'écran est vide.

Nous allons maintenant ouvrir le fichier à assembler et l'ancien fichier.

- Cliquez « file->open » et ouvrez le fichier « Led_tmr0.asm »
- Cliquez « file->open » et ouvrez le fichier « Led16F84.asm ».

Maintenant, vous disposez des 2 fichiers sur votre écran, mais rappelez-vous :

Seul le ou les fichiers qui sont précisés dans les nœuds (« node ») du projet sont pris en compte au moment de l'assemblage. Egalement ceux précisés comme inclus dans ces mêmes fichiers (directive « include »).

Par conséquent, 2 fichiers sont bien affichés à l'écran, mais le fichier « Led_16F84.asm » n'a aucune influence sur l'assemblage, car il ne fait pas partie du nœud du projet, et n'est pas non plus utilisé par une directive d'inclusion dans le seul fichier nœud de notre projet (Led_tmr0.asm).

7.3.3 La méthode conseillée

Vous l'aurez déjà compris, la méthode que je trouve la plus sûre, est d'effectuer des « copier/coller » à partir du fichier source vers le fichier cible. Ceci présente l'avantage de ne rien oublier durant les modifications de code.

Commençons donc par remplir nos zones de commentaires.

```
;*****
;   Ce fichier est la base de départ pour une programmation avec      *
;   le PIC 16F876. Il contient les informations de base pour          *
;   démarrer.                                                         *
;                                                                       *
;*****
;
;   NOM:      Led_tmr0                                                *
;   Date:     07/04/2002                                              *
;   Version:  1.0                                                    *
;   Circuit:  platine d'expérimentation                             *
;   Auteur:   Bigonoff                                               *
;                                                                       *
;*****
;
;   Fichier requis: P16F876.inc                                       *
;                                                                       *
;                                                                       *
;                                                                       *
;*****
;   Exercice de migration d'un programme de 16F84 vers le 16F876.    *
;   Clignotement d'une LED à une fréquence de 1Hz en utilisant les   *
;   interruptions du timer0.                                          *
;                                                                       *
;*****
```

Ensuite, intéressons-nous à notre fichier « Led_16F84.asm », et regardons dans l'ordre :
Les 2 premières lignes, à savoir :

```
LIST      p=16F84           ; Définition de processeur
#include <p16F84.inc>        ; Définitions des constantes
```

Ont déjà leur équivalent dans le fichier « Led-tmr0.asm » sous la forme de :

```
LIST      p=16F876          ; Définition de processeur
#include <p16F876.inc>       ; fichier include
```

Nous n'avons donc pas à nous en occuper. Pour rappel, il s'agit de la déclaration du type de processeur, et du nom du fichier contenant les assignations correspondantes à inclure.

Ensuite, nous trouvons la directive **CONFIG** qui définit le mode de fonctionnement du processeur. Nous devons transcrire cette ligne, et non la copier simplement, étant donné que leur structure est différente sur les 2 PICs.

Au niveau du fichier pour le 16F84, nous trouvons :

```
_CONFIG    _CP_OFF & _WDT_OFF & _PWRTE_ON & _HS_OSC
```

Nous notons alors le mode de fonctionnement choisi, à savoir :

- Pas de protection programme en lecture
- Pas de watchdog
- Mise en service du reset retardé à la mise sous tension
- Oscillateur en mode HS.

Nous allons ensuite transposer ce mode de fonctionnement au fichier du 16F876, en complétant les nouveaux modes en fonction de nos besoins. Pour vous faciliter la vie, les commentaires des différents modes de fonctionnement font partie du fichier maquette, donc également, par conséquence, de votre fichier « Led_tmr0.asm ».

Prenons la ligne par défaut, et voyons ce que nous devons changer :

```
_CONFIG _CP_OFF & _DEBUG_OFF & _WRT_ENABLE_OFF & _CPD_OFF & _LVP_OFF &  
_BODEN_OFF & _PWRTE_ON & _WDT_OFF & _HS_OSC
```

Voyons donc les bits dans l'ordre :

CP OFF

Absence de protection du programme en lecture. C'est ce que nous voulions.

DEBUG OFF

Pas de mode debug en service : conservons ce paramètre, puisque nous n'utilisons pas de fonction supplémentaire à celles prévues dans le 16F84.

WRT ENABLE OFF

A conserver pour les mêmes raisons

CPD OFF

Idem

LVP OFF

De même

BODEN OFF

Ne pas se tracasser de tout ceci pour l'instant

PWRTE ON

Laissons ce paramètre comme pour le 16F84, c'est à dire dans sa valeur actuelle

WDT OFF

Pas de watchdog, tout comme pour l'exercice sur le 16F84

HS OSC

Avec un quartz à 20MHz, le tableau nous donne cette valeur pour l'oscillateur.

Vous constatez donc que la méthode est simple :

- Vous appliquez les paramètres du source du 16F84 au fichier du 16F876
- Vous laissez les autres bits dans leur configuration par défaut
- Si vous vouliez ajouter des options ou des modifications, vous corrigez alors éventuellement la directive **CONFIG** en conséquence.

Donc, pour notre cas particulier, il n'y a rien à changer dans notre ligne de directive. Passons donc aux assignations système.

Du côté du 16F84, nous avons :

```
OPTIONVAL EQUH'0087' ; Valeur registre option
; Résistance pull-up OFF
; Préscaler timer à 256
```

```
INTERMASK EQUH'00A0' ; Interruptions sur tmr0
```

Vous constaterez en regardant les datasheets, que le registre **OPTION** a strictement la même configuration pour les 2 PICs, donc vous pouvez dans un premier temps recopier les mêmes valeurs pour le fichier 16F876 de votre exercice.

Par contre, INTERMASK initialisait les interruptions, au niveau du registre **INTCON**. Nous avons vu qu'il y a des changements au niveau des interruptions, donc, pour éviter toute erreur, nous allons reprendre bit par bit les valeurs en question.

Ici, nous avons donc, comme documenté et vérifié, une mise en service des interruptions du timer0. Dans ce cas-ci, vous constaterez que la valeur pour les 2 fichiers sera identique.

Dans le fichier 16F876, j'ai opté pour une valeur sous forme binaire, pour faciliter la compréhension. Donc, pour mettre les interruptions timer0 en service, il suffit de **positionner le bit 5 à « 1 »**. Notez qu'il est inutile de s'occuper de **GIE**, car il sera traité dans la routine d'initialisation.

Les 2 lignes correspondantes pour notre fichier « **Led_tmr0.asm** » deviennent donc (pensez à toujours adapter vos commentaires) :

```
; REGISTRE OPTION_REG (configuration)
; -----
OPTIONVAL EQUB'10000111'
; RBPV      b7 :    1= Résistance rappel +5V hors service
; PSA       b3 :    0= Assignation prédiviseur sur Tmr0
; PS2/PS0   b2/b0   valeur du prédiviseur
;                               111 = 1/128      1/256

; REGISTRE INTCON (contrôle interruptions standard)
; -----
INTCONVAL EQUB'00100000'
; GIE       b7 : masque autorisation générale interrupt
;                               ne pas mettre ce bit à 1 ici
; TOIE      b5 : masque interruption tmr0
```

Du côté du fichier 16f876, vous noterez également les assignations pour les valeurs des registres PIE1 et PIE2. Comme aucune des interruptions utilisées dans ces registres ne sont utilisées ici, vous pouvez les laisser tels quels. Nous verrons également plus loin la fin de cette zone.

Ensuite, nous trouvons notre zone des définitions :

```
; *****
;                               DEFINE                               *
; *****
#define LED PORTA, 2 ; LED
```

Nous supprimons donc l'exemple fourni dans le fichier 16F876 :

```
; exemple
; -----
```

```
#DEFINE LED1 PORTB,1 ; LED de sortie 1
```

et nous le remplaçons par la véritable définition de notre fichier source :

```
#DEFINE LED PORTA,2 ; LED
```

Nous arrivons aux zones de macros, zones qu'il ne faut pas modifier, les macros 16F84 et 16F876 étant différentes. Seules les macros personnelles peuvent être ajoutées après avoir éventuellement été modifiées. Nous n'avons pas de telles macros dans cet exercice.

Nous voici à la zone des variables. Le 16F876 contient plus de banques que le 16F84. Mais, comme le programme provient d'un 16F84, ces banques sont forcément inutilisées.

Cependant, les adresses des zones sont différentes, il ne faudra donc recopier que le contenu des zones, et non leur déclaration.

Du côté 16F84, nous avons :

```
; *****
;                                     *
;                 DECLARATIONS DE VARIABLES
; *****

CBLOCK 0x00C ; début de la zone variables
w_temp :1 ; Sauvegarde du registre W
status temp : 1 ; Sauvegarde du registre STATUS
cmpt : 1 ; compteur de passage
ENDC ; Fin de la zone
```

Ici, il nous faut opérer avec une grande attention. En effet, nous disposons de plusieurs banques de variables dans le 16F876. On pourrait être tenté de tout mettre dans la banque0, mais, en cas de modifications ou d'ajouts ultérieurs, il faut prévoir que certaines variables, principalement **celles utilisées pour la sauvegarde des registres au niveau des interruptions**, doivent être sauveées dans la zone commune.

La première étape consiste donc à **ne pas tenir compte de ces variables**, car elles sont déjà déclarées dans le fichier 16F876. Les autres variables pourront en général être copiées dans la banque0.

Dans ce cas, nous constatons que les variables **w_temp** et **status_temp** sont les variables de sauvegarde déjà déclarées. Nous ne nous en occupons donc pas. La seule variable propre au programme est la variable **cmpt**. Nous copions donc sa déclaration dans la banque0.

Ceci nous donne donc, au niveau de la banque 0, et après avoir supprimé les lignes d'exemples :

```
; *****
;                                     *
;                 VARIABLES BANQUE 0
; *****
; Zone de 80 bytes
; -----
CBLOCK 0x20 ; Début de la zone (0x20 à 0x6F)
cmpt : 1 ; compteur de passage
```

ENDC

; Fin de la zone

Quant à la banque commune, elle conserve les déclarations des variables de sauvegarde. Les autres banques étant inutilisées dans notre exemple pourront donc être supprimées des déclarations.

```
;*****
;                               VARIABLES ZONE COMMUNE                               *
;*****
; Zone de 16 bytes
; -----
CBLOCK 0x70          ; Début de la zone (0x70 à 0x7F)
w_temp : 1           ; Sauvegarde registre W
status_temp : 1      ; sauvegarde registre STATUS
FSR_temp : 1         ; sauvegarde FSR (si indirect en interrupt)
PCLATH_temp : 1      ; sauvegarde PCLATH (si prog>2K)
ENDC
```

Nous trouvons maintenant la routine d'interruption principale. Cette routine dépend du type de PIC utilisé, nous n'y touchons donc pas pour l'instant. Vous allez voir que cette maquette va donc une nouvelle fois vous simplifier la vie.

Vous trouvez ensuite le code correspondant à la routine d'interruption du timer0.

```
;*****
;                               INTERRUPTION TIMER 0                               *
;*****
inttimer
    decfsz cmpt , f      ; décrémente compteur de passages
    return              ; pas 0, on ne fait rien
    movlw b'00000100'    ; sélectionner bit à inverser
    xorwf PORTA , f      ; inverser LED
    movlw 7              ; pour 7 nouveaux passages
    movwf cmpt           ; dans compteur de passages
    return              ; fin d'interruption timer
```

Puisque la gestion principale et les switches vers les sous-routines d'interruption font partie de votre maquette, vous n'avez donc plus qu'à copier tout ce qui suit l'étiquette « inttimer » de votre fichier 16F84 dans l'emplacement prévu de votre fichier 16F876. Ceci nous donne :

```
;*****
;                               INTERRUPTION TIMER 0                               *
;*****
inttmr0
    decfsz cmpt , f      ; décrémente compteur de passages
    return              ; pas 0, on ne fait rien
    movlw b'00000100'    ; sélectionner bit à inverser
    xorwf PORTA , f      ; inverser LED
    movlw 7              ; pour 7 nouveaux passages
    movwf cmpt           ; dans compteur de passages
    return              ; fin d'interruption timer
```

Comme il n'y a pas d'autres interruptions traitées, voici donc terminée la partie interruption. N'est-ce pas simple ? Bien sûr, il reste plein de code inutile, mais le but de cet exercice n'est pas l'optimisation, mais la compréhension des mécanismes de conversion. Une

fois l'exercice terminé, il vous sera loisible, à titre d'exercice personnel, d'optimiser le source obtenu. En attendant, voyons la suite.

Nous en sommes maintenant à la routine d'initialisation. Pour notre 16F84, nous avons :

```

;*****
;                               INITIALISATIONS                               *
;*****

init
    clrf  PORTA      ; Sorties portA à 0
    clrf  PORTB      ; sorties portB à 0
    BANK1            ; passer banque1
    clrf  EEADR       ; permet de diminuer la consommation
    movlw OPTIONVAL   ; charger masque
    movwf OPTION_REG ; initialiser registre option

                    ; Effacer RAM
                    ; -----
    movlw 0x0c        ; initialisation pointeur
    movwf FSR         ; pointeur d'adressage indirect
init1
    clrf  INDF        ; effacer ram
    incf  FSR,f       ; pointer sur suivant
    btfss FSR,6       ; tester si fin zone atteinte (>=40)
    goto  init1       ; non, boucler
    btfss FSR,4       ; tester si fin zone atteinte (>=50)
    goto  init1       ; non, boucler

                    ; initialiser ports
                    ; -----
    bcfLED ; passer LED en sortie
    BANK0            ; passer banque0

    movlw INTERMASK  ; masque interruption
    movwf INTCON     ; charger interrupt control

                    ; initialisations variables
                    ; -----
    movlw 7           ; charger 7
    movwf cmpt        ; initialiser compteur de passages

    goto  start       ; sauter programme principal

```

Nous devons distinguer les différentes parties de cette étape. La première partie est gérée de nouveau par notre fichier maquette. Inutile de s'en préoccuper. Il en va de même pour l'effacement de la zone RAM. Notez que notre fichier maquette n'efface que la zone RAM de la banque0. Si vous souhaitez effacer les autres banques, à vous d'ajouter le code nécessaire. Ce ne sera cependant pas nécessaire ici. Les lignes non surlignées représentent les étapes prises en charge par la routine d'initialisation standard, celles en vert sont des lignes spécifiques au programme.

Nous voyons donc qu'il nous suffit d'ajouter la configuration des entrées/sorties et l'initialisation de notre compteur pour terminer notre routine d'initialisation.

Vous constatez que notre routine d'initialisation 16F876 inclus les initialisations des entrées/sorties, en faisant référence à des assignations. En effet, nous trouvons :


```

init
    ; initialisation PORTS (banque 0 et 1)
    ; -----
    BANK0                ; sélectionner banque0
    clrf PORTA            ; Sorties PORTA à 0
    clrf PORTB            ; sorties PORTB à 0
    clrf PORTC            ; sorties PORTC à 0
    bsf STATUS,RP0       ; passer en banque1
    movlw B'00000110'     ; PORTA en mode digital
    movwf ADCON1          ; écriture dans contrôle A/D
    movlw DIRPORTA        ; Direction PORTA
    movwf TRISA           ; écriture dans registre direction
    movlw DIRPORTB        ; Direction PORTB
    movwf TRISB           ; écriture dans registre direction
    movlw DIRPORTC        ; Direction PORTC
    movwf TRISC           ; écriture dans registre direction

```

Vous constatez que les directions des PORTS sont inclus dans les assignations DIRPORTx, assignations qui sont déclarées dans la zone des assignations système :

```

; DIRECTION DES PORTS I/O
; -----
DIRPORTA EQU B'00111111' ; Direction PORTA (1=entrée)
DIRPORTB EQU B'11111111' ; Direction PORTB
DIRPORTC EQU B'11111111' ; Direction PORTC

```

Il nous suffit donc de transposer notre initialisation « bsf LED » qui place le RA2 en sortie directement dans la zone des assignations système.

Nous modifierons donc la ligne concernée en :

```
DIRPORTA EQU B'00111011' ; RA2 en sortie
```

Restent les lignes :

```

movlw 7 ; charger 7
movwf cmpt ; initialiser compteur de passages

```

Pour travailler proprement, nous déclarons la valeur « 7 » dans une assignation, au niveau de la zone des « définie » par exemple :

```

; *****
;                                     DEFINE                                     *
; *****
#DEFINE LED PORTA,2 ; LED de sortie
#DEFINE CMPTVAL 7 ; valeur de recharge du compteur

```

Notez que nous pouvions également écrire :

```
CMPTVAL EQU 7
```

Qui revient strictement au même dans ce cas.

Nous placerons donc la ligne d'initialisation du compteur dans notre routine d'initialisation, en utilisant notre assignation (ou définition). La routine d'initialisation devient donc :

```

;*****
;
;               INITIALISATIONS
;*****
init

        ; initialisation PORTS (banque 0 et 1)
        ; -----
BANK0      ; sélectionner banque0
clrfs PORTA ; Sorties PORTA à 0
clrfs PORTB ; sorties PORTB à 0
clrfs PORTC ; sorties PORTC à 0
bsf STATUS,RP0 ; passer en banque1
movlw ADCONVAL ; PORTA en mode digital/analogique
movwf ADCON1 ; écriture dans contrôle A/D
movlw DIRPORTA ; Direction PORTA
movwf TRISA ; écriture dans registre direction
movlw DIRPORTB ; Direction PORTB
movwf TRISB ; écriture dans registre direction
movlw DIRPORTC ; Direction PORTC
movwf TRISC ; écriture dans registre direction

        ; Registre d'options (banque 1)
        ; -----
movlw OPTIONVAL ; charger masque
movwf OPTION_REG ; initialiser registre option

        ; registres interruptions (banque 1)
        ; -----
movlw INTCONVAL ; charger valeur registre interruption
movwf INTCON ; initialiser interruptions
movlw PIE1VAL ; Initialiser registre
movwf PIE1 ; interruptions périphériques 1
movlw PIE2VAL ; initialiser registre
movwf PIE2 ; interruptions périphériques 2

        ; Effacer RAM banque 0
        ; -----
bcf STATUS,RP0 ; sélectionner banque 0
movlw 0x20 ; initialisation pointeur
movwf FSR ; d'adressage indirect
init1
clrfs INDF ; effacer ram
incf FSR,f ; pointer sur suivant
btfss FSR,7 ; tester si fin zone atteinte (>7F)
goto init1 ; non, boucler

        ; initialiser variable
        ; -----
movlw CMPTVAL ; charger valeur d'initialisation
movwf cmpt ; initialiser compteur de passages

        ; autoriser interruptions (banque 0)
        ; -----
clrfs PIR1 ; effacer flags 1
clrfs PIR2 ; effacer flags 2
bsf INTCON,GIE ; valider interruptions
goto start ; programme principal

```

Les lignes surlignées en jaune introduisent correspondent à notre initialisation du PORTA. Ces lignes sont inchangées, la modification intervenant dans la zone des assignations système.

Les lignes surlignées en vert correspondent à l'initialisation du compteur suivant le « define » déclaré précédemment.

Cette méthode présente l'avantage de pouvoir changer de valeur sans devoir rechercher toutes les occurrences dans le programme, au risque d'en oublier une. Comme nous avons analysé notre programme, nous savons que cette valeur est également utilisée dans la routine d'interruption du timer 0 :

```
movlw 7 ; pour 7 nouveaux passages
```

Donc, nous modifions également cette ligne en utilisant notre définition :

```
movlw CMPTVAL ; pour CMPTVAL nouveaux passages
```

Ainsi, toute modification de la valeur de recharge du compteur ne nécessitera que l'édition de l'assignation. Ceci évite d'oublier des valeurs en chemin.

Arrivés à ce stade, il ne nous reste plus que notre programme principal :

```
;*****  
;  
;          PROGRAMME PRINCIPAL          *  
;*****  
start  
    goto start ; boucler  
    END
```

Qu'il nous suffise de recopier tel quel vers notre fichier 16F876. Attention cependant à ce que la directive END figure bien en fin de programme.

Lancez l'assemblage en pressant <F10>. Si vous ne vous êtes pas trompés, l'assemblage devrait se dérouler sans problème. Le rapport contenant, comme d'habitude les « warnings » des registres situés dans des banques différentes de 0.

A ce stade, vous pouvez alors fermer votre fichier « Led_16F84 », vous n'en aurez plus besoin. Restez dans la fenêtre de MPLAB uniquement votre fichier « Led_tmr0.asm ». Programmez votre PIC16F876 avec le fichier « Led_tmr0.hex » obtenu et testez le résultat sur votre platine d'expérimentation.

Lancez l'alimentation, vous constatez, si vous ne vous êtes pas trompé que la LED clignote, mais à une fréquence bien plus élevée qu'un clignotement par seconde.

Si vous examinez le programme, vous constaterez que c'est logique, puisque la base de temps est donnée par le timer TMR0. Or ce timer compte en synchronisme avec la fréquence du quartz du PIC.

Le programme était conçu pour un quartz à 4MHz, et maintenant nous utilisons un quartz à 20MHz. La vitesse est donc 5 fois trop importante.

Si vous allez dans la routine d'interruption du timer0, vous verrez que c'est la variable cmpt qui compte le nombre de passages dans tmr0. Une fois cmpt à 0, on inverse la LED et on réinitialise cmpt avec la valeur CMPTVAL.

Souvenez-vous que cette valeur était fixée à 7 dans la zones des définitions. Il suffit donc pour résoudre notre problème, de multiplier CMPTVAL par 5.

Remplacez donc :

```
#DEFINE CMPTVAL 7      ; valeur de recharge du compteur  
par  
#DEFINE CMPTVAL D'35'  ; valeur de recharge du compteur
```

Attention, D'35', et non 35, qui serait interprété comme 0x35, soit 53 décimal.

Réassemblez avec <F10>, reprogrammez le processeur et relancez l'alimentation : La LED clignote à une fréquence de 1Hz. Vous voyez l'intérêt d'avoir utilisé une assignation : plus besoin de rechercher les 2 endroits où on initialise cmpt.

Ne courez pas montrer à votre épouse.... Non, ça je vous l'ai déjà dit dans la première partie.

Le fichier assembleur tel qu'il devrait être à la fin de cet exercice vous est fourni dans le répertoire « fichiers », comme d'habitude, et comme les exercices pratiques de la suite de cet ouvrage.

Notes : ...

Notes : ...

8. Optimisons un peu

Plusieurs d'entre vous, à ce niveau, vont trouver que le programme précédent est un peu gros pour les fonctions qu'il exécute. Si vous êtes dans ce cas, vous n'avez pas tort. C'est pourquoi j'intercale ce chapitre ici pour commencer à vous parler d'optimisation.

Cependant, toutes les méthodes d'optimisation ne peuvent être décrites ici, j'en reparlerai donc tout au long de cette seconde partie.

Et tout d'abord : « **Optimiser** », en voici un bien grand mot. **Pourquoi faire ?**

Et bien, dans ce cas précis, justement, pour rien du tout. Il ne sert strictement à rien d'optimiser ce programme. En effet, ce programme est suffisamment petit pour entrer dans le 16F876 (et même le 16F873). De plus notre programme passe la plupart de son temps à attendre une interruption en exécutant sans fin la boucle du programme principal. Qu'il attende là ou ailleurs, quelle importance ?

Je vous propose juste cette petite optimisation pour commencer à vous habituer à ces techniques pour le cas où elle serait nécessaire, mais tout d'abord, un peu de « théorie appliquée ».

8.1 Les deux grands types d'optimisation

On peut dire qu'une optimisation est destinée à rendre plus efficace un programme. On peut distinguer, dans la plupart des cas, 2 grands type d'optimisations :

- La diminution de la taille du programme
- L'amélioration de la vitesse de traitement ou de réaction à un événement.

Il semble au premier regard que ces 2 techniques soient complémentaires. En effet, la diminution de la taille du programme induit une diminution du nombre de lignes, donc de la durée d'exécution. En y regardant de plus près, vous verrez qu'il n'en est rien.

8.2 Travail sur un exemple concret

Prenons un exemple concret sous la forme d'une portion de programme, parfois appelé « code ». Imaginons un programme qui place la valeur « 5 » dans 20 emplacement mémoires consécutifs :

```
Start
    movlw D'20'           ; pour 20 boucles
    movwf compteur        ; dans variable de comptage
    movlw dest             ; adresse de destination
    movwf FSR              ; dans pointeur indirect
Loop
    movlw 0x05             ; valeur à mettre dans emplacements
    movwf INDF             ; placer valeur 0x05 dans destination
    incf FSR,f             ; pointer sur emplacement suivant
    decfsz compteur,f      ; décrémenter compteur de boucles
```

```
goto loop          ; pas fini, suivant  
nop                ; instruction suivante dans le programme
```

Que pouvons-nous dire de ce bout de « code » ?

En fait, nous voyons que nous avons initialisé 20 variables (dest, dest+1 dest+D'19') avec la valeur « 5 » tout en n'utilisant que 9 instructions.

Vous allez me dire qu'il est impossible d'optimiser ce programme fictif ? Détrompez-vous. Il faut tout d'abord vous rappeler qu'il y a 2 voies d'optimisations, l'optimisation en taille et l'optimisation en vitesse d'exécution.

Si vous regardez ce programme, vous verrez qu'il est difficile de faire plus court. Par contre nous allons voir qu'on peut facilement faire plus rapide, preuve que les 2 voies ne sont pas toujours liées, et même peuvent être opposées.

Supposons que cette portion de code intervienne à un niveau où la vitesse de réaction est critique.

Si vous avez remarqué que le contenu de **W** ne changeait jamais dans la boucle, bravo, vous commencez alors à raisonner en programmeur. Si ce n'est pas encore le cas, aucun problème, c'est en travaillant qu'on devient travailleur, c'est donc en « PICant » qu'on devient « PIColeur » (ne me prenez surtout pas au mot, je ne veux pas d'ennuis avec votre épouse).

Donc, pour rester sérieux (ça change), calculons la vitesse d'exécution (en cycles d'instructions) de notre programme.

- Pour les 4 premières lignes, nous aurons donc 4 cycles.
- Ensuite, les 3 premières lignes de la boucle (3 cycles) seront exécutées 20 fois, car 20 boucles. Ceci nous donne 60 cycles.
- Ensuite, plus compliqué, la condition du « decfsz » sera fausse (pas de saut) les 19 premières fois, et vraie (saut) la 20^{ème} fois. Soit un total de $19+2 = 21$ cycles.
- Vient enfin le « goto » qui produira 19 fois un saut, la 20^{ème} fois il ne sera pas exécuté suite au saut du « decfsz ». Soit donc un total de $19*2 = 38$ cycles.

Rappelez-vous en effet que les instructions qui amènent un saut nécessitent 2 cycles (voir première partie).

Donc, pour résumer, ce programme présente les caractéristiques suivantes :

- Taille : 9 mots de programme
- Temps d'exécution : 123 cycles

Revenons-en à notre remarque sur la valeur inchangée de « W » à l'intérieur de la boucle. Si « W » ne change pas, inutile de répéter son chargement à chaque passage dans la boucle. Il faudra cependant l'initialiser au moins une fois, donc avant l'exécution de cette boucle.

Ceci nous donne :

Start


```

movlw D'20'      ; pour 20 boucles
movwf compteur   ; dans variable de comptage
movlw dest       ; adresse de destination
movwf FSR        ; dans pointeur indirect
movlw 0x05       ; valeur à mettre dans emplacements
Loop
movwf INDF       ; placer valeur 0x05 dans destination
incf FSR,f       ; pointer sur emplacement suivant
decfsz compteur,f ; décrémente compteur de boucles
goto loop        ; pas fini, suivant
nop              ; instruction suivante dans le programme

```

Vous voyez donc que la taille du programme est strictement inchangée, puisqu'on n'a fait que déplacer l'instruction « movlw 0x05 » en dehors de la boucle. Par contre, durant l'exécution du programme, cette instruction ne sera plus exécutée qu'une seule fois, au lieu des 20 fois de l'exemple précédent.

Nous avons donc un gain de 19 cycles, qui nous ramène le temps d'exécution à 104 cycles. Nous avons donc optimisé notre programme au niveau temps d'exécution. La taille restant inchangée, ces 2 méthodes ne sont pas incompatibles.

Maintenant, imaginons que ces 104 cycles soient inacceptables pour notre programme, qui nécessite de réagir beaucoup plus vite. Ecrivons donc tout « bêtement » :

```

Start
movlw 0x05      ; valeur à mettre dans les emplacements
movwf dest      ; placer 0x05 dans emplacement 1
movwf dest+1    ; placer 0x05 dans emplacement 2
movwf dest+2    ; placer 0x05 dans emplacement 3
movwf dest+3    ; placer 0x05 dans emplacement 4
movwf dest+4    ; placer 0x05 dans emplacement 5
movwf dest+5    ; placer 0x05 dans emplacement 6
movwf dest+6    ; placer 0x05 dans emplacement 7
movwf dest+7    ; placer 0x05 dans emplacement 8
movwf dest+8    ; placer 0x05 dans emplacement 9
movwf dest+9    ; placer 0x05 dans emplacement 10
movwf dest+A    ; placer 0x05 dans emplacement 11
movwf dest+B    ; placer 0x05 dans emplacement 12
movwf dest+C    ; placer 0x05 dans emplacement 13
movwf dest+D    ; placer 0x05 dans emplacement 14
movwf dest+E    ; placer 0x05 dans emplacement 15
movwf dest+F    ; placer 0x05 dans emplacement 16
movwf dest+10   ; placer 0x05 dans emplacement 17
movwf dest+11   ; placer 0x05 dans emplacement 18
movwf dest+12   ; placer 0x05 dans emplacement 19
movwf dest+13   ; placer 0x05 dans emplacement 20

```

Que dire de ce programme ? Et bien, calculons ses caractéristiques comme précédemment :

- Taille du programme : 21 mots de programme
- Temps d'exécution : 21 cycles.

Nous constatons donc que nous avons augmenté la taille du programme (donc perdu en « optimisation taille », mais, par contre, nous sommes passé de 104 à 21 cycles de temps d'exécution. Nous avons donc gagné en « optimisation temps ».

Nous voyons donc de façon évidente que les 2 types d'optimisation peuvent nécessiter des solutions opposées. En effet, si j'avais commencé par vous donner ce dernier exemple, la plupart d'entre-vous aurait immédiatement déclaré : « Ce programme n'est pas optimisé ».

En fait, il l'est, mais au niveau du temps, pas au niveau de la taille. Et il est tellement optimisé au niveau temps qu'il est impossible de l'optimiser encore d'avantage. Tout ceci pour vous dire : « méfiez-vous des grands « pros » pour lesquels votre programme n'est jamais assez bien optimisé ».

Si votre programme réalise ce qu'il a à faire dans les temps impartis, alors il est suffisamment optimisé. Dans ce cas, mieux vaut privilégier la lisibilité du code, afin d'en faciliter la maintenance. Je ne suis pas un fan de l'optimisation sans raison.

8.3 Le choix du type d'optimisation

A ce stade, vous allez vous demander comment vous devez procéder en pratique, et quelles procédures utiliser. Il n'y a pas de réponse passe-partout, **il s'agit d'une question de bon sens**. Voici ce que je vous conseille, et que je pratique par moi-même.

En premier lieu, donner priorité à la clarté et à l'« ouverture » du programme. Par « ouverture », j'entends la construction du programme de façon à permettre sa modification ultérieure. Par exemple, le programme précédent, dans sa première version (et sa première amélioration) est plus clair, plus agréable à lire, et plus simple à modifier que la dernière version. Pour changer le nombre de boucles, il suffit dans le premier cas, de modifier la valeur « D'20' ».

En second lieu, et si c'est nécessaire, déterminer le type d'optimisation. Si par exemple, vous manquez de place pour votre programme, ou si vous ne souhaitez pas utiliser plusieurs pages, vous optimiserez la taille de celui-ci. Par contre, si certaines séquences sont critiques en temps d'exécution (par exemple pour certaines interruptions), vous tenterez d'optimiser au niveau temps d'exécution.

Dans tous les cas, placez un commentaire dans votre programme à l'endroit de l'optimisation, pour vous rappeler en cas de modification ultérieure la méthode utilisée et la raison de cette optimisation. En effet, c'est clair que si vous reprenez un jour la dernière version du programme précédent, vous allez vous dire « mais pourquoi ai-je programmé de cette façon ? Optimisons donc ceci en diminuant la taille ». Et crac, vous allez allonger la durée d'exécution, alors qu'il y avait probablement une bonne raison de l'avoir raccourcie.

Rappelez-vous qu'il ne sert à rien de vouloir à tout prix optimiser un programme si le besoin ne s'en fait pas sentir. Préférez de loin un programme clair et bien structuré.

8.4 Application pratique

Nous allons reprendre notre exercice **Led_tmr0** et tenter de l'optimiser au niveau de la clarté et de la taille. Si vous avez compris ce qui précède vous savez qu'il est inutile d'optimiser ce programme, si ce n'est pour en améliorer la clarté.

Je vais effectuer un copier/coller du programme « `Led_tmr0` » en « `Led_opti.asm` ». Ceci afin de conserver dans les exemples les 2 versions du programme. Vous pouvez, en ce qui vous concerne, travailler directement dans votre fichier original.

La première chose qu'il vous vient à l'esprit, en parcourant le source, est qu'il y a plein de définitions, macros, et autres assignations inutiles. Vous pouvez en effet les supprimer, mais ceci ne diminuera en rien la taille du fichier exécutable obtenu ni sa vitesse d'exécution. Rappelez-vous en effet que ce ne sont que de **simples facilités d'écritures**. Ces directives ne sont remplacées par du code que si elles sont utilisées effectivement dans le programme.

Néanmoins, nous allons les supprimer pour rendre le fichier source plus compact et plus rapide à lire. Il ne s'agit donc pas ici d'optimisation, tout au plus peut-on parler de suppression de textes inutiles.

La meilleure façon pour savoir si une assignation, définition ou autre directive de « traitement de texte » est utilisée, est de placer un « ; » devant sa déclaration. On lance ensuite l'assemblage. S'il n'y a pas de message d'erreur, c'est que cette directive n'est pas utilisée dans notre programme. Nous allons en profiter également pour éliminer les commentaires inutiles pour notre application.

Prenons donc le début du programme :

```
;*****
;
;   Exercice d'optimisation en taille d'un programme simple
;   Optimisation sur base du fichier "Led_tmr0"
;
;*****
;
;   NOM:      Led_opti
;   Date:     13/04/2002
;   Version:  1.0
;   Circuit:  platine d'expérimentation
;   Auteur:   Bigonoff
;
;*****
;
;   Fichier requis: P16F876.inc
;
;*****
;   Permet d'éliminer le code superflu
;
;*****
```

```
LIST      p=16F876          ; Définition de processeur
#include <p16F876.inc>       ; fichier include
```

```
__CONFIG __CP_OFF & __DEBUG_OFF & __WRT_ENABLE_OFF & __CPD_OFF & __LVP_OFF &
__BODEN_OFF & __PWRTE_ON & __WDT_OFF & __HS_OSC
```

Ici, nous nous contentons de commenter l'en-tête, comme affiché ci-dessous. Au niveau de la configuration, nous ne changeons rien.

Viennent ensuite tous les commentaires généraux concernant les configurations possibles. Nous ne conservons que les commentaires liés aux choix que nous avons faits, inutile de conserver les autres. Ceci nous donne :

```
;_CP_OFF          Pas de protection
;_DEBUG_OFF       RB6 et RB7 en utilisation normale
;_WRT_ENABLE_OFF  Le programme ne peut pas écrire dans la flash
;_CPD_OFF         Mémoire EEprom déprotégée
;_LVP_OFF         RB3 en utilisation normale
;_BODEN_OFF       Reset sur chute de tension hors service
;_PWRTE_ON        Démarrage temporisé
;_WDT_OFF         Watchdog hors service
;_HS_OSC          Oscillateur haute vitesse (4Mhz<F<20Mhz)
```

Notez qu'il ne s'agit ici que de commentaires (précédés par « ; »). Libre à vous donc de tout supprimer, ce que je ne vous conseille cependant pas pour des raisons de facilité de maintenance.

Ensuite, nous trouvons les assignations système. Si vous tentez de les faire précéder d'un « ; », vous obtiendrez des erreurs au moment de l'assemblage. Ceci est normal, car le fichier maquette qui nous a servi de base de travail utilise ces assignations. Nous verrons donc ceci plus tard.

Nous arrivons dans la zone de macros. Si vous faites des essais ou des recherches, vous vous apercevrez que la seule macro utilisée dans cet exercice est la macro « BANK0 ». Vous pouvez donc franchement supprimer les autres.

```
; *****
;
;                               MACRO                               *
; *****

; Changement de banques
; -----

BANK0 macro                ; passer en banque0
    bcf STATUS,RP0
    bcf STATUS,RP1
endm
```

Notez que ceci ne constitue pas une optimisation, puisque le code de la macro ne se retrouve qu'aux endroits où elle est utilisée.

Venons-en maintenant à la routine d'interruption principale. Ici nous commençons réellement l'optimisation en taille (et en vitesse). La première chose à laquelle on pourrait penser est d'éliminer toutes les vérifications concernant les interruptions non utilisées dans ce programme. Procédons donc à cette épuration, sachant que nous n'utilisons que l'interruption tmr0 :

```
; *****
;
;                               ROUTINE INTERRUPTION                *
; *****

;sauvegarder registres
;-----
org    0x004                ; adresse d'interruption
```

```

movwf w_temp          ; sauver registre W
swapf STATUS,w        ; swap status avec résultat dans w
movwf status_temp     ; sauver status swappé
movf FSR , w          ; charger FSR
movwf FSR_temp        ; sauvegarder FSR
movf PCLATH , w       ; charger PCLATH
movwf PCLATH_temp     ; le sauver
clrf PCLATH           ; on est en page 0
BANK0                 ; passer en banque0

; Interruption TMR0
; -----

btfsc INTCON,T0IE      ; tester si interrupt timer autorisée
btfss INTCON,T0IF      ; oui, tester si interrupt timer en cours
goto intsw1            ; non test suivant
call inttmr0           ; oui, traiter interrupt tmr0
bcf INTCON,T0IF        ; effacer flag interrupt tmr0
goto restorereg        ; et fin d'interruption

; restaurer registres
; -----
restorereg
movf PCLATH_temp , w ; recharger ancien PCLATH
movwf PCLATH         ; le restaurer
movf FSR_temp , w    ; charger FSR sauvé
movwf FSR            ; restaurer FSR
swapf status_temp,w  ; swap ancien status, résultat dans w
movwf STATUS         ; restaurer status
swapf w_temp,f        ; Inversion L et H de l'ancien W
                        ; sans modifier Z
swapf w_temp,w        ; Réinversion de L et H dans W
                        ; W restauré sans modifier status
retfie               ; return from interrupt

```

Si vous assemblez ceci, vous aurez un message d'erreur du type :

Symbol not previously defined (intsw1)

En fait, ceci est logique, car vous avez supprimé la ligne où devait s'effectuer le saut en cas d'une interruption non provoquée par tmr0 :

```
goto intsw1 ; non test suivant
```

Comme il n'y a pas de test suivant, vu qu'il n'y a qu'une seule interruption utilisée, ce test devrait pointer sur « restorereg ». Remplacez donc cette ligne par :

```
goto restorereg ; non test suivant
```

A ce moment, cela fonctionne, mais n'a plus guère de sens, vu que nous aurons un algorithme du type : Si interruption non provoquée par tmr0, sauter en restorereg, sinon, traiter interruption timer0 puis sauter en restorereg. C'est logique, vu que la seule façon d'accéder à cette partie de code, c'est qu'il y ait eu interruption. Or, la seule interruption autorisée est l'interruption tmr0. Inutile donc d'effectuer un test.

Nous aurons donc :

```
        ; Interruption TMR0
        ; -----
call    inttmr0      ; traiter interrupt tmr0
bcf     INTCON,T0IF  ; effacer flag interrupt tmr0
goto    restaurereg  ; et fin d'interruption
```

Nous pouvons donc aller plus loin, en constatant qu'il devient inutile pour la lisibilité d'inclure un appel vers une sous-routine. Autant écrire dans ce cas le code directement dans la routine principale d'interruption. Déplaçons donc le contenu de la routine `inttmr0` à la place de l'appel de cette sous-routine :

```
        ; Interruption TMR0
        ; -----
decfsz  cmpt , f     ; décrémenter compteur de passages
return  ; pas 0, on ne fait rien
movlw   b'00000100'  ; sélectionner bit à inverser
xorwf   PORTA , f    ; inverser LED
movlw   CMPTVAL      ; pour CMPTVAL nouveaux passages
movwf   cmpt         ; dans compteur de passages
bcf     INTCON,T0IF  ; effacer flag interrupt tmr0
goto    restaurereg  ; et fin d'interruption
```

Ne pas, bien évidemment, déplacer le « `return` » (ni l'étiquette « `inttmr0` »).

Nous pouvons maintenant supprimer toutes les structures des différentes interruptions, structures qui ne contiennent que des « `return` ». Elles ne seront jamais appelées, et, du coup, parfaitement inutiles et consommatrices d'espace (pas de temps).

Continuons à examiner notre routine d'interruption en l'examinant depuis le début. Nous voyons que plusieurs registres sont sauvegardés. **Rappelez-vous qu'une sauvegarde est indispensable si les 2 conditions suivantes sont simultanément remplies :**

- **Le registre sauvegardé doit être utilisé dans le programme principal**
- **Le registre sauvegardé doit être modifié dans la routine d'interruption**

Si nous examinons le programme à partir du moment où les interruptions sont en service, c'est à dire à partir de la ligne « `bsf INTCON,GIE` », vous voyez que le programme comporte uniquement 2 sauts.

Voyons maintenant les registres à sauvegarder effectivement :

Le premier registre est le registre « **W** ». Dans ce cas précis, et extrêmement rare, il n'est pas utilisé dans le programme principal, il n'y a donc pas de raison de le sauvegarder.

Ensuite, nous trouvons la sauvegarde du registre **STATUS**. Comme notre programme principal ne contient aucune utilisation de ce registre, il n'est donc pas nécessaire de le sauvegarder. Ce cas précis est également extrêmement rare, étant donné que **STATUS** est utilisé pour les changements de banques (**RP0,RP1**), pour les tests (`btfsx STATUS,x`), pour les opérations de rotation (utilisation du carry) etc.

Nous pouvons dire que pour la presque totalité des programmes, les registres « **W** » et **STATUS** seront systématiquement sauvegardés. Mais, étant donné que le but de ce chapitre est l'optimisation, nous optimiserons donc.

Continuons avec la sauvegarde de **FSR**. Notre programme principal n'utilise pas l'adressage indirect, pas plus que notre routine d'interruption. Donc, 2 bonnes raisons de supprimer cette sauvegarde.

Maintenant, voyons **PCLATH** dont la sauvegarde est également inutile, vu que notre programme tient en page0, et n'utilise pas de tableaux mettant en œuvre **PCLATH**. A supprimer donc également.

Il est évident que la suppression de la sauvegarde inclus la suppression de la restauration de ces registres. Autre conséquence, cette suppression inclus également la suppression des variables utilisées pour cette sauvegarde. Voyons ce qu'il reste de notre routine d'interruption :

```
org 0x004                ; adresse d'interruption
clrf PCLATH              ; on est en page 0
BANK0                   ; passer en banque0

    ; Interruption TMR0
    ; -----
decfsz cmpt , f         ; décrémenter compteur de passages
return                 ; pas 0, on ne fait rien
movlw b'00000100'      ; sélectionner bit à inverser
xorwf PORTA , f        ; inverser LED
movlw CMPTVAL          ; pour CMPTVAL nouveaux passages
movwf cmpt             ; dans compteur de passages

bcf INTCON,T0IF        ; effacer flag interrupt tmr0
goto restorereg        ; et fin d'interruption

    ; restaurer registres
    ; -----
restorereg
retfie                 ; return from interrupt
```

On peut dire qu'elle a sensiblement maigri. Néanmoins, regardons-la en détail. Nous constatons que la première ligne place **PCLATH** à 0. Or, si on regarde le datasheet de Microchip, on constate que ce registre est mis à 0 lors d'un reset. Comme il reste inchangé durant l'exécution de notre programme, nous pouvons nous passer de cette ligne. Nous pouvons raisonner de façon identique pour **RP0** et **RP1**, forcés à 0 par la macro « **BANK0** », qui est de ce fait inutile, et peut être supprimée de la routine d'interruption. En conséquence, n'étant plus utilisée nulle part, nous pouvons nous passer également de sa déclaration.

Regardons attentivement, et nous constatons la présence de la ligne « goto restorereg », qui n'a plus aucune utilité, étant donné que l'adresse de saut suit l'adresse de l'instruction. C'est donc un saut qui n'en est plus un. Supprimons donc cette ligne. Il nous reste :

```

;*****
;
;                               ROUTINE INTERRUPTION                               *
;*****

    org 0x004      ; adresse d'interruption

        ; Interruption TMR0
        ; -----
    decfsz    cmpt , f      ; décrémenteur compteur de passages
    return    ; pas 0, on ne fait rien
    movlw     b'00000100'   ; sélectionner bit à inverser
    xorwf     PORTA , f     ; inverser LED
    movlw     CMPTVAL       ; pour CMPTVAL nouveaux passages
    movwf     cmpt          ; dans compteur de passages

    bcf       INTCON,T0IF   ; effacer flag interrupt tmr0
    retfie    ; return from interrupt

```

Voici notre programme fortement simplifié. Lancez l'assemblage, placez le code dans votre PIC, et vous constatez... que ça ne fonctionne plus. Pourquoi ?

En fait, si vous simulez en pas à pas, vous constaterez que l'interruption n'a lieu qu'une seule fois. En effet, l'instruction « **return** » effectue une sortie de la routine d'interruption **SANS remettre le bit GIE à 1**. Il n'y aura donc pas de nouvelle interruption. De plus, le flag T0IF ne sera pas resetté lors de la sortie de la routine d'interruption.

Cette instruction se trouvait à l'origine dans la sous-routine « inttmr0 » appelée depuis la routine d'interruption principale. Elle permettait de revenir dans cette routine, qui, elle, se terminait par « retfie » après un reset du flag T0IF. Ce n'est plus le cas ici.

Il vous faut donc remplacer l'instruction « return » par un saut à l'endroit voulu, qui permet de rétablir la situation d'origine.

Notre routine d'interruption finale est donc :

```

;*****
;
;                               ROUTINE INTERRUPTION                               *
;*****

    org 0x004      ; adresse d'interruption

        ; Interruption TMR0
        ; -----
    decfsz    cmpt , f      ; décrémenteur compteur de passages
    goto      fin          ; pas 0, sortir après reset du flag
    movlw     b'00000100'   ; sélectionner bit à inverser
    xorwf     PORTA , f     ; inverser LED
    movlw     CMPTVAL       ; pour CMPTVAL nouveaux passages
    movwf     cmpt          ; dans compteur de passages
fin
    bcf       INTCON,T0IF   ; effacer flag interrupt tmr0
    retfie    ; return from interrupt

```

Voici notre routine d'interruption fortement allégée. Voyons maintenant notre routine d'initialisation.

La ligne suivante peut être supprimée, comme expliqué précédemment :

```
BANK0          ; sélectionner banque0
```

Ensuite, les effacements préventifs des PORTs peuvent également être supprimés pour cette application :

```
clrf  PORTA      ; Sorties PORTA à 0
clrf  PORTB      ; sorties PORTB à 0
clrf  PORTC      ; sorties PORTC à 0
```

Par contre, les lignes suivantes sont absolument indispensables, comme je l'ai déjà expliqué, et comme nous le verrons en détail dans le chapitre sur les conversions analogiques/numériques.

```
movlw  ADCON1VAL ; PORTA en mode digital/analogique
movwf  ADCON1    ; écriture dans contrôle A/D
```

Ensuite les lignes d'initialisation des directions des registres peuvent être modifiées :

```
movlw  DIRPORTA    ; Direction PORTA
movwf  TRISA        ; écriture dans registre direction
movlw  DIRPORTB    ; Direction PORTB
movwf  TRISB        ; écriture dans registre direction
movlw  DIRPORTC    ; Direction PORTC
movwf  TRISC        ; écriture dans registre direction
```

En effet, les ports sont tous placés en entrée lors d'un reset, inutile donc de les y forcer, d'autant que seul le PORTA est utilisé. De plus, comme une seule sortie est utilisée, il suffit de forcer le bit correspondant à 0 :

```
bcf    LED          ; placer RA2 en sortie
```

En effet, **LED** va être remplacé par « **PORTA,2** ». Comme on pointe sur la banque1, du fait de la première ligne, ceci sera exécuté comme « **TRISA,2** ». Si vous doutez, relisez vite la première partie.

Plus loin, nous trouvons les initialisations des registres d'interruption. une seule interruption est utilisée ici, qui n'utilise pas les interruptions périphériques. Comme **INTCON** est mis à 0 également lors d'un reset, nous pourrions nous limiter à placer le bit d'autorisation correspondant à 1. La suppression des valeurs inutilisées permet également de supprimer les définitions correspondantes :

```
          ; registres interruptions (banque 1)
          ; -----
bsf      INTCON,T0IE ; interruptions tmr0 en service
```

Vient l'effacement de la RAM, qui n'est pas nécessaire ici, vu que ces emplacements ne sont pas utilisés. Supprimons-le.

L'initialisation de la variable utilisée(cmnt) n'est utile que pour la durée du premier allumage de la LED. Comme la valeur à l'initialisation est aléatoire, on pourrait avoir un

premier allumage après une durée allant au maximum jusque 8 secondes. Bien que ce ne soit pas gênant, conservons cette initialisation.

Les 2 reset de PIRx ne sont pas nécessaires non plus, donc on supprime. Quant au « goto start » il est de fait inutile, car l'adresse de saut suit directement le saut.

Voici donc le programme épuré, pour ne pas dire « optimisé ». Assemblez-le et vérifiez son fonctionnement correct.

```

LIST      p=16F876          ; Définition de processeur
#include <p16F876.inc>      ; fichier include

__CONFIG  _CP_OFF & _DEBUG_OFF & _WRT_ENABLE_OFF & _CPD_OFF & _LVP_OFF &
_BODEN_OFF & _PWRTE_ON & _WDT_OFF & _HS_OSC

;_CP_OFF          Pas de protection
;_DEBUG_OFF       RB6 et RB7 en utilisation normale
;_WRT_ENABLE_OFF  Le programme ne peut pas écrire dans la flash
;_CPD_OFF         Mémoire EEprom déprotégée
;_LVP_OFF         RB3 en utilisation normale
;_BODEN_OFF       Reset tension hors service
;_PWRTE_ON        Démarrage temporisé
;_WDT_OFF         Watchdog hors service
;_HS_OSC          Oscillateur haute vitesse (4Mhz<F<20Mhz)

;*****
;                               ASSIGNATIONS SYSTEME                               *
;*****
; REGISTRE OPTION_REG (configuration)
; -----
OPTIONVAL EQUB'10000111'
; RBPV          b7 : 1= Résistance rappel +5V hors service
; PSA           b3 : 0= Assignment prédiviseur sur Tmr0
; PS2/PS0       b2/b0 valeur du prédiviseur
;
;               111 = 1/128      1/256

; REGISTRE ADCON1 (ANALOGIQUE/DIGITAL)
; -----
ADCON1VAL EQUB'00000110' ; PORTA en mode digital

;*****
;                               DEFINE                               *
;*****
#define LED PORTA,2      ; LED de sortie
#define CMPTVAL D'35'    ; valeur de recharge du compteur

;*****
;                               VARIABLES BANQUE 0                               *
;*****
; Zone de 80 bytes
; -----
CBLOCK 0x20          ; Début de la zone (0x20 à 0x6F)
cmpt : 1             ; compteur de passage
ENDC                ; Fin de la zone

;*****
;                               VARIABLES ZONE COMMUNE                               *
;*****
; Zone de 16 bytes

```

```

; -----
; CBLOCK 0x70          ; Début de la zone (0x70 à 0x7F)
; ENDC

; //////////////////////////////////////
;                               I N T E R R U P T I O N S
; //////////////////////////////////////
; *****
;                               DEMARRAGE SUR RESET
; *****
; org 0x000             ; Adresse de départ après reset
; goto    init          ; Initialiser

; *****
;                               ROUTINE INTERRUPTION
; *****
; org 0x004             ; adresse d'interruption

;                               ; Interruption TMR0
;                               ; -----
; decfsz cmpt , f       ; décrémente compteur de passages
; goto    fin           ; pas 0, on sort de l'interruption
; movlw   b'00000100'   ; sélectionner bit à inverser
; xorwf   PORTA , f     ; inverser LED
; movlw   CMPTVAL        ; pour CMPTVAL nouveaux passages
; movwf   cmpt           ; dans compteur de passages
fin
; bcf     INTCON,T0IF    ; effacer flag interrupt tmr0
; retfie                ; return from interrupt

; //////////////////////////////////////
;                               P R O G R A M M E
; //////////////////////////////////////
; *****
;                               INITIALISATIONS
; *****
init
;                               ; initialisation PORTS (banque 0 et 1)
;                               ; -----
; bsf     STATUS,RP0    ; passer en banque 1
; movlw   ADCON1VAL     ; PORTA en mode digital/analogique
; movwf   ADCON1        ; écriture dans contrôle A/D
; bcf     LED            ; placer RA2 en sortie

;                               ; Registre d'options (banque 1)
;                               ; -----
; movlw   OPTIONVAL     ; charger masque
; movwf   OPTION_REG    ; initialiser registre option

;                               ; registres interruptions (banque 0)
;                               ; -----
; bsf     INTCON,T0IE    ; interruptions tmr0 en service

;                               ; initialiser variable
;                               ; -----
; movlw   CMPTVAL        ; charger valeur d'initialisation
; movwf   cmpt           ; initialiser compteur de passages

;                               ; autoriser interruptions (banque 0)
;                               ; -----
; bsf     INTCON,GIE     ; valider interruptions

```

```

;*****
;
;               PROGRAMME PRINCIPAL
;*****
start
    goto start      ; boucler
    END             ; directive fin de programme

```

Si vous regardez attentivement, vous voyez que dans nos suppressions, nous avons effacé la ligne qui permettait de repointer sur la banque 0. Replaçons donc cette ligne à l'endroit voulu :

```

        ; initialisation PORTS (banque 1)
        ; -----
bsf     STATUS,RP0    ; passer en banque1
movlw   ADCON1VAL     ; PORTA en mode digital/analogique
movwf   ADCON1        ; écriture dans contrôle A/D
bcf     LED           ; placer RA2 en sortie

        ; Registre d'options (banque 1)
        ; -----
movlw   OPTIONVAL     ; charger masque
movwf   OPTION_REG    ; initialiser registre option
bcf     STATUS,RP0    ; repasser en banque 0

        ; registres interruptions (banque 0)
        ; -----
bsf     INTCON,T0IE   ; interruptions tmr0 en service

```

8.4 Optimisations particulières

Nous venons de voir les 2 grands types d'optimisation, mais il en existe d'autres, liées à des cas particuliers. Ces types d'optimisation peuvent être par exemple :

- L'optimisation des ressources matérielles externes (par exemple, comment diminuer le nombre d'I/O nécessaires pour le fonctionnement du projet). Un cas typique est l'utilisation du multiplexage des pins utilisées.
- L'optimisation des ressources matérielles internes (par exemple, diminution du nombre de timers utilisés).
- L'optimisation des niveaux de sous-programme (pour ne pas dépasser les 8 niveaux maximum autorisés).
- L'optimisation permise par un arrangement étudié des ressources

Comme pour le cas précédent, toutes ces optimisations ne sont pas forcément compatibles. Il est clair que, par exemple, utiliser le multiplexage des pins va nécessiter plus de code et plus de temps d'exécution.

Il est une fois de plus question de priorité. Retenez donc que :

Il n'y a pas une seule optimisation, il y a toujours plusieurs optimisations parfois contradictoires. Parler d'optimiser un programme est donc incorrect ou incomplet. Il faut préciser de quel type d'optimisation on parle.

Il y a donc probablement d'autres types d'optimisation, mais le but de ce chapitre est de vous faire comprendre le raisonnement à appliquer, et non de tout détailler de façon exhaustive.

8.4.1 Optimisation des niveaux de sous-programmes

Je vais vous toucher un mot des méthodes pour diminuer le nombre de niveaux de sous-routines utilisés dans vos programmes. En effet, je reçois pas mal de courrier suite à la première partie du cours concernant les sous-routines, principalement des erreurs de « stack ».

Tout d'abord, rappelez-vous qu'il ne faut pas confondre niveaux de sous-routines et nombre de sous-routines. Les premiers sont limités à 8, le second n'est limité que par la capacité mémoire programme de votre pic.

Pour déterminer le niveau de sous-routine utilisé dans votre programme, il suffit de le parcourir en ajoutant 1 à chaque call rencontré, et en soustrayant 1 à chaque return rencontré. Le nombre maximal atteint lors de ce comptage détermine le niveau maximum de sous-programme utilisé par votre programme.

N'oubliez cependant pas d'ajouter 1 à ce nombre maximum, si vous utilisez les interruptions, augmenté du niveau de sous-programme utilisé dans la routine d'interruption.

Pour rester concrets, imaginons un programme fictif. Ce programme est incomplet, n'essayez donc pas de le faire tourner, ou alors ajoutez ce qu'il manque.

```
goto    start        ; programme
;*****
;                               ROUTINE INTERRUPTION                               *
;*****

        ;sauvegarder registres
        ;-----
org     0x004         ; adresse d'interruption
movwf  w_temp         ; sauver registre W
swapf  STATUS,w       ; swap status avec résultat dans w
movwf  status_temp    ; sauver status swappé

        ; Interruption TMR0
        ; -----
btfsc  INTCON,T0IE     ; tester si interrupt timer autorisée
btfss  INTCON,T0IF     ; oui, tester si interrupt timer en cours
goto   intsw1          ; non test suivant
call   inttmr0         ; oui, traiter interrupt tmr0
bcf    INTCON,T0IF     ; effacer flag interrupt tmr0
goto   restorereg      ; et fin d'interruption

        ; Interruption RB0/INT
        ; -----
intsw1
call   intrb0          ; oui, traiter interrupt RB0
```

```

    bcf    INTCON,INTF      ; effacer flag interrupt RB0

    ;restaurer registres
    ;-----
restorereg
    swapf  status_temp,w    ; swap ancien status, résultat dans w
    movwf  STATUS           ; restaurer status
    swapf  w_temp,f         ; Inversion L et H de l'ancien W
    ; sans modifier Z
    swapf  w_temp,w         ; Réinversion de L et H dans W
    retfie                  ; return from interrupt

;*****
;                               INTERRUPTION TIMER 0                               *
;*****
inttmr0
    incf   compteur,f       ; incrémenter compteur
    return ; fin d'interruption timer

;*****
;                               INTERRUPTION RB0/INT                               *
;*****
intrb0
    movf   mvariable        ; charger mvariable
    call   divis            ; diviser par 2
    return ; fin d'interruption RB0/INT

;*****
;                               TRAITER REGISTRE                               *
;*****
traitreg
    movwf  mvariable        ; mettre w dans mvariable
    call   multipli         ; multiplier par 2
    return ; retour

;*****
;                               Diviser par 2                               *
;*****
divis
    bcf    STATUS,C         ; effacer carry
    rrf    mvariable,f      ; diviser mvariable par 2
    return ; retour

;*****
;                               Multiplier par 2                               *
;*****
multipli
    bcf    STATUS,C         ; effacer carry
    rlf    mvariable,f      ; multiplier mvariable par 2
    return ; retour

;*****
;                               Ajouter 1                               *
;*****
ajout
    incf   mvariable,f      ; ajouter 1
    return ; retour

;*****
;                               programme principal                               *
;*****
start
    movlw  0x05             ; 5 dans w

```

```

call  traitreg      ; mettre le double dans mavariable
call  ajout         ; incrémenter mavariable
goto  start         ; boucler
END                ; directive fin de programme

```

Etudions donc les niveaux de sous-routines de ce petit programme, du reste parfaitement inutile. Commençons par le programme principal :

- On commence par le niveau 0. En première ligne, on trouve « goto start ». Il s'agit d'un saut, non d'un appel de sous-routine, donc on n'ajoute rien
- On arrive à l'étiquette « start ». On avance dans le programme, et on tombe sur la ligne « call traitreg ». Il s'agit d'un call, donc on ajoute 1. On est donc à 1 niveau.
- On poursuit à l'étiquette « traitreg » et on avance dans la sous-routine. On arrive à la ligne « call multipli ». On ajoute donc 1, nous voici à 2. Poursuivons donc à l'adresse « multipli ».
- On déroule le programme, et on arrive à « return ». On décompte donc 1. Nous revoici à 1. Retour donc dans la routine traitreg, et plus précisément à la ligne « return ». On décompte de nouveau 1. Nous revoici à 0, et dans le programme principal, preuve que tout se passe bien.
- On poursuit le programme principal et on arrive à la ligne « call ajout ». On ajoute 1, ce qui nous fait 1. On passe à l'étiquette « ajout ». On avance dans le programme et on trouve la ligne « return ». On soustrait 1, ce qui nous fait 0, et retour au programme principal. On démarre de 0, on termine à 0, tout est correct.

Rappelez-vous que si, à un moment donné, vous obtenez une valeur supérieure à 8 ou inférieure à 0, vous avez forcément fait une erreur dans la construction de votre programme.

En général il s'agit d'une méprise entre « goto » et « call », ou un oubli ou ajout d'un « return ».

Nous voyons donc que la valeur maximale atteinte lors de l'exécution de notre programme est de 2. Donc, notre programme principal utilise un niveau de sous-programme de 2.

Etant donné qu'une interruption peut avoir lieu à n'importe quel moment de notre programme (et donc au moment où nous sommes dans la sous-routine multipli, de niveau 2), il convient d'ajouter à ce niveau ceux utilisés dans les routines d'interruption.

Imaginons donc qu'une interruption intervienne au moment où nous nous trouvons dans le cas le plus défavorable, c'est à dire au niveau 2 :

- L'interruption a lieu, et nécessite une sauvegarde sur la pile (stack), comme expliqué dans la première partie. Nous étions à 2, nous voici à 3.
- Nous voici dans la routine d'interruption, nous arrivons à la ligne « call inttmr0 ». Nous ajoutons donc 1, nous en sommes à 4.

- Dans la sous-routine « `inttmr0` », nous trouvons « `return` », donc nous revoici à 3, et dans la routine d'interruption principale.
- Nous trouvons ensuite « `call intrb0` », ce qui nous ramène à 4, nous voici dans « `intrb0` ».
- Nous trouvons ensuite « `call divis` » qui nous amène à 5 dans la sous-routine « `divis` ».
- Dans cette routine, nous trouvons « `return` », nous revoici à 4 dans la suite de la routine « `intrb0` ».
- Nous poursuivons et nous trouvons un « `return` ». Nous voici à 3 dans la routine d'interruption principale, qui se termine par « `retfie` », qui est un « `return` » particulier.
- Nous sortons donc de la routine d'interruption et nous faisons -1, ce qui nous ramène au niveau 2 du départ. De nouveau, preuve que tout s'est bien passé.

Nous voyons donc que notre programme complet nécessite une profondeur de niveau de sous-routines de 5 (valeur maximale rencontrée). Nous sommes donc en dessous des 8 niveaux permis, tout est bon.

Mais tout n'est pas toujours si rose, aussi, si votre programme dépasse les 8 niveaux permis, je vais vous donner quelques astuces pour en diminuer la valeur.

La première chose à faire, c'est de regarder si toutes les sous-routines sont utiles. En effet, une sous-routine appelée une seule fois dans un programme n'est pas vraiment indispensable. Mais ça, tout le monde peut le faire, il ne s'agit pas d'optimisation.

Tout d'abord un préambule : En général, l'optimisation des niveaux de sous-routines diminue la clarté du programme, et sont donc à réserver aux applications pour lesquelles on n'a pas le choix. Principalement pour éviter le débordement de la pile, bien entendu (niveaux >8), mais également parfois pour gagner du temps d'exécution.

Dans tous les autres cas, ne tentez pas d'optimiser les niveaux de sous-programmes.

Je ne sais pas si c'est vraiment une bonne idée de parler de ce type d'optimisation, mais comme je reçois beaucoup de courrier à ce sujet, je m'y suis finalement décidé.

Examinons tout d'abord la routine d'interruption principale. Si vous regardez attentivement, vous distinguez que les sous-routines « `inttmr0` » et « `intrb0` » ne sont appelées qu'à partir d'un seul endroit dans le programme. Ce sera en général vrai pour tous les programmes utilisant la maquette que je vous fournis. Donc, partant de là, il est possible de gagner un niveau de sous-routine à cet endroit.

En effet, si on appelle la sous-routine depuis un seul endroit, on connaît donc le point de retour de cette sous-routine, qui sera la ligne suivant son appel. Vous suivez ?

Donc, on peut remplacer le « `call` » vers cette sous-routine par un « `goto` », et le « `return` » de cette sous-routine par un autre « `goto` ». J'en vois qui sont en train de fumer des méninges, pour d'autres c'est clair. Vous allez voir qu'en pratique avec la routine « `inttmr0` », ce n'est pas très compliqué.


```

; Interruption TMR0
; -----
btfsc INTCON,T0IE      ; tester si interrupt timer autorisée
btfss INTCON,T0IF      ; oui, tester si interrupt timer en cours
goto intsw1            ; non test suivant
goto inttmr0           ; oui, traiter interrupt tmr0
retour
bcf INTCON,T0IF        ; effacer flag interrupt tmr0
goto restorerereg      ; et fin d'interruption
; suite du traitement interrupt

;*****
;                               INTERRUPTION TIMER 0                               *
;*****
inttmr0
incf  compteur,f       ; incrémenter compteur
goto  retour           ; fin d'interruption timer

```

Vous voyez donc que vous avez éliminé un « call » avec le « return » correspondant. Sachez que vous pouvez faire de même avec la sous-routine « intrb0 », vous gagnez en fait un niveau de sous-routine au total de votre programme. Mais ceci au dépend de la structuration de votre programme, je vous le rappelle.

Tant que nous y sommes, nous pouvons constater qu'il n'y a qu'une seule instruction (**bcf INTCON,T0IF**) entre le point de retour et un nouveau saut. Nous en profitons donc pour optimiser en taille et en temps la routine obtenue en déplaçant cette instruction et en supprimant un saut.

```

; Interruption TMR0
; -----
btfsc INTCON,T0IE      ; tester si interrupt timer autorisée
btfss INTCON,T0IF      ; oui, tester si interrupt timer en cours
goto intsw1            ; non test suivant
goto inttmr0           ; oui, traiter interrupt tmr0
; suite du traitement interrupt

;*****
;                               INTERRUPTION TIMER 0                               *
;*****
inttmr0
incf  compteur,f       ; incrémenter compteur
bcf INTCON,T0IF       ; effacer flag interrupt tmr0
goto  retorerereg      ; fin d'interruption timer

```

Voici donc un niveau de gagné. Voyons maintenant une autre astuce, pas très recommandable, et donc, je le rappelle, à n'utiliser qu'en cas d'absolue nécessité. Je l'explique car vous risquez de la rencontrer en analysant des programmes écrits par d'autres.

Bon, accrochez-vous et regardez les 2 lignes suivantes tirées de la routine d'interruption :

```

call divis      ; diviser par 2
return          ; fin d'interruption RB0/INT

```

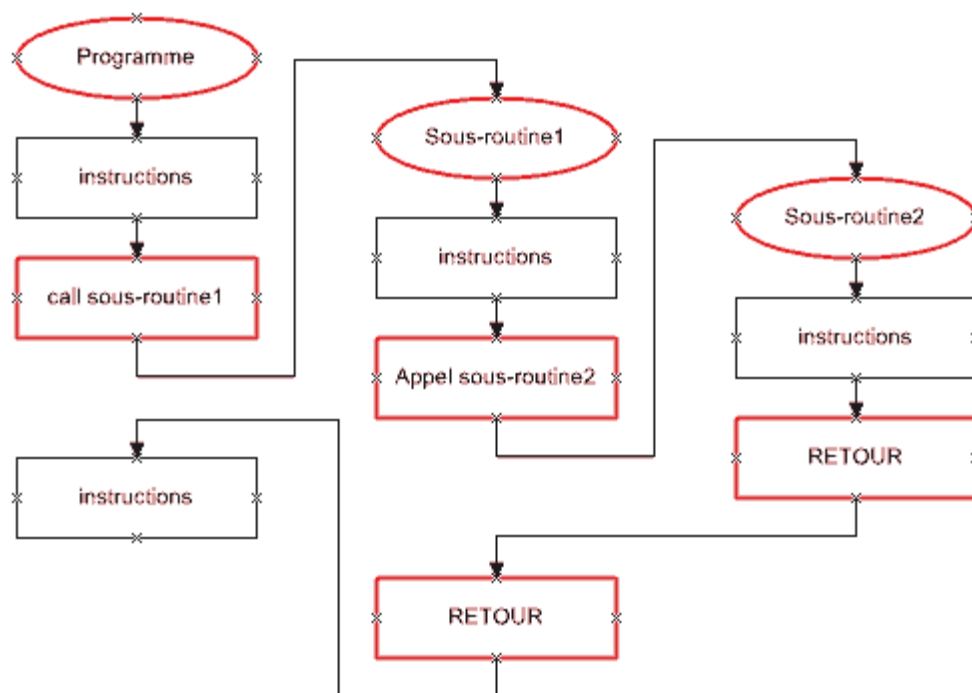
Vous ne remarquez rien ? Effectivement, c'est loin de sauter aux yeux. En fait, il s'agit d'un « **call** » suivi directement par un « **return** ».

Bon, vous allez me dire que vous n'êtes pas plus avancé. J'y arrive. Je vais raisonner en français plutôt qu'un organigramme dans un premier temps.

- Au moment de l'exécution de cette sous-routine, se trouve sur la pile l'adresse de retour du programme appelant
- A l'appel de la sous-routine « divis » j'empile l'adresse de retour sur la pile et je saute en « divis »
- Au retour de cette routine « divis », je dépile l'adresse de retour et je reviens sur la ligne « return » de ma sous-routine affichée.
- A l'exécution du « return », je dépile l'adresse du programme appelant et je retourne à celui-ci.

Je suppose que ça s'embrouille encore plus dans votre esprit, ce n'est pas le but recherché, mais, en me relisant, ça semble logique. Je continue donc mon explication.

Vous voyez donc, qu'en fait j'empile l'adresse de retour au moment de l'appel de la routine « divis ». Mais cette adresse de retour ne va me servir que pour permettre l'exécution de l'instruction « return », qui elle-même va dépiler l'adresse du programme appelant. Bon, un petit ordinogramme pour montrer tout ça ne sera pas superflu.



Que constatons-nous de spécial sur cet ordinogramme ? Et bien, si vous suivez le déroulement chronologique illustré par les flèches, vous voyez que vous rencontrez 2 « return » consécutifs.

Si vous analysez bien, vous remarquez que ceci est dû au fait que l'appel de la sous-routine2 est suivi directement par le « return » marquant la fin de la sous-routine1.

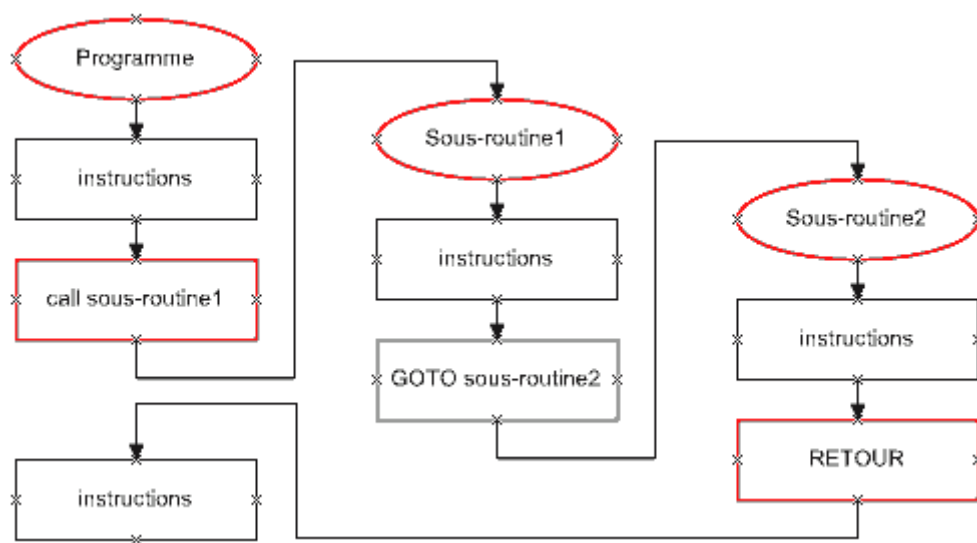
Vous ne voyez toujours pas ? En fait, si on traduit ces 2 retours consécutifs en français, on pourrait dire :

On dépile l'adresse de retour empilée afin de pointer sur l'instruction qui demande de dépiler l'adresse de retour afin d'obtenir l'adresse de retour vers le programme.

On peut donc se dire que si l'adresse de retour dépilée dans la sous-routine2 ne sert qu'à obtenir l'adresse de retour qui sera dépilée par la sous-routine1, on pourrait souhaiter retourner directement de la sous-routine2 vers le programme principal. Vous commencez à comprendre ?

En fait, comme on ne peut pas dépiler 2 adresses à la fois, on pourrait tout aussi bien se dire qu'il suffit de ne pas empiler l'adresse de retour vers la sous-routine1, ce qui fait que le retour de la sous-routine2 permettrait le retour direct vers le sous-programme. Comme on n'aurait pas empilé, on gagnerait un niveau de sous-routine.

Or, comment ne pas empiler ? Simple ! Il suffit de remplacer le « call » par un « goto ». Ceci vous donne la chronologie suivante :



Un petit mot d'explication. En fait, si vous suivez à partir du programme, vous voyez que nous rencontrons un « call » vers la sous-routine1. On empile donc l'adresse de retour vers le programme principal. Ensuite, on arrive dans la sous-routine1, dans laquelle un goto nous renvoie à la sous-routine2. Dans celle-ci on trouve un « return » qui dépile l'adresse précédemment empilée, soit celle du retour vers le programme principal, sans repasser par la sous-routine1.

De cette façon, la sous-routine2 peut être appelée de façon classique par un « call », ou appelée par un « goto », auquel cas le retour s'effectue depuis le programme qui a appelé la sous-routine qui a appelé la sous-routine2

Si vous comptez les niveaux de sous-routine, vous avez gagné un niveau. Ceci se traduit en langage d'assemblage par :

```

;*****
;                               INTERRUPTION RB0/INT                               *
;*****
intrb0
    movf    mavariabale        ; charger mavariabale
    goto    divis              ; diviser par 2

;*****
;                               Diviser par 2                               *
;*****
divis
    bcf     STATUS,C           ; effacer carry
    rrf     mavariabale,f       ; diviser mavariabale par 2
    return                      ; retour

```

ATTENTION : ne me faites pas écrire ce que je n'ai pas écrit. Je vous rappelle que je déconseille fortement ce type de pratique, à ne réserver qu'aux cas désespérés. J'explique ceci afin de vous aider à comprendre certaines astuces que vous risquez de rencontrer si vous examinez des programmes réalisés par d'autres.

Il existe encore d'autres méthodes. Vous verrez par exemple dans le chapitre consacré au debugger, comment utiliser une sous-routine sans utiliser un seul niveau sur la pile. Mais lisez dans l'ordre, quand vous en serez arrivés là, il sera temps de vous casser les méninges.

8.4.2 L'organisation des ressources

Je vais maintenant vous montrer une astuce très simple concernant des zones de données à copier, à comparer, etc.

Le problème est que vous ne disposez que d'un seul pointeur indirect, FSR. La manipulation simultanée de 2 zones implique une gymnastique permanente au niveau de ce registre.

Imaginons le petit programme suivant (pour tester au simulateur) :

```

LIST      p=16F876 ; Définition de processeur
#include <p16F876.inc>; fichier include

; Zone de 80 bytes
; -----

CBLOCK 0x20          ; Début de la zone (0x20 à 0x6F)
Zone1 : D'14'        ; table de 14 octets
Zone2 : D'14'        ; table de 14 octets
cmpt : 1             ; variable locale
sav1 : 1              ; sauvegarde pointeur1
sav2 : 1              ; sauvegarde pointeur2
inter : 1             ; zone intermédiaire

ENDC                ; Fin de la zone

```

```

ORG 0x00
start
    movlw D'14'          ; nombre d'octets à transférer
    movwf cmpt           ; dans compteur
    movlw Zone1           ; adresse zone1
    movwf sav1           ; dans pointeur1
    movlw Zone2           ; adresse zone2
    movwf sav2           ; dans pointeur2
loop
    movf sav1,w           ; charger pointeur1
    movwf FSR             ; dans pointeur indirect
    movf INDF,w           ; charger valeur pointée
    movwf inter           ; dans valeur intermédiaire

    movf sav2,w           ; charger pointeur2
    movwf FSR             ; dans pointeur indirect
    movf inter,w          ; charger valeur intermédiaire
    movwf INDF            ; transférer valeur
    incf sav1,f           ; incrémenter pointeur1
    incf sav2,f           ; incrémenter pointeur2
    decfsz cmpt,f         ; décrémenter compteur d'éléments
    goto loop             ; pas dernier, suivant
END

```

Vous constatez que ce n'est vraiment pas pratique de modifier et sauver sans arrêt FSR. Ceci impose 2 variables de sauvegarde et l'utilisation d'une variable intermédiaire pour stocker la donnée à transférer.

La première chose à laquelle vous pouvez penser, c'est d'économiser les variables de sauvegarde de FSR, puisque la distance entre Zone1 et Zone2 est fixe. Ceci nous donne :

```

start
    movlw D'14'          ; nombre d'octets à transférer
    movwf cmpt           ; dans compteur
    movlw Zone1           ; adresse zone1
    movwf FSR            ; dans pointeur indirect

loop
    movf INDF,w          ; charger valeur pointée
    movwf inter           ; dans valeur intermédiaire

    movlw Zone2-Zone1     ; écart entre les 2 zones
    addwf FSR,f           ; pointer sur zone2
    movf inter,w          ; charger valeur intermédiaire
    movwf INDF            ; transférer valeur
    movlw Zone1-Zone2+1   ; écart entre zone2 et zone1 suivant
    addwf FSR,f           ; pointer sur suivant
    decfsz cmpt,f         ; décrémenter compteur d'éléments
    goto loop             ; pas dernier, suivant
END

```

C'est déjà mieux, mais on a toujours besoin d'une sauvegarde intermédiaire. La ligne comprenant Zone1-Zone2+1 s'explique par le fait que pour pointer sur la Zone1, on doit ajouter la valeur négative Zone1-Zone2, et, qu'ensuite, on doit incrémenter cette valeur pour passer à la variable suivante, d'où le « +1 » qui permet d'éviter l'incrémentation.

On peut alors se dire qu'il serait judicieux de placer les tables pour qu'un seul bit différencie leur emplacement. Zone1 va de B'0010 0000' à B' 0010 1101'

La Zone2 commence dès lors en B'0010 1110'. On remarque que si on décale tout simplement Zone2 de 2 emplacements, elle s'étendra de B'0011 0000' à B'0011 1101'. Entre Zone1 et Zone2, seul le bit 4 de l'emplacement est modifié.

Profitons de cette astuce :

```

; Zone de 80 bytes
; -----

CBLOCK 0x20          ; Début de la zone (0x20 à 0x6F)
Zone1 : D'14'        ; table de 14 octets : NE PAS DEPLACER
cmpt : 1             ; variable locale
libre : 1            ; emplacement libre
Zone2 : D'14'        ; table de 14 octets : NE PAS DEPLACER
ENDC                ; Fin de la zone

ORG 0x00
start
    movlw D'14'        ; nombre d'octets à transférer
    movwf cmpt         ; dans compteur
    movlw Zone1         ; adresse zone1
    movwf FSR          ; dans pointeur indirect

loop
    movf INDF,w        ; charger valeur pointée
    bsf FSR,4          ; pointer zone2
    movwf INDF         ; transférer valeur
    incf FSR,f         ; pointer sur suivant
    bcf FSR,4          ; repointer Zone1
    decfsz cmpt,f      ; décrémente compteur d'éléments
    goto loop         ; pas dernier, suivant
END

```

Dans ce cas, on peut parler d'optimisation, aussi bien en place qu'en vitesse d'exécution, et même en nombre de variables RAM utilisées.

Notez que nous avons déplacé Zone2 en y insérant le cmpt, plus une variable inutilisée qui pourra servir à un autre usage. N'oubliez pas de préciser dans votre zone de variables les zones qu'il ne faudra plus déplacer.

Vous pouvez aussi utiliser les assignations (Zone1 EQU 0x20), mais alors faites très attention aux double-emplois, je vous déconseille cette méthode.

Vous pouvez également utiliser cette technique en plaçant vos tables dans des banques différentes. Si vous placez, par exemple Zone1 en 0x20 (banque0) et Zone2 en 0xA0 (banque1), le passage de la source à la destination se fera cette fois en modifiant le bit 7 de FSR.

Et si vous placez Zone1 en 0x20 (banque0) et Zone2 en 0x120 (banque2), alors le passage de l'une à l'autre ne nécessitera que la modification du bit IRP du registre STATUS.

Vous voyez donc que simplement définir un emplacement judicieux pour vos variables peut vous permettre des optimisations intéressantes.

Notes : ...

Notes : ...

9. Les différents types de reset

Nous passons maintenant à l'étude des différentes causes qui peuvent aboutir à provoquer un **reset** de notre PIC. C'est à dire un **redémarrage** depuis **l'adresse 0x00** et la réinitialisation éventuelle de certains registres.

Il peut être en effet parfois utile de pouvoir détecter, au niveau de la procédure d'initialisation, le type de reset qui vient d'être exécuté. Ceci afin de permettre de réaliser des opérations différentes.

On pourrait avoir, par exemple, un procédure qui initialiserait le système en cas de premier démarrage, et une autre qui permettrait de récupérer un processus « planté » après un débordement de watchdog suite à un violent parasite secteur.

Nous allons donc voir comment distinguer et reconnaître la cause du reset en cours.

9.1 Les 6 types de reset

Nous pouvons dire que si notre programme est contraint de redémarrer au début de sa séquence, à l'adresse 0x00, c'est qu'un des événements suivant est survenu (on excepte naturellement un saut volontaire de la part de l'utilisateur de type « goto 0x00 ») :

- Apparition de la tension d'alimentation après une coupure
- Application d'un niveau bas sur la pin MCLR durant le déroulement du programme
- Application d'un niveau bas sur la pin MCLR durant le mode « sleep »
- Débordement du watchdog durant le déroulement du programme
- **Débordement du watchdog durant le mode « sleep »**
- Remontée de la tension d'alimentation après une chute partielle

Parmi ces 6 types, **le débordement du watchdog durant le mode de sommeil n'est pas à proprement parler un reset**, puisqu'il provoque le réveil du PIC, sans provoquer un reset à l'adresse 0x00, ni même la modification des registres affectés par les resets classiques.

Cependant, j'intègre ce cas ici, étant donné qu'il met en œuvre les mêmes procédures de détection.

Il existe au moins une façon non documentée pour provoquer volontairement un reset par le programme. Comme je déconseille vivement ce genre de procédés, je n'en parlerai pas.

9.2 Le reset et les registres STATUS et PCON

Pour pouvoir déterminer quel événement a provoqué le reset, il nous faudra plusieurs bits. Etant donné que nous avons 6 causes de reset possible, nous avons besoin d'un minimum de 3 bits. En réalité, nous allons voir que 4 bits sont utilisés, 2 se trouvent dans le registre STATUS, et 2 dans un nouveau registre, le registre **PCON**.

Voici la description du registre **PCON**

b0 : **BOR** : **B**rown **O**ut **R**eset : Reset sur chute de tension
b1 : **POR** : **P**ower **O**n **R**eset : Reset par mise sous tension
b2-b7 : N.U. Non utilisés : non implémentés, lus comme « 0 »

Les bits concernés par le reset dans le registre STATUS sont les suivants :

b3 : **PD** : **P**ower **D**own bit : passage en mode sleep
b4 : **TO** : **T**ime **O**ut bit : reset par débordement du watchdog

Notez que ces bits, ainsi que les bits qui nous intéressent sont actifs à l'état bas, c'est-à-dire, pour rappel, que c'est lorsqu'ils sont à « 0 » qu'ils indiquent que l'événement a eu lieu.

Il vous faut noter que le bit BOR est dans un état indéterminé lors de la mise sous tension. Ce bit est forcé à 0 lorsqu'une chute de tension provoque le reset du PIC. Donc, si vous voulez détecter ce passage à 0, il vous faudra tout d'abord placer ce bit à 1 dans votre routine d'initialisation.

Vous pourrez par exemple utiliser les lignes suivantes :

```
init
    bsf    STATUS,RP0    ; passer en banque 1
    btfss  PCON,POR      ; tester si mise sous tension
    bsf    PCON,BOR      ; oui, alors forcer BOR à 1
    bcf    STATUS,RP0    ; repasser banque0, suite de l'initialisation
```

9.3 Détermination de l'événement

Je vous donne le tableau qui vous permettra de détecter à quel type de reset vous êtes confrontés dans votre routine d'initialisation. Vous auriez pu trouver ce tableau uniquement par déduction des explications précédentes.

POR	BOR	TO	PD	Événement qui a provoqué le reset
0	X	1	1	Mise sous tension du PIC
0	X	0	X	Impossible, puisque TO est mis à 1 à la mise sous tension
0	X	X	0	Impossible, puisque PD est mis à 1 à la mise sous tension
1	0	1	1	Retour de la tension d'alimentation après une chute de tension
1	1	0	1	Reset par débordement du watchdog
1	1	0	0	Sortie du mode sleep par débordement du watchdog
1	1	U	U	Reset provoqué par la broche MCLR
1	1	1	0	Reset par MCLR durant le mode sleep

Je vais vous commenter un peu ce tableau, afin de vous éviter de vous casser les méninges. Vous allez voir que tout est logique :

La première ligne concerne la mise sous tension du PIC, c'est à dire le passage de ce que le PIC considère comme une absence d'alimentation, vers ce qu'il considère comme une alimentation correcte. Les valeurs exactes de ces niveaux sont donnés, pour ceux que ça intéresse, dans le datasheet, au niveau des caractéristiques électriques.

Dans votre programme, vous détecterez cet événement en examinant le bit POR. S'il vaut 0, alors on est dans le cas de la mise sous tension, encore appelée Power On Reset.

Comme la mise sous tension provoque automatiquement la mise à 1 de TO et de PD, les lignes 2 et 3 du tableau sont donc impossibles. Inutile donc de tester ces configurations, elles ne se présenteront jamais.

La quatrième ligne du tableau vous montre que le simple test du bit BOR vous permet de déterminer que le reset a été provoqué par une chute de la tension d'alimentation. La façon dont le PIC détermine une chute de tension est également spécifiée dans les caractéristiques électriques.

Notez qu'une perte de tension totale, provoquera, au moment de la réapparition de la tension, un P.O.R. plutôt qu'un B.O.R. On peut donc estimer qu'un B.O.R. est consécutif à une chute de l'alimentation qui place le PIC en « risque de plantage », alors qu'un P.O.R. est la conséquence d'une perte d'alimentation qui impose l'arrêt total de l'électronique du PIC.

Une chose importante à vous rappeler, c'est que le bit BOR est dans un état indéterminé lors d'une mise sous tension. Il pourrait tout aussi bien être à 1 qu'à 0. C'est pour ça que si vous désirez utiliser ce bit, vous devrez le forcer à 1 lors du Power On Reset, afin de pouvoir tester son éventuel passage à 0.

Une seconde chose à noter est que le bit BOR, donc le reset de type « Brown Out Reset » n'est utilisé QUE si le bit BODEN a été validé au niveau de la directive `_CONFIG`. Suivant votre application, vous avez donc 2 possibilités :

- 1) Soit vous n'utilisez pas la fonction BOR, donc vous estimez qu'il est préférable que votre PIC fonctionne coûte que coûte, même si une chute d'alimentation risque de provoquer un plantage de son programme
- 2) Soit vous décidez que le PIC ne peut fonctionner que si la tension permet un fonctionnement sans risque, et donc vous activez le bit BODEN

Pour situer les esprits, vous pouvez considérer, par exemple qu'un petit gadget qui fait clignoter une LED avec une alimentation par pile doit fonctionner le plus longtemps possible, donc vous ne validez pas BODEN. Par contre, un montage qui pilote une scie circulaire automatique ne devra fonctionner que s'il n'existe aucun risque de plantage du programme, donc vous activerez BODEN.

Pour comprendre les lignes suivantes, il vous faut savoir que le bit PD peut être remis à 1 par un des 2 événements suivants :

- Mise sous tension
- Exécution de l'instruction « `clrwdt` ».

La mise à 0 est, comme je l'ai dit plus haut, provoquée par l'instruction « `sleep` »

Sachant ceci, si nous examinons la cinquième ligne du tableau, nous voyons que TO nous indique que le reset a été provoqué par le débordement du watchdog. En examinant en surplus le bit PD à 1, nous pouvons en déduire que le programme était en phase d'exécution normale

d'une instruction. Donc, dans cette circonstance, nous savons que le watchdog a provoqué un reset de notre programme en cours d'exécution.

Passons alors à la **sixième ligne**, qui ne se différencie que par le bit PD, cette fois à 0. Comme je l'ai dit, ce bit est mis à 0 par l'instruction « sleep ». Nous pouvons donc en déduire que le **watchdog** a réveillé le PIC qui était placé en mode « sleep ». Il n'y a donc pas de reset provoqué dans ce cas.

La **septième ligne** devrait vous laisser un peu perplexe. En effet, nous voyons 2 fois la mention « u » pour « unchanged » ou « inchangé ». Si les bits peuvent être à 1 ou à 0, ils peuvent être dans un des états correspondant aux deux lignes précédentes. Comment dans ce cas savoir quel est le type de reset à traiter ?

En fait, il suffit de réfléchir. Nous voyons donc d'après cette ligne qu'une action sur **MCLR** provoque un reset qui ne modifie ni TO, ni PD.

A partir de là, imaginons que $TO = PD = 1$. Dans ce cas, on peut dire qu'on a affaire à un reset qui n'est ni un P.O.R., ni un B.O.R., puisque les bits correspondants sont à « 1 ». Ce n'est pas non plus un débordement de watchdog, puisque $TO = 1$. Ne reste donc que la possibilité que le reset soit provoqué par une action sur MCLR.

Si, par contre, le bit PD était à 0, cela voudrait dire qu'on a agit sur MCLR pendant que le PIC était en mode « sleep ».

Vous allez me dire, et si TO valait 0 ? Et bien, dans ce cas, cela voudrait dire que nous venions d'avoir affaire à un reset de type débordement de watchdog, juste avant notre action sur MCLR. Donc, il nous suffisait de remettre TO à 1 une fois notre reset de type « watchdog » intervenu, pour éviter de nous trouver confronté à cette incertitude.

Si vous examinez la **huitième ligne** de notre tableau, vous voyez qu'elle correspond à un des cas que je viens d'expliquer, à savoir un reset de type **MCLR** alors que le PIC était en mode « sleep »

Au terme de l'analyse de ce tableau, vous devez être capable, le jour où vous en aurez besoin, de déterminer quel est le type de reset qui vous amène à exécuter votre routine d'initialisation.

Notez que je n'ai pas intégré les routines de test de reset dans le fichier maquette, car il est assez peu fréquent qu'on doive les utiliser. Et quand on doit les utiliser, il est rare qu'on doive différencier tous les cas. Il vous suffira donc, le cas échéant, de revenir lire ce chapitre et de traiter votre cas particulier en fonction de ce que je vous ai expliqué.

Je ne ferai pas d'exercices sur ce chapitre, car cela me semble inutile et demanderait beaucoup de manipulations pour tester tous les cas possibles. J'estime que ce serait de la perte de temps, d'autant que les programmes se résumeraient à tester les différents bits dans l'ordre donné par le précédent tableau.

Par contre, une donnée intéressante à obtenir est l'état des différents registres après chaque type de reset. Je vous conseille donc de regarder le **tableau 12-6** du datasheet qui vous donnera toutes les informations utiles à ce niveau.

Certains registres sont en effet affectés de façon différente selon le type de reset intervenu.

Notes : ...

10. Les ports entrée/sortie

Nous allons maintenant voir dans ce chapitre les différents ports du 16F87x dans leur utilisation en tant que PORT I/O, c'est-à-dire sans utiliser les fonctions spéciales des pins concernées.

Nous étudierons ces fonctions spéciales dans les chapitres correspondants au moment voulu. Les PICs 16F87x disposent en effet d'innombrables possibilités, ce qui ne rend pas simple la description exhaustive de toutes les utilisations possibles.

Commençons donc par les vieilles connaissances, et en premier lieu par :

10.1 Le PORTA

Nous en avons déjà parlé dans le chapitre sur la conversion des programmes. En fait, le PORTA, dans sa partie PORT I/O est fort semblable à celui observé sur le 16F84.

Nous trouvons donc ici 6 pins I/O numérotées de RA0 à RA5. Nous avons donc 6 bits utiles dans le registre PORTA et 6 bits dans le registre TRISA. Les bits 6 et 7 de ces registres ne sont pas implémentés. Comme toujours, ils seront lus comme des « 0 ».

La principale différence avec le 16F84 provient de ce que le PORTA, au moment du reset, est configuré comme un ensemble d'entrées analogiques. Donc, nous devons forcer une valeur dans le registre ADCON1 dans notre routine d'initialisation pour pouvoir utiliser ce port comme port d'entrée/sortie de type général.

Je verrai le registre ADCON1 dans le chapitre consacré au convertisseur A/D. L'important, ici, est de savoir que pour utiliser le PORTA de la même façon que sur le 16F84, nous devons placer la valeur B'x000011x', dans lequel x vaut 0 ou 1. Pour les plus curieux d'entre-vous, cette valeur provient du tableau 11-2, mais nous y reviendrons.

Donc, pour utiliser le PORTA en mode I/O, nous placerons par exemple la valeur B'00000110' dans ADCON1, soit 0x06. Notez que le registre ADCON1 se trouve en banque 1.

```
bsf    STATUS,RP0      ; passer en banque 1
movlw  0x06             ; valeur pour mode « numérique »
movwf  ADCON1           ; dans ADCON1
xxxx                                ; suite des initialisations.
```

Si vous examinez la routine d'initialisation de votre fichier maquette, vous verrez que l'initialisation de ADCON1 a été prévue. De sorte que ceci vous évite cet oubli très fréquent, à en croire le courrier que je reçois à ce sujet.

Attention à ce que l'utilisation de cette valeur influe également sur le fonctionnement du PORTE, comme vous le verrez plus loin.

Le reste du PORTA est strictement identique à celui du 16F84, avec la même remarque au niveau de RA4, qui est toujours en drain ouvert, c'est-à-dire qu'il ne permet pas d'imposer un

niveau « +5V » sur la pin correspondante. C'est également une erreur que je constate fréquemment, au niveau électronique, cette fois.

10.2 Le PORTB

Rien de particulier à dire sur ce PORTB, qui est **identique à celui du 16F84**, du moins pour la partie I/O classique.

Notez la pin **RB0** qui, en configuration d'entrée, est de type **« trigger de Schmitt »** **quand elle est utilisée en mode interruption** « INT ». La lecture simple de RB0 se fait, elle, de façon tout à fait classique, en entrée de type TTL.

Je vous renvoie donc à la première partie pour plus d'informations sur ce PORT.

10.3 Le PORTC

Nous voici dans les nouveautés. C'est un PORT qui n'existait pas sur le 16F84. Voyons donc, toujours au niveau de son utilisation « classique », quelles sont ses caractéristiques.

Tout d'abord au niveau programmation, c'est un **PORT tout ce qu'il y a de plus classique**. Nous trouvons donc un registre TRISC localisé dans la banque 1, qui permet de décider quelles sont les entrées et quelles sont les sorties. Le fonctionnement est identique à celui des autres TRIS, à savoir que le positionnement d'un bit à « 1 » place la pin en entrée, et que le positionnement de ce bit à « 0 » place la dite pin en sortie.

Notez, comme pour tous les autres ports, que la mise sous tension du PIC, et tout autre reset, force tous les bits utiles de TRISx à 1, ce qui place toutes les pins en entrée. Je n'y reviendrai plus.

Nous trouvons également le registre PORTC, qui, comme les autres PORTx, se trouve dans la banque 0. Son fonctionnement est, de nouveau, identique à celui des autres PORTx

Ensuite, au niveau électronique, nous noterons que toutes les pins, lorsqu'elles sont configurées en entrée, sont des **entrées de type « trigger de Schmitt »**. Je vous rappelle que ceci permet d'éviter (en simplifiant) les incertitudes de niveau sur les niveaux qui ne sont ni des 0V, ni des +5V, donc, en général, sur les signaux qui varient lentement d'un niveau à l'autre.

A la vue de ceci, l'utilisation pour vous du PORTC ne devrait poser aucun problème.

10.4 Le PORTD

Tout d'abord, il faut noter, mais vous l'aurez remarqué, que ce port **n'est présent que sur les 16F877, et non sur les 16F876**. Je vous rappelle qu'afin d'éviter de traîner partout des équivalences, quand je parle 16F877, j'inclus le 16F874 et autres PICs compatibles. Il en va de même pour le 16F876 par rapport au 16F873. Quand il y a une différence (comme la taille mémoire), je fais la distinction explicitement.

Une fois de plus, ce PORT fonctionne de façon identique aux autres, dans son mode de fonctionnement général.

Le registre TRISD comportera donc les 8 bits de direction, pendant que le registre PORTD correspond aux pins I/O concernées.

Notez qu'ici également, les 8 pins I/O, en mode entrée, sont du type « trigger de Schmitt ».

Prenez garde que le fonctionnement de ce port dépend également de la valeur que vous placerez dans TRISE, qui concerne pourtant, à première vue, le PORTE. Mais, au moment d'une mise sous tension, la valeur qui est placée automatiquement dans TRISE configure votre PORTD en port I/O de type général. Vous ne devez donc pas, à priori, vous en occuper si vous voulez rester dans ce mode.

10.5 Le PORTE

De nouveau, et fort logiquement, ce port n'est présent que sur les PICs de type 16F877.

Il ne comporte que 3 pins, RE0 à RE2, mais, contrairement aux autres ports, les bits non concernés de TRISE sont, cette fois, implémentés pour d'autres fonctions.

Vous remarquerez que les pins REx peuvent également être utilisées comme pins d'entrées analogiques. C'est de nouveau le registre ADCON1 qui détermine si ce port sera utilisé comme port I/O ou comme port analogique.

Par défaut, le port est configuré comme port analogique, et donc, comme pour le PORTA, vous devrez placer la valeur adéquate dans ADCON1 pour pouvoir l'utiliser comme port I/O numérique. Notez cependant que la valeur 0x06 utilisée dans l'explication pour le PORTA, place du même coup votre PORTE en mode numérique.

Vous n'avez donc, une fois de plus, rien à ajouter à votre fichier maquette pour fonctionner dans ce mode.

Au niveau électronique, les pins REx utilisées en entrée seront, une fois de plus, du type « trigger de Schmitt ».

10.5.1 Le registre TRISE

Le registre TRISE dispose de certains bits de configuration qui ne sont pas présents sur les autres TRISx. Microchip a voulu probablement faire des économies de registre en utilisant ces bits pour des fonctions qui ne relèvent pas de la détermination du sens des entrées/sorties.

Voyons donc le rôle de chacun des bits de TRISE :

b7 :	IBF	: Input Buffer Full status bit
b6 :	OBF	: Output Buffer Full status bit
b5 :	IBOV	: Input Buffer OVerflow detect bit

b4 : PSPMODE : Parallel Slave Port MODE select bit
b3 : 0 : Non implémenté (lu comme « 0 »)
b2 : : Direction I/O pour RE2 (1 = entrée)
b1 : : Direction I/O pour RE1
b0 : : Direction I/O pour RE0

Nous voyons que les bits 0 à 2 fonctionnent de façon identique à ceux des autres TRISx. Un bit à « 1 » positionne la pin en entrée, un bit à « 0 » la positionne en sortie.

Les bits 7 à 5 seront étudiés en temps utile, examinons plutôt le bit 4 (PSPMODE).

C'est ce bit qui détermine si le PORTD (et non le PORTE) sera utilisé en port I/O classique. Si vous placez 1 ici, le PORTD sera considéré comme un port d'interfaçage avec un microprocesseur. Nous y reviendrons dans le chapitre suivant.

Par chance, si les bits 0 à 2 sont bien forcés à « 1 » lors d'une mise sous tension, plaçant RE0 à RE2 en entrée, par contre, les autres bits sont forcés à « 0 » lors de cette mise sous tension. Donc le PORTD fonctionnera alors comme un port I/O classique, vous ne devez rien faire de spécial.

Notes : ...

Notes : ...

11. Le PORTD en mode PSP

11.1 A quoi sert le mode PSP ?

Le mode **PSP** (pour **P**arallel **S**lave **P**ort) 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 préparée 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 asynchrones avec le programme de votre PIC.

Voici un chapitre qui va intéresser les électroniciens « pointus » désireux d'interfacer leur PIC avec un autre processeur, de façon à, par exemple, obtenir un super UART.

Pour les autres, ne vous inquiétez pas si vous ne comprenez pas tout, car vous n'utiliserez probablement jamais cette possibilité, et, du reste, cela ne vous empêchera pas de profiter des autres possibilités de votre PIC.

La première chose est de bien se mettre d'accord sur les termes. Nous parlons ici du PORTD en mode parallèle. Il ne s'agit pas ici du port parallèle de votre imprimante, mais de la méthode de communication entre 2 microprocesseurs.

Ensuite, il faut comprendre que ce mode est un **mode « Slave » (esclave)**, c'est à dire sous le contrôle du microprocesseur sur lequel vous connectez votre PIC. C'est ce processeur qui va prendre le contrôle de votre PORTD.

11.1 Comment passer en MODE PSP ?

Commencez par regarder **la figure 3-9** du datasheet.

Ceux qui sont attentifs et qui ont lu le chapitre précédent ont déjà un élément de réponse. La première chose à faire est de **forcer le bit PSPMODE du registre TRISE à « 1 »**.

Ensuite, vous voyez que **les 3 pins RE0/RE2** sont utilisées dans un de leur mode spécial et portent de fait les dénominations **RD/CS/WR**. Ces pins sont actives à l'état bas. Ces pins sont sous contrôle du microprocesseur maître, ce qui fait que **vous devez les configurer en entrée via les bits b0/b2 de TRISE**.

Mais vous remarquez également, si vous êtes toujours attentif, que c'est la valeur imposée par une mise sous tension, donc inutile de vous en occuper.

Vous notez 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, vous utiliserez les interruptions pour gérez ce mode de fonctionnement, vous devez donc mettre en service l'interruption PSP comme indiqué dans le chapitre sur les interruptions.

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

11.2 Connexion des PORTD et PORTE en mode PSP

Si nous reprenons nos 3 pins du PORTE, avec les noms utilisés pour cette fonction, c'est à dire **RD**, **CS**, et **WR**, nous pourrions dire :

La pin « **CS** » est la pin « **C**hip **S**elect », que tous les électroniciens spécialisés dans les microprocesseurs connaissent.

Pour les autres, on peut dire que le « Chip Select » 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.

Je pourrais donner ici un cours complet sur les techniques d'interfaçages, mais ce n'est pas le but de cet ouvrage. Je part du principe que quelqu'un qui veut interfacer son PIC avec une carte à microprocesseur possède suffisamment de connaissances à ce niveau pour se passer de mes explications.

Si ce n'était pas le cas, et si je recevais du courrier précis concernant ce point, j'ajouterai une annexe pour en parler.

Ensuite, nous trouvons les signaux « **RD** » pour « **R**ea**D** » et « **WR** » pour « **W**Rite ». De nouveau ces signaux sont actifs à l'état bas. Notez que sur certains microprocesseurs, il n'existe qu'une seule pin RD/WR. A vous donc dans ce cas de générer électroniquement les 2 signaux, ce qui n'est pas compliqué.

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.

11.3 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, vous voyez donc qu'il existe 2 cas : soit le microprocesseur écrit dans le pic, soit il lit.

Considérons tout d'abord 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 exécute un « movf PORTD,w » 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 lorsque vous lisez le PORTD, vous obtenez 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.

Vous voyez donc que tout ceci est extrêmement simple. Ce n'est pas plus compliqué au

niveau d'une lecture de donnée depuis le microprocesseur :

- Le PIC écrit sa donnée dans le latch d'écriture, qui est également PORTD.
- Ceci provoque le positionnement du bit OBF (Output Buffer Full = buffer de sortie plein)
- Quand le microprocesseur le décide, il effectue une lecture du PORTD, et récupère alors la valeur que vous y avez 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.

11.4 Le fonctionnement matériel

Les chronogrammes des figures 3-10 et 3-11 du datasheet vous donnent les déroulements chronologiques des différentes étapes.

Un spécialiste des techniques à microprocesseurs n'aura aucun mal à les interpréter, je vais cependant vous les résumer de façon simple, et « en clair ». Voyons tout d'abord la chronologie d'une écriture.

Avant tout, souvenez-vous que dans un échange, 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

Ce chapitre était particulièrement ardu à comprendre, j'en suis conscient. Les explications risquent cependant de vous être utiles un jour, c'est pourquoi j'ai abordé le sujet de façon assez détaillée.

Vous vous rendez compte qu'il m'est particulièrement difficile de vous donner un exercice pratique mettant en œuvre ce mode particulier, car cela nécessiterait de construire une seconde carte gérée en maître par un microprocesseur.

Mais je suis convaincu qu'une fois cet ouvrage et le précédent assimilés, vous pourrez construire le cas échéant votre programme de communication inter-processeurs.

Notes :

12. Les accès à la mémoire eeprom

Les accès en mémoire eeprom ne sont pas plus compliqués que pour le 16F84. Il suffit en effet de respecter la procédure indiquée par Microchip.

Le fichier maquette que je vous fournis contient 2 macros destinées à l'écriture et à la lecture en eeprom.

Ces macros mettent en jeu 4 registres, à savoir EEADR, EEDATA, EECON1, et EECON2. Ces registres ont déjà été décrits dans la première partie consacrée au 16F84. Néanmoins, le registre EECON1 subit pour cette version quelques modifications que nous allons décrire.

Le registre EEADR est toujours utilisé pour placer l'adresse relative en EEPROM, tandis que EEDATA contiendra la donnée lue ou à écrire.

Comme pour le 16F84, le registre EECON2 n'en est pas véritablement un. Microchip l'utilise en tant que registre de commande. L'écriture de valeurs spécifiques dans EECON2 provoque l'exécution d'une commande spécifique dans l'électronique interne du PIC.

Vous disposez d'une zone de 128 octets en mémoire EEPROM pour les modèles 16F873/4, et d'une zone de 256 octets pour les modèles 16F876/7.

L'adresse relative de l'accès EEPROM est donc comprise entre 0x00 et 0xFF, ce qui permet de n'utiliser qu'un registre de 8 bits pour définir cette adresse, tout comme sur le 16F84.

12.1 Le registre EECON1

b7 :	EEPGD	:	EEProm Program/Data select bit
b6 :	N.U.	:	Non Utilisé
b5 :	N.U.	:	Non Utilisé
b4 :	N.U.	:	Non Utilisé
b3 :	WRERR	:	Write Error flag bit
b2 :	WREN	:	Write Enable
b1 :	WR	:	Write control bit
b0 :	RD	:	Read control bit

Le bit **EEPGD** permet de sélectionner si on désire écrire en mémoire EEPROM ou en mémoire FLASH. Et oui, c'est possible au niveau du 16F87x, nous verrons comment dans le chapitre suivant. Il s'agit donc d'un nouveau bit, qui n'existait pas sur le EECON1 du 16F84.

Par contre, nous constatons la disparition du bit EEIF (le flag d'interruption de fin d'écriture EEPROM). Celui-ci a tout simplement migré vers le registre PIR2, vous vous en souvenez peut-être.

Les autres bits demeurent inchangés, avec pour rappel :

- WRERR qui indique une interruption anormale du cycle d'écriture (par exemple, un reset)
- WREN qui permet d'autoriser les opérations d'écriture (sorte de sécurité pour éviter les écritures accidentelles)
- WR qui permet de lancer le cycle d'écriture
- RD qui permet de lancer le cycle de lecture

Passons maintenant aux applications concrètes :

12.2 L'accès en lecture

Commençons par examiner la procédure de lecture, via la macro concernée :

```

EEPROM macro                ; lire eeprom(adresse & résultat en w)
    clrwdt                  ; reset watchdog
    bcf STATUS,RP0          ; passer en banque2
    bsf STATUS,RP1
    movwf EEADR              ; pointer sur adresse eeprom
    bsf STATUS,RP0          ; passer en banque3
    bcf EECON1,EEPGD        ; pointer sur eeprom
    bsf EECON1,RD           ; ordre de lecture
    bcf STATUS,RP0          ; passer en banque2
    movf EEDATA,w           ; charger valeur lue
    bcf STATUS,RP1          ; passer en banque0
endm

```

La procédure est extrêmement simple. Elle consiste tout simplement à écrire l'adresse relative EEPROM concernée dans le registre EEADR, de positionner le bit RD, et de récupérer la valeur lue qui se trouve alors dans le registre EEDATA.

Pour cette macro, l'adresse de lecture est passée dans le registre « W », la valeur lue est également retournée en W.

Nous pouvons donc appeler cette macro par une portion de code comme ceci :

```

movlw    adreseeprom        ; charger l'adresse de lecture dans W
EEPROM    ; lire EEPROM
movwf    madata             ; sauver valeur lue

```

Rien donc de particulier ou de neuf dans cette procédure. La seule chose à se souvenir, c'est que l'emplacement physique réel de l'EEPROM commence à l'adresse 0x2100, alors que l'adresse relative que nous devons employer dans notre programme commence, elle, à la valeur 0x00. Autrement dit, lorsque vous accédez à l'adresse EEPROM 0x01 dans votre programme, la donnée se trouvera physiquement à l'adresse 0x2101 de votre PIC.

Notez que l'instruction « clrwdt » n'est pas vraiment nécessaire, mais est utile lors de l'appel successif de la macro.

12.3 L'accès en écriture

Procédons comme pour la lecture, en examinons la macro de notre fichier maquette.

```
WEEPROM macro addwrite ; la donnée se trouve dans W
LOCAL loop
bcf STATUS,RP0 ; passer en banque2
bsf STATUS,RP1
movwf EEDATA ; placer data dans registre
movlw addwrite ; charger adresse d'écriture
movwf EEADR ; placer dans registre
bsf STATUS,RP0 ; passer en banque3
bcf EECON1 , EEPGD ; pointer sur mémoire data
bsf EECON1 , WREN ; autoriser accès écriture
bcf INTCON , GIE ; interdire interruptions
movlw 0x55 ; charger 0x55
movwf EECON2 ; envoyer commande
movlw 0xAA ; charger 0xAA
movwf EECON2 ; envoyer commande
bsf EECON1 , WR ; lancer cycle d'écriture
bsf INTCON , GIE ; réautoriser interruptions
loop
clrwdt ; effacer watchdog
btfsc EECON1 , WR ; tester si écriture terminée
goto loop ; non, attendre
bcf EECON1 , WREN ; verrouiller prochaine écriture
bcf STATUS , RP0 ; passer en banque0
bcf STATUS , RP1
endm
```

Ici, nous aurons 2 paramètres à passer à la macro, à savoir la donnée à écrire, et l'adresse à laquelle on doit écrire cette donnée. Comme nous n'avons qu'un seul registre W, nous choisirons donc d'ajouter un argument à cette macro, argument appelé « addwrite » pour adresse d'écriture. La valeur est donc passée dans W, l'adresse est donnée en argument.

La procédure n'est guère compliquée : on commence par mettre la donnée à écrire dans EEDATA et l'adresse d'écriture dans EEADR. Ensuite, on met en service les écritures sur la mémoire eeprom.

On interdit ensuite les interruptions (recommandé par Microchip), et on exécute « bêtement » la séquence de lancement de l'écriture. On peut alors réautoriser les interruptions.

Il nous faut ensuite attendre la fin de l'écriture, après quoi on peut de nouveau « remettre le verrou » en interdisant de nouveau les écritures.

Il faut cependant être attentif et noter quelques particularités de notre macro.

En premier lieu, vous constatez que notre macro remplace le bit GIE à 1 afin de réautoriser les interruptions précédemment interdites. Il va de soi que s'il n'est pas souhaitable que ce bit soit positionné dans votre programme, il vous faudra alors supprimer les lignes concernant GIE dans la macro.

Ensuite, la boucle d'attente d'effacement de WR (qui repasse à 0 une fois l'écriture terminée) peut être remplacée par une mise en sommeil du PIC. Vous savez en effet qu'il est possible de réveiller le PIC sur le positionnement d'un flag d'interruption. Le flag concerné (EEIF) permet donc, en fin d'écriture, de réveiller le PIC.

Vous pouvez également vous dire qu'il est dommage d'attendre la fin de l'écriture avant de poursuivre votre programme. En effet, si vous n'avez qu'une seule valeur à écrire, rien ne vous interdit d'exécuter des instructions durant que le PIC effectue l'écriture EEPROM en interne. Ce que vous ne pouvez pas faire, c'est procéder à une nouvelle écriture avant que la précédente ne soit terminée.

Vous pouvez alors imaginer, à condition de laisser en permanence le bit WREN positionné, de tester WR AVANT de procéder à l'écriture. Ainsi, la première écriture n'attend pas, la seconde attendra que la première soit terminée avant d'écrire, mais n'attendra pas que la seconde soit achevée avant de sortir. Il suffit alors de déplacer la boucle avant la séquence imposée pour l'écriture.

Enfin, n'oubliez pas que vous pouvez utiliser les interruptions pour être prévenu de la fin de l'écriture, et donc de la possibilité d'écrire l'octet suivant.

Je vous rappelle que la durée de vie minimale garantie de l'eprom est de 100.000 cycles d'écriture par adresse.

A titre d'information, l'écriture d'un octet en mémoire eeprom nécessite une durée de l'ordre de 4ms. Avec un 16F876 cadencé à 20MHz, cela représente pas moins de 20.000 cycles d'instruction. Vous voyez donc que les écritures en EEPROM nécessitent énormément de temps à l'échelle du PIC. La petite boucle qui semble anodine, en place de l'utilisation des interruptions, vous fait donc perdre 20.000 cycles (et tout ça en 3 lignes). Notez que le « clrwdt » n'est en réalité pas nécessaire, puisque l'écriture ne prend qu'au maximum 4ms, mais est utile dans le cas d'exécutions successives de cette même macro. Ca ne change du reste rien au temps « perdu » dans la boucle d'attente.

12.4 Initialisation d'une zone eeprom

Nous l'avions déjà vu, mais pour compléter ce chapitre, je vais le rappeler. La directive pour déclarer des valeurs en EEPROM est « DE ». Notez que ces valeurs seront alors écrites en EEPROM directement au moment de la programmation. Ne pas oublier dans ce cas de précéder la directive « DE » de la directive ORG pointant sur un emplacement EEPROM valide (de 0x2100 à 0x21FF).

Exemple :

```
ORG 0x2100      ; zone eeprom
DE 2,4,8,2      ; mettre 2 dans 0x2100, 4 dans 0x2101 etc.
DE 'A'          ; mettre 'A' (0x41) en 0x2104
```

13. Les accès à la mémoire programme

13.1 Généralités

Voici une possibilité intéressante qui n'existe pas sur notre bon vieux 16F84 : la possibilité d'écrire des données dans la mémoire de programme, c'est-à-dire en mémoire FLASH.

Nous allons voir que les procédures sont fort semblables aux accès en mémoire eeprom. Les registres utilisés pour ces premiers le sont également ici.

Seulement, il faut que vous vous rappeliez que la mémoire de programme est constituée, non pas d'octets, mais de mots de 14 bits. Donc, l'écriture et la lecture concerne des mots de 14 bits.

Vous vous rendez bien compte qu'un mot de 14 bits ne peut pas tenir dans un registre de 8 bits. Donc les données devront être passées par le biais de 2 registres.

Si, maintenant, nous raisonnons sur l'adresse concernée par l'opération de lecture ou d'écriture, nous voyons très bien également que cette adresse ne pourra tenir dans un seul registre. En effet, la zone mémoire se caractérise par une adresse construite sur 13 bits. Ceci nous impose donc de nouveau 2 registres pour la définition de l'adresse.

Forts de ce qui précède, nous pouvons alors en conclure que nous aurons besoin de 2 registres supplémentaires, ajoutés aux 4 registres utilisés pour les accès EEPROM.

Nous aurons donc pour l'adressage, les registres EEADR et EEADRH. Vous comprenez immédiatement (je l'espère) que le registre EEADRH contient les bits de poids fort (Hight) de l'adresse d'accès.

De même, nous aurons pour les données, les registres EEDATA et EEDATH. Ce dernier, également, contient les bits de poids fort de la donnée lue ou à écrire.

Attention de bien vous souvenir que les données en mémoire programme sont des données de 14 bits (forcément la même longueur que les instructions), alors que les adresses sont codées sur 13 bits.

EEADRH contiendra donc au maximum 5 bits utiles, alors que EEDATH en contiendra 6.

J'espère que jusqu'à présent, c'est clair. Si ça ne l'était pas, relisez calmement ce qui précède.

13.2 Les accès en lecture

Commençons par le plus simple, une lecture de donnée dans la mémoire programme. Voyons donc le code suivant :

```

bcf    STATUS,RP0      ; passer en banque 2
bsf    STATUS,RP1
movlw  high  adresse   ; charger octet fort adresse
movwf  EEADRH         ; placer dans pointeur adresse haut
movlw  low   adresse   ; charger octet faible adresse
movwf  EEADR          ; placer dans pointeur adresse bas
bsf    STATUS,RP0      ; passer banque 3
bsf    EECON1,EEPGD    ; pointer sur mémoire flash
bsf    EECON1,RD       ; lancer la lecture
bcf    STATUS,RP0      ; passer banque2
nop                    ; 2 cycles nécessaires
movf   EEDATA,w        ; charger 8 lsb
xxxx                    ; sauver ou utiliser 8 lsb
movf   EEDATH,w        ; charger 6 msb
xxxx                    ; sauver ou utiliser 6 msb
bcf    STATUS,RP1      ; passer banque0

```

Si vous regardez attentivement, vous voyez, dans cette portion de code, 2 directives. Il s'agit de « high » et de « low ». Leur utilisation est très simple :

« high » signale à MPASM qu'il doit prendre l'octet de poids fort de la donnée sur 16 bits placée derrière.

« low », fort logiquement, indique qu'il faut prendre l'octet de poids faible.

Prenons un exemple concret avec la portion de code suivante :

```

ORG    0x1F05
label
movlw  low   label     ; on charge donc low 0x1F05, donc on charge 0x05
movlw  high  label     ; on charge donc high 0x1F05, donc on charge 0x1F

```

Ce n'est pas plus compliqué que ça. Examinons maintenant la structure de notre lecture en mémoire FLASH.

On commence par initialiser les 2 pointeurs d'adresse, en utilisant les 2 directives précédentes. « adresse » représente dans ce cas une étiquette placée dans notre zone de données (nous verrons un exemple concret).

Ensuite, on signale qu'on travaille en mémoire FLASH, et non en mémoire EEPROM, en positionnant le bit EEPGD, puis on lance la lecture.

Le PIC va mettre 2 cycles pour lire la donnée en FLASH. Durant ces 2 cycles, Microchip suggère d'utiliser 2 instructions « nop ». J'ai cependant profité du fait que je devais changer de banque pour utiliser utilement un de ces 2 cycles. Ne reste donc qu'un « nop ». Notez que durant ces 2 cycles, certains registres, comme EEADR sont inaccessibles. Prudence donc si vous tentez d'utiliser l'autre cycle pour réaliser une opération. A vous de tester, comme je l'ai fait.

Donc, après ces 2 cycles, nous pouvons récupérer les 14 bits de la donnée, dans les registres EEDATA et EEDATH et en faire l'usage que nous voulons. Notez que rien ne vous empêche de n'utiliser que le registre EEDATA si vous n'utilisez que 8 bits.

Une fois de plus, la routine ne présente aucune difficulté particulière. Voyons maintenant l'écriture.

13.3 Les accès en écriture

La procédure d'écriture ne présente pas non plus de difficulté particulière, il convient simplement d'être attentif à certains points précis. Voyons donc cette portion de code :

```
bcf    STATUS,RP0      ; passer en banque 2
bsf    STATUS,RP1
movlw  high  adresse    ; charger octet fort adresse
movwf  EEADRH           ; placer dans pointeur adresse haut
movlw  low   adresse    ; charger octet faible adresse
movwf  EEADR            ; placer dans pointeur adresse bas
movlw  datahaute        ; 6 bits hauts de la donnée à écrire
movwf  EEDATH           ; placer dans registre data haute
movlw  databasse        ; 8 bits bas de la donnée à écrire
movwf  EEDATA           ; dans registre data basse
bsf    STATUS,RP0      ; passer banque 3
bsf    EECON1,EEPGD     ; pointer sur mémoire programme
bsf    EECON1,WREN      ; autoriser les écritures (verrou)
bcf    INTCON,GIE       ; interdire les interruptions
movlw  0x55             ; charger 0x55
movwf  EECON2           ; envoyer commande
movlw  0xAA             ; charger 0xAA
movwf  EECON2           ; envoyer commande
bsf    EECON1 , WR      ; lancer cycle d'écriture
nop                    ; 2 cycles inutilisables
nop
bsf    INTCON,GIE       ; réautoriser les interruptions
bcf    EECON1,WREN      ; réinterdire les écritures (verrou).
```

Vous constatez que, tout comme la procédure de lecture en mémoire programme est fort semblable à celle de lecture en EEPROM, celle d'écriture en mémoire programme est fort semblable à celle d'écriture en EEPROM.

Nous commençons, en effet, par placer l'adresse dans les 2 pointeurs (souvenez-vous, 13 bits).

Ensuite, nous plaçons la donnée dans les 2 registres prévus. Notez que rien ne vous oblige d'utiliser les 14 bits, si vous souhaitez mémoriser des octets, il vous suffit de ne pas utiliser EEDATH.

Il nous faut maintenant indiquer que nous travaillons en mémoire programme, en positionnant EEPGD.

Vient, une fois de plus, la procédure standard d'écriture imposée par Microchip. Comme dans le cas précédent, cette procédure utilise des valeurs à placer dans le registre fictif EECON2.

Le Pic va alors s'arrêter, le temps d'effectuer l'écriture dans la mémoire programme. Lorsqu'il se réveillera, entre 4 et 8 ms plus tard, les 2 cycles suivants ne seront pas exécutés.

Celui-ci va continuer à avancer dans le déroulement du programme, mais n'interprétera pas les 2 commandes suivantes. Je vous conseille donc (tout comme Microchip) de mettre 2 « nop » à cet endroit. Ce temps est nécessaire au PIC pour écrire les données en mémoire programme.

Vous pouvez maintenant, si vous le voulez, réautoriser les interruptions, puis remettre le verrou de protection des écritures.

13.4 Particularités et mise en garde

Vous constatez donc que les méthodes d'accès sont fort semblables pour les accès en EEPROM et en FLASH. Il importe toutefois de bien cerner quelques différences notables.

- 1) Les accès en EEPROM sont limités à 256 emplacements, alors qu'en mémoire programme, vous avez accès à l'intégralité des 2K mots de la mémoire. Vous pouvez donc créer, par exemple, de plus gros tableaux.
- 2) Les données stockées en EEPROM ont une taille limitée à 8 bits, alors que les données stockées en mémoire programme ont une taille maximale de 14 bits.
- 3) Vous ne pouvez écrire dans la mémoire programme que si vous avez précisé **WRT_ENABLE ON** dans votre directive de configuration.

Forts de tout ceci, vous allez me dire maintenant « alors je vais utiliser toujours les écritures en mémoire programme, puisqu'il n'y a que des avantages ».

Malheureusement, il y a un inconvénient, et il est de taille :

La durée de vie garantie de la mémoire FLASH est seulement de 1000 cycles d'écriture, contre 100.000 pour la mémoire EEPROM. Donc, vous voyez qu'une utilisation trop intensive de ce type d'écriture peut mener très rapidement à la destruction du PIC.

Si vous vous trompez dans la réalisation de votre programme, par exemple, et que votre routine d'écriture en mémoire programme tourne en boucle continue, ceci provoquera la destruction théorique de votre flash au bout de 1000 boucles.

Sachant qu'une procédure complète prend de l'ordre de 25 cycles, ceci nous donne une durée de vie de l'ordre de 25000 cycles. Votre PIC serait donc détruit après 5ms (oui, j'ai bien dit 5 millièmes de seconde). Vous voyez donc qu'il faut être extrêmement prudent dans l'utilisation de cette routine. Vous voyez donc l'intérêt du bit de verrou (WREN).

Notez que les toutes dernières versions de 16F87xx ont vu la durée de vie de la mémoire flash augmentée d'un facteur 100.

Voici, pour ma part comment j'utilise ces différentes zones :

En général, j'utilise les données en mémoire programme pour les données fixes, c'est-à-dire écrites au moment de la programmation. J'évite donc d'utiliser la routine d'écriture en mémoire FLASH. J'utilise par contre régulièrement celle de lecture, qui n'est pas destructive.

Je place les paramètres qui sont susceptibles de modifications éventuelles en mémoire EEPROM, dans la mesure du possible.

Ceci explique que je vais réaliser un exercice avec vous qui n'utilise que la lecture en mémoire programme. Mais une fois la lecture comprise, l'écriture l'est aussi, il suffit de recopier la procédure standardisée.

13.5 Initialisation d'une zone en mémoire programme

Vous allez me dire qu'il suffit d'utiliser la directive « **DE** ». Malheureusement, cette directive ne permet que d'écrire des données de la taille d'un octet.

Mais, en cherchant un peu dans les directives autorisées, nous trouvons la directive « **DA** », qui est à l'origine prévue pour écrire des chaînes de caractères en mémoire programme. Elle s'accommode fort bien de données de la taille de 14 bits.

Un petit exemple vaut mieux qu'un long discours :

```
ORG    0x0200
DA     0x1FF, 0x155    ; place 1FF en mémoire programme 0x200 et 0x155 en 0x201
DA     0xFFFF         ; devrait placer 0xFFFF en mémoire 0x202, mais, la taille
                        ; d'une case de mémoire programme étant limitée à 14 bits,
                        ; la valeur réelle sera de 0x3FFF
```

13.6 La technique du « bootloader ».

Je n'entrerai pas ici dans les détails, puisque j'utiliserai cette possibilité dans la troisième partie du cours. Je vous explique simplement en quoi consiste le raisonnement.

On part du principe que le PIC peut écrire dans la mémoire programme.

On place donc une portion de code en mémoire haute, qui va communiquer avec un périphérique extérieur afin de recevoir un programme à exécuter (ce peut être un PC, une EPROM externe, etc.).

Ce programme sera chargé par la routine du PIC, et placé en zone d'exécution. Ainsi, le PIC se sera programmé tout seul, sans l'intervention d'un programmeur.

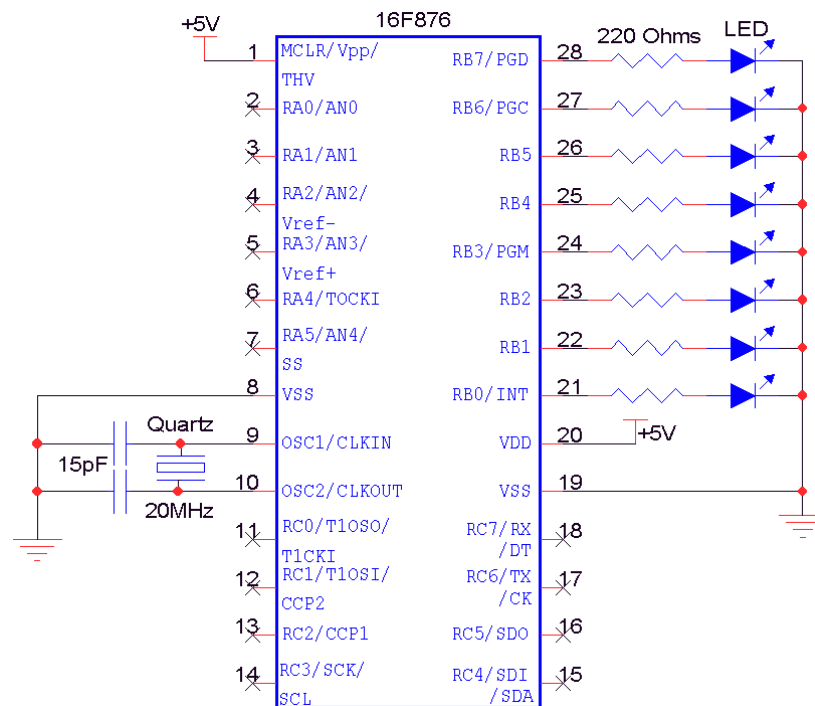
La seule limite est qu'il faut préalablement charger le programme d'écriture (Bootloader) à l'aide d'un programmeur classique, et l'activer au moment opportun. J'y reviendrai, car cela autorise de nouvelles perspectives.

13.6 Application pratique : un chenillard

Voici qui clôture la théorie sur ce sujet. Mais je ne vais pas vous laisser là. Nous allons donc, si vous voulez bien, réaliser un exemple pratique. Et, pour faire quelque chose d'original, qui diriez-vous d'un petit jeu de lumière, style chenillard ?

Je vous donne le schéma utilisé. Notez que ce chenillard ne nécessitera que 8 LEDs en plus de notre PIC. Je vous donnerai cependant un petit schéma qui vous permettra, si vous le voulez, de piloter directement des spots 220V.

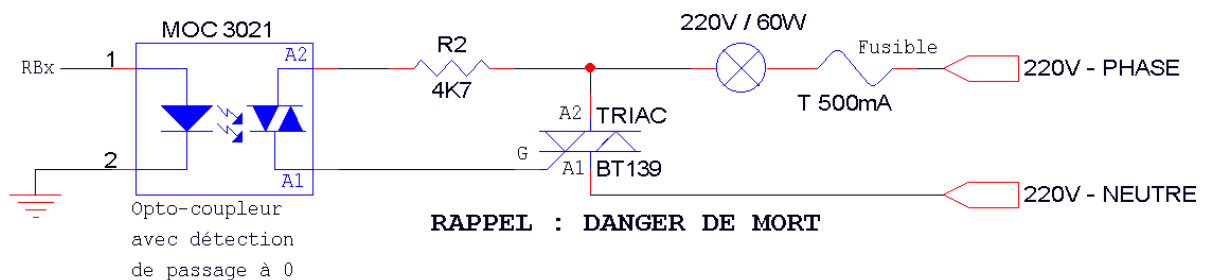
Voyons d'abord le chenillard à LEDs :



Vous voyez que c'est on ne peut plus simple, et ne vous ruinera pas.

Pour ceux qui désirent utiliser des lampes 220V, voici le schéma que je leur propose.

Remplace chaque LED dans le schéma



ATTENTION DANS CE CAS : LES TENSION MISES EN JEU PEUVENT ETRE MORTELLES. IL EST DE CE FAIT HORS DE QUESTION DE REALISER CE MONTAGE SUR UNE PLATINE D'ESSAIS. UN CIRCUIT IMPRIME DEVRA ETRE REALISE, MIS EN BOITIER, ET LES MESURES DE SECURITE CORRECTES (TERRES, FUSIBLES, CEM) DEVRONT ETRE PRISES.

Donc, si vous n'êtes pas un habitué des montages électroniques, ignorez ce schéma, il y va de votre sécurité. Réalisez la version à LEDs.

Afin d'avoir une idée de ce que nous avons à faire, nous allons créer un « pseudo-code ». Ceci présente l'avantage de pouvoir définir les différentes fonctions dont nous aurons besoin, et la façon donc elles vont s'articuler.

Un pseudo-code n'est jamais que la description en français de ce que notre programme va devoir réaliser. J'aurais pu utiliser un ordinogramme, mais il faut varier les plaisirs, et satisfaire tout le monde. J'ai souvent remarqué que les partisans du pseudo-code étaient de farouches adversaires des ordinogrammes, et inversement.

Pour ma part, je ne vois pas en quoi l'utilisation de l'un empêche l'utilisation de l'autre. Tout est histoire de « felling » et dépend également de l'application en elle-même. Certains programmes se laissant mettre plus volontiers dans une forme que dans une autre.

Nous allons devoir lire des valeurs successives dans la mémoire programme, et envoyer ces valeurs sur le PORTB, afin d'allumer les LEDS concernées.

Bien entendu, afin de rester à une échelle de temps « humaine », nous allons devoir temporiser entre 2 allumages successifs. Dans le cas contraire, nous n'aurions le temps de rien voir.

Nous allons donc imaginer que nous allumerons les LEDs à une vitesse de l'ordre de 5 allumages par seconde. Ceci nous donnera donc un temps d'attente de l'ordre de 200ms.

Voyons donc ce que nous donne notre pseudo-code :

```
Début
  Initialisation du système
Boucle
  Je lis 1 octet en FLASH
  Je l'envoie sur le PORTB
  J'attends 200ms
  Je recommence la boucle
```

Vous voyez, ça présente l'avantage d'être clair.

Nous voyons donc que nous aurons besoin d'une routine pour lire en FLASH, d'une routine de temporisation, quant à l'envoi sur le PORTB, il ne nécessite qu'une instruction.

Il nous faut cependant réfléchir à la routine de lecture d'un octet en flash. Nous allons devoir pointer sur le début de notre zone de données, et, une fois toutes les données envoyées, nous devons recommencer. Il nous faut donc déterminer quand nous arrivons au bout de la zone de données.

On pourrait indiquer la longueur totale, et compter les octets, mais ceci nécessitera d'utiliser un compteur de 16 bits, pour le cas où nous déciderions d'entrer plus de 256 données.

Il serait ingénieux de penser que nous avons 8 LEDs à piloter, et que nos données comportent 14 bits. Il nous reste donc 6 bits qui ne sont pas utilisés. Pourquoi dans ce cas ne pas décider qu'un des bits positionné à 1 indique que la dernière valeur est atteinte ? Nous choisirons par exemple le premier bit libre, à savoir le bit 8.

Notre sous-routine pourra alors être du type :

```
Lire en Flash
  Lire un mot en flash
  Incrémenter pointeur FLASH pour prochaine lecture
  Bit 8 = 1 ?
    Si oui, pointer sur première donnée en flash
Retour
```

Reste notre routine de temporisation, et, pour savoir si la temporisation est terminée, je propose de se servir de l'interruption timer0, qui positionnera un flag pour le signaler.

Sachant que nous voulons une valeur approximative de 200ms, nous utiliserons un prédiviseur de 256, ce qui nous donne un temps maxi de $(256*256)/5000000$, soit 13,1 ms. Pour obtenir nos 200ms, nous devons donc passer approximativement 15 fois dans notre routine d'interruption, avant de signaler que le temps est écoulé.

```
Interruption timer0
  Décrémenter compteur
  Compteur = 0 ?
    Oui, positionner flag tmp
    Et recharger compteur avec la valeur 15
  Retour d'interruption
```

Notre sous-routine de temporisation devient alors tout simplement :

```
Tempo
  Flag tmp positionné ?
    Non, boucler sur tempo
    Oui, effacer flag tmp
  et retour
```

Vous voyez donc que nous avons de la sorte réalisé notre programme. Il ne reste « plus qu'à » remplacer les phrases en français par des instructions.

Nous allons donc maintenant nous intéresser au programme en langage d'assemblage. Commencez, comme d'habitude, par effectuer un copier/coller de votre fichier « m16F876.asm ». Renommez la copie en « lum1.asm ». Créez un projet et chargez ce fichier.

Créons tout d'abord notre zone de commentaires, comme je vous le répète sans arrêt (oui, je sais, je suis « casse-bonbons » à ce niveau).

```
;*****
;  Réalisation d'un mini-chenillard à 8 LEDs                                     *
;                                                                                   *
;*****
;                                                                                   *
;  NOM: Lum1                                                                    *
;  Date: 23/04/2002                                                            *
;  Version: 1.0                                                                *
;  Circuit: Circuit maquette (ou C.I. si lampes 220V)                        *
;  Auteur: Bigonoff                                                            *
;                                                                                   *
;*****
;                                                                                   *
;  Fichier requis: P16F876.inc                                                 *
```

```

;                                lumdat.inc                                *
;                                                                *
;*****
;                                                                *
;    Notes: les 8 sorties sont sur le PORTB. Un niveau haut allume la LED *
;            correspondante.                                           *
;            Exercice sur les lectures en mémoire FLASH                *
;            La fréquence du quartz est de 20Mhz                       *
;                                                                *
;*****

```

Maintenant, intéressons-nous à la configuration, ainsi qu'à l'habituel « include ». J'ai décidé de mettre le watchdog en service, le reset de la configuration étant des plus classiques :

```

LIST      p=16F876                ; Définition de processeur
#include <p16F876.inc>             ; fichier include

_CONFIG   _CP_OFF & _DEBUG_OFF & _WRT_ENABLE_OFF & _CPD_OFF & _LVP_OFF &
_BODEN_OFF & _PWRTE_ON & _WDT_ON & _HS_OSC

; _CP_OFF                Pas de protection
; _DEBUG_OFF             RB6 et RB7 en utilisation normale
; _WRT_ENABLE_OFF        Le programme ne peut pas écrire dans la flash
; _CPD_OFF               Mémoire EEprom déprotégée
; _LVP_OFF               RB3 en utilisation normale
; _BODEN_OFF             Reset tension hors service
; _PWRTE_ON              Démarrage temporisé
; _WDT_ON                Watchdog en service
; _HS_OSC                Oscillateur haute vitesse (20Mhz)

```

Ensuite, nous avons les valeurs qui seront placées dans les différents registres à l'initialisation. Il suffit de se rappeler que nous allons travailler avec le PORTB en sortie (donc, inutile de mettre les résistance internes en service), et que nous avons besoin d'une interruption sur le timer0, avec un prédiviseur de 256.

Nous n'effacerons pas tous les registres inutilisés, car je compte réutiliser ce programme plus loin dans le cours, afin de l'améliorer avec de nouvelles fonctions. Effacez donc seulement les assignations pour PIE2, et celle concernant le PORTC.

```

;*****
;                                ASSIGNATIONS SYSTEME                                *
;*****

; REGISTRE OPTION_REG (configuration)
; -----
OPTIONVAL EQU    B'10000111'
; RBPUR    b7      : 1= Résistance rappel +5V hors service
; PSA      b3      : 0= Assignation prédiviseur sur Tmr0
; PS2/PS0  b2/b0   : valeur du prédiviseur
;                                111 = 1/256

; REGISTRE INTCON (contrôle interruptions standard)
; -----
INTCONVAL EQU    B'00100000'
; GIE      b7      : masque autorisation générale interrupt
; PEIE     b6      : masque autorisation générale périphériques
; TOIE     b5      : masque interruption tmr0
; INTE     b4      : masque interruption RB0/Int

```

```

; RBIE      b3 : masque interruption RB4/RB7
; T0IF      b2 : flag tmr0
; INTF      b1 : flag RB0/Int
; RBIF      b0 : flag interruption RB4/RB7

; REGISTRE PIE1 (contrôle interruptions périphériques)
; -----
PIE1VAL EQU B'00000000'
; PSPIE     b7 : Toujours 0 sur PIC 16F786
; ADIE      b6 : masque interrupt convertisseur A/D
; RCIE      b5 : masque interrupt réception USART
; TXIE      b4 : masque interrupt transmission USART
; SSPIE     b3 : masque interrupt port série synchrone
; CCP1IE    b2 : masque interrupt CCP1
; TMR2IE    b1 : masque interrupt TMR2 = PR2
; TMR1IE    b0 : masque interrupt débordement tmr1

; REGISTRE ADCON1 (ANALOGIQUE/DIGITAL)
; -----
ADCON1VAL EQU B'00000110' ; PORTA en mode digital

; DIRECTION DES PORTS I/O
; -----

DIRPORTA EQU B'00111111' ; Direction PORTA (1=entrée)
DIRPORTB EQU B'00000000' ; Direction PORTB

```

Ensuite, les assignations du programme. Pour l'instant, nous savons déjà que nous avons une constante, à savoir les 15 passages dans l'interruption du timer0 pour atteindre nos 200ms. Définissons donc cette constante qui va servir à recharger le compteur (recharge en anglais se dit « reload ») . J'utilise souvent des noms de variables ou de fonctions en anglais, car l'anglais est plus court et ne comporte pas d'accents.

```

; *****
;                               ASSIGNATIONS PROGRAMME                               *
; *****

RELOADEQU D'15' ; nombre de passages dans tmr0

```

Prenez garde, une fois de plus, de ne pas tomber dans le piège grossier qui consisterait à oublier le « D ». Dans ce cas, vous auriez 0x15 passages, soit D'21' passages. Je suis certain que certains étaient prêts à tomber dans le panneau.

Je vous rappelle que la meilleure méthode est de toujours indiquer le système avant la valeur. Ainsi, on ne se trompe jamais (enfin, quand je dis « jamais », j'oublie peut-être de dire que ça m'est déjà arrivé à moi aussi...).

Nous n'avons pas encore de « define » pour l'instant, laissons la zone vide, quant aux macros, nous ne conserverons que celles concernant les banques.

```

; *****
;                               DEFINE                               *
; *****

; *****
;                               MACRO                               *
; *****

```



```

; Changement de banques
; -----

BANK0 macro                ; passer en banque0;
    bcf    STATUS,RP0
    bcf    STATUS,RP1
endm

BANK1 macro                ; passer en banque1
    bsf    STATUS,RP0
    bcf    STATUS,RP1
endm

BANK2 macro                ; passer en banque2
    bcf    STATUS,RP0
    bsf    STATUS,RP1
endm

BANK3 macro                ; passer en banque3
    bsf    STATUS,RP0
    bsf    STATUS,RP1
endm

```

Ce que nous pouvons dire sur nos zones de variables, après avoir effacé les exemples, c'est que nous avons besoin, à priori, d'une variable pour compter les passages dans le timer0.

Par contre, notre programme aura une taille de moins de 2K, inutile donc de sauvegarder PCLATH. Inutile également de sauvegarder FSR, nous n'utiliserons pas l'adressage indirect dans notre routine d'interruption toute simple. Ceci nous donne :

```

; *****
;                               VARIABLES BANQUE 0                               *
; *****
; Zone de 80 bytes
; -----

    CBLOCK    0x20        ; Début de la zone (0x20 à 0x6F)
    cmpt : 1              ; compteur de passages dans tmr0
    ENDC          ; Fin de la zone

; *****
;                               VARIABLES ZONE COMMUNE                               *
; *****
; Zone de 16 bytes
; -----

    CBLOCK    0x70        ; Début de la zone (0x70 à 0x7F)
    w_temp : 1            ; Sauvegarde registre W
    status_temp : 1       ; sauvegarde registre STATUS
    ENDC

```

Nous n'aurons nul besoin de la zone RAM en banques 1,2, et 3. Donc, nous effaçons ce qui les concerne.

Si, à se stade, nous lançons l'assemblage, nous aurons évidemment des erreurs relatives aux assignations que nous avons supprimées. Il ne suffit pas d'effacer les assignations (ce qui ne nous fait rien gagner en taille du programme), il faut surtout effacer les instructions qui y font référence.

Donc, tout d'abord, dans la routine d'interruption principale, nous allons effacer la sauvegarde et la restauration de PCLATH et de FSR.

Profitions-en pour supprimer les tests d'interruptions qui ne nous seront pas utiles. Nous n'avons besoin ici que de l'interruption tmr0. Cependant, je compte dans un futur exercice, réutiliser ce fichier. Je vous demande donc de conserver également l'interruption concernant le convertisseur A/D.

Après un minimum de réflexion, et en vous rappelant qu'il est inutile, comme notre routine est conçue, de tester la dernière interruption (si ce n'est aucune autre, c'est donc celle-ci), vous devriez arriver au résultat suivant :

```
;*****
;                               ROUTINE INTERRUPTION                               *
;*****
;sauvegarder registres
;-----
org    0x004                ; adresse d'interruption
movwf  w_temp              ; sauver registre W
swapf  STATUS,w           ; swap status avec résultat dans w
movwf  status_temp        ; sauver status swappé
BANK0                                ; passer en banque0

; Interruption TMR0
; -----
btfsc  INTCON,T0IE        ; tester si interrupt timer autorisée
btfss  INTCON,T0IF        ; oui, tester si interrupt timer en cours
goto   intswl             ; non test suivant
call   inttmr0            ; oui, traiter interrupt tmr0
bcf    INTCON,T0IF        ; effacer flag interrupt tmr0
goto   restorerereg       ; et fin d'interruption

; Interruption convertisseur A/D
; -----
intswl
call   intad              ; traiter interrupt
bcf    PIR1,ADIF          ; effacer flag interrupt

;restaurer registres
;-----
restorerereg
swapf  status_temp,w     ; swap ancien status, résultat dans w
movwf  STATUS            ; restaurer status
swapf  w_temp,f          ; Inversion L et H de l'ancien W
                                ; sans modifier Z
swapf  w_temp,w          ; Réinversion de L et H dans W
                                ; W restauré sans modifier status
retfie                    ; return from interrupt

;*****
;                               INTERRUPTION TIMER 0                               *
;*****
inttmr0
return                ; fin d'interruption timer

;*****
;                               INTERRUPTION CONVERTISSEUR A/D                               *
;*****
intad
```

```
return      ; fin d'interruption
```

Passons à l'initialisation, et supprimons ce dont nous n'aurons nul besoin, comme l'initialisation du PORTC (conservez le PORTA, nous en aurons besoin ultérieurement) ou encore l'initialisation de PIE2.

Nous pouvons également supprimer l'effacement de la RAM en banque0, nous n'utilisons d'ailleurs que peu de variables. **Profitions-en pour initialiser notre unique variable actuelle** avec la valeur de rechargement. Voici ce que vous devriez obtenir :

```
;*****
;                               INITIALISATIONS                               *
;*****
init
    ; initialisation PORTS (banque 0 et 1)
    ; -----
    BANK0          ; sélectionner banque0
    clrf PORTA      ; Sorties PORTA à 0
    clrf PORTB      ; sorties PORTB à 0
    bsf STATUS,RP0  ; passer en banque1
    movlw ADCON1VAL  ; PORTA en mode digital/analogique
    movwf ADCON1     ; écriture dans contrôle A/D
    movlw DIRPORTA   ; Direction PORTA
    movwf TRISA      ; écriture dans registre direction
    movlw DIRPORTB   ; Direction PORTB
    movwf TRISB      ; écriture dans registre direction

    ; Registre d'options (banque 1)
    ; -----
    movlw OPTIONVAL  ; charger masque
    movwf OPTION_REG ; initialiser registre option

    ; registres interruptions (banque 1)
    ; -----
    movlw INTCONVAL  ; charger valeur registre interruption
    movwf INTCON     ; initialiser interruptions
    movlw PIE1VAL    ; Initialiser registre
    movwf PIE1       ; interruptions périphériques 1

    ; initialiser variables
    ; -----
    movlw RELOAD      ; valeur de rechargement
    movwf cmpt        ; dans compteur de passage tmr0

    ; autoriser interruptions (banque 0)
    ; -----
    clrf PIR1        ; effacer flags 1
    bsf INTCON,GIE    ; valider interruptions
    goto start       ; programme principal
```

Passons maintenant aux choses sérieuses, en commençant par élaborer notre programme. Reprenons donc notre pseudo-code, et tout d'abord la routine principale. Tout vous semble correct ? Nous en reparlerons.

```
Boucle
    Je lis 1 octet en FLASH
    Je l'envoie sur le PORTB
    J'attends 200ms
```

Je retourne en Boucle

Il est très simple de convertir ceci en langage d'assemblage.

```
;*****
;                               PROGRAMME PRINCIPAL                               *
;*****

start
    clrwdt          ; effacer watch dog
    call readflash  ; lire un octet en flash
    movwf PORTB     ; le mettre sur le PORTB (allumage LEDs)
    call tempo      ; attendre 200ms
    goto start      ; boucler
END                ; directive fin de programme
```

Voyons maintenant notre sous-routine de temporisation, pour cela, reprenons notre pseudo-code :

```
Tempo
    Flag tmp positionné ?
        Non, boucler sur tempo
        Oui, effacer flag
        et retour
```

Nous avons besoin d'un flag, c'est-à-dire d'une variable booléenne, ou encore, d'un bit dans un octet. Il nous faut donc définir cette variable. Nous choisirons donc un octet en tant que « flags ». Cet octet contient 8 bits, donc 8 variables booléennes potentielles. Nous définirons le bit 0 comme notre flag booléen, que nous appellerons « flagtempo ». Notre zone de variables en banque 0 devient donc :

```
CBLOCK 0x20          ; Début de la zone (0x20 à 0x6F)
    cmpt : 1          ; compteur de passages dans tmr0
    flags : 1         ; flags divers
                ; b0 : flag de temporisation
ENDC                ; Fin de la zone
```

```
#DEFINE flagtempo flags,0 ; flag de temporisation
```

Le « define », est, comme d'habitude, destiné à nous faciliter les écritures.

Quand nous ajoutons une variable, le premier réflexe doit consister à s'occuper de son éventuelle initialisation. Au départ, nous choisissons donc d'effacer l'ensemble des flags.

```
        ; initialiser variables
        ; -----
    movlw RELOAD      ; valeur de rechargement
    movwf cmpt        ; dans compteur de passage tmr0
    clrf flags        ; effacer les flags
```

Nous pouvons donc maintenant écrire notre routine de temporisation, que nous plaçons, suivant notre habitude, entre la routine d'initialisation et le programme principal :

```
;*****
;                               TEMPORISATION                               *
;*****
;-----
```

```
; Attend que le flag soit positionné par la routine d'interruption tmr0
;-----
tempo
    btfss flagtempo    ; tester si flag tempo positionné
    goto  tempo        ; non, attendre
    bcf  flagtempo     ; oui, effacer le flag
    return             ; et retour
```

Bien entendu, pour que le flag puisse être positionné, il nous faut écrire notre routine d'interruption tmr0 à partir de notre pseudo-code :

```
Interruption timer0
    Décrémenter compteur
    Compteur = 0 ?
        Oui, positionner flag tmp
        Et recharger compteur avec la valeur 15
    Retour d'interruption
```

De nouveau, rien de plus simple, toutes les variables utiles ont déjà été définies.

```
;*****
;                               INTERRUPTION TIMER 0                               *
;*****
inttmr0
    decfsz    cmpt,f    ; décrémenter compteur de passage
    return    ; pas 0, rien d'autre à faire
    bsf       flagtempo ; si 0, positionner flag de temporisation
    movlw     RELOAD    ; valeur de rechargement
    movwf     cmpt      ; recharger compteur
    return    ; fin d'interruption timer
```

Voici donc notre temporisation opérationnelle. Ne reste plus que la lecture des données en mémoire programme.

Données, vous avez dit ? Mais oui, bien sûr, il faut d'abord les y mettre, nos données. Souvenez-vous que l'initialisation des données en mémoire programme peut se faire avec la directive « DA ».

Mais je vais en profiter pour utiliser la technique des « include », ainsi, nous ferons « d'une pierre deux coups ».

Afin d'éviter de traîner toutes les valeurs d'allumage de nos LEDs dans le programme principal, et également afin de vous éviter de devoir tout recopier, nous allons créer un fichier séparé contenant nos données.

Commencez donc par aller dans le menu « file -> new ». une fenêtre intitulée... « untitled » (ça ne s'invente pas) s'ouvre à l'écran. Allez dans le menu « files -> save as... » et sauvez-la sous le nom « lumdat.inc » dans le même répertoire que votre projet.

Vous auriez pu également choisir l'extension « .asm », ou encore « .txt », mais prenez l'habitude d'utiliser l'extension « .inc » pour vos fichiers « INClude », ceci vous permettra de mieux vous y retrouver.

Dès que c'est fait, le nouveau nom et le chemin apparaissent dans l'intitulé de la fenêtre.

Vous avez maintenant 2 fenêtres ouvertes dans l'écran de MPLAB. Mais, souvenez-vous, ce n'est pas parce que le fichier est ouvert à l'écran qu'il fait partie de notre projet.

Nous allons donc l'inclure dans celui-ci en ajoutant tout simplement une directive « include ». dans notre fichier « .asm ».

```
LIST      p=16F876      ; Définition de processeur
#include <p16F876.inc>    ; fichier include
#include <lumdat.inc>     ; données d'allumage
```

Mais bon, ce n'est pas tout d'inclure le fichier, encore faut-il savoir à quelle adresse les données vont aller se loger. En tout état de cause, il faut que ce soit après notre programme, et non en plein à l'adresse de démarrage 0x00. Le plus simple est donc d'imposer l'adresse de chargement. Vous vous souvenez sans doute que cela s'effectue à l'aide de la directive « ORG ».

Reste à choisir l'adresse de départ de nos données, donc la valeur à placer suite à notre directive « ORG ». Evitons de perdre de la place, écrivons-les directement à la fin de notre programme.

Et comment connaître l'adresse de fin de notre programme ? C'est simple, il suffit d'y placer une étiquette (attention, AVANT la directive END, sinon cette étiquette ne serait pas prise en compte).

```
start
    clrwdt      ; effacer watch dog
    call readflash ; lire un octet en flash
    movwf PORTB ; le mettre sur le PORTB (allumage LEDs)
    call tempo   ; attendre 200ms
    goto start   ; boucler
mesdata      ; emplacement de mes données (lumdat.inc)
END           ; directive fin de programme
```

Maintenant, nous allons dans notre fenêtre « lumdat.inc », et nous créons un en-tête, suivi de la directive « ORG » complétée de l'adresse de position de nos données, c'est-à-dire « mesdata ».

```
; *****
;                                     DONNEES D'ALLUMAGE                                     *
; *****
;-----
; un niveau "1" provoque l'allumage de la sortie correspondante
;-----

ORG mesdata      ; début des data en fin de programme
```

A ce stade, nous allons inscrire des données dans notre fichier « .inc », en nous souvenant que la dernière donnée aura son bit numéro 8 positionné à 1, ce qui indiquera à notre programme la fin de la zone. Ecrivons donc en nous souvenant qu'un « 1 » allume la LED correspondante, et qu'un « 0 » l'éteint.

Ajoutons donc quelque données, et pourquoi pas :

```
DA B'00000001' ; valeurs d'allumage
DA B'00000010'
```

```

DA B'00000100'
DA B'00001000'
DA B'00010000'
DA B'00100000'
DA B'01000000'
DA B'10000000'
DA B'01000000'
DA B'00100000'
DA B'00100000'
DA B'00010000'
DA B'00001000'
DA B'00000100'
DA B'00000010' |B'100000000' ; dernière valeur, on force b8

```

Notez qu'on aurait pu simplement écrire la dernière ligne de la façon suivante :

```
DA B'1000000010' ; dernière valeur avec b8 à 1
```

Ce qui était strictement équivalent. Nous pouvons fermer notre fenêtre de données (ce n'est pas parce qu'elle est fermée qu'elle n'est plus utilisée), ou, mieux, la minimiser, ce qui nous permettra de la rouvrir si nous voulons modifier les valeurs.

N'oubliez cependant pas auparavant de sauvegarder en pressant **<ctrl><s>** ou en utilisant « **save all** ».

Revenons-en à nos moutons... je voulais dire à notre lecture des données :

```

Lire en Flash
  Lire un mot en flash
  Incrémenter pointeur FLASH pour prochaine lecture
  Bit 8 = 1 ?
    Si oui, pointer sur première donnée en flash
Retour

```

En regardant cette sous-routine, nous voyons qu'on utilise un pointeur. Donc, il devra également être initialisé.

Ce pointeur, si vous vous rappelez, est un pointeur codé sur 2 octets, car l'accès à la totalité de la zone mémoire programme nécessite 13 bits. Ce pointeur est constitué, pour les accès en mémoire programme, des registres **EEADR** et **EEADRH**. Nous allons donc les initialiser de façon à ce que la première lecture corresponde à la première donnée écrite dans notre fichier « lumdat.inc ».

Ajoutons donc à nos lignes sur les initialisations des variables. Vous pensez à ça ?

```

; initialiser variables
; -----
movlw RELOAD          ; valeur de rechargement
movwf cmpt            ; dans compteur de passage tmr0
clrf flags            ; effacer les flags
movlw low mesdata      ; adresse basse des data
movwf EEADR            ; dans pointeur bas FLASH
movlw high mesdata     ; adresse haute des data
movwf EEADRH           ; dans pointeur haut FLASH

```

C'est bien d'avoir pensé aux directives « **low** » et « **high** » que nous avons vues précédemment. Rien à redire ?

Malheureusement, dans ce cas, vous avez tout faux au niveau du résultat final. Vous êtes tombé dans le piège classique. Les banques, vous les avez oubliées ? Et oui, **EEADR** et **EEADRH** sont en banque 2. Si donc, vous avez pensé à :

```

; initialiser variables
; -----
movlw RELOAD      ; valeur de rechargement
movwf cmpt        ; dans compteur de passage tmr0
clrf flags        ; effacer les flags
bsf STATUS,RP1    ; passer en banque2
movlw low mesdata ; adresse basse des data
movwf EEADR       ; dans pointeur bas FLASH
movlw high mesdata ; adresse haute des data
movwf EEADRH      ; dans pointeur haut FLASH
bcf STATUS,RP1    ; repasser en banque 0

```

alors je vous donne un « bon point » (passez me voir à la « récré »).

Mais alors, si réellement vous avez pensé à :

```

; initialiser variables
; -----
bcf STATUS,RP0    ; repasser en banque0
movlw RELOAD      ; valeur de rechargement
movwf cmpt        ; dans compteur de passage tmr0
clrf flags        ; effacer les flags
bsf STATUS,RP1    ; passer en banque2
movlw low mesdata ; adresse basse des data
movwf EEADR       ; dans pointeur bas FLASH
movlw high mesdata ; adresse haute des data
movwf EEADRH      ; dans pointeur haut FLASH
bcf STATUS,RP1    ; repasser en banque 0

```

Alors je vous dit « BRAVO ». En effet, il fallait penser que notre routine d'initialisation des variables était fausse, puisque nous avons oublié de repasser en banque 0, les initialisations précédentes concernant la banque 1 (et en plus, c'était noté).

Donc, rappelez-vous de toujours bien réfléchir et de garder l'esprit ouvert. Il ne suffit pas de réaliser « bêtement » tout ce que je vous dis sans réfléchir plus loin. D'autant que j'estime que les erreurs sont profitables, je vous en fais donc faire quelques-unes « classiques » volontairement. Il y en aura d'autres, qu'on se le dise (et dans votre intérêt, essayez de les trouver sans « tricher »).

Tout est maintenant prêt pour l'écriture de notre routine de lecture d'un octet en mémoire programme. Essayez d'écrire vous-même cette routine.

```

;*****
;                               LIRE UN OCTET EN FLASH                               *
;*****
;-----
; Lit un octet en mémoire programme, puis pointe sur le suivant
; retourne l'octet lu dans w
; si le bit 8 du mot lu vaut 1, on pointe sur la première donnée
;-----
readflash
    BANK3                ; pointer sur banque3

```



```

bsf    EECON1,EEPGD      ; accès à la mémoire programme
bsf    EECON1,RD         ; lecture
nop                                         ; 2 cycles d'attente
nop
bcf    STATUS,RP0        ; pointer sur banque2
incf   EEADR,f           ; incrémenter pointeur bas sur données
btfsc  STATUS,Z          ; tester si débordé
incf   EEADRH,f          ; oui, incrémenter pointeur haut
btfss  EEDATH,0          ; tester bit 0 data haute = bit 8 donnée
goto   readfsuite        ; si 0, sauter instructions suivantes
movlw  low mesdata       ; si 1, adresse basse des data
movwf  EEADR             ; dans pointeur bas FLASH
movlw  high mesdata      ; adresse haute des data
movwf  EEADRH            ; dans pointeur haut FLASH
readfsuite
movf   EEDATA,w          ; charger 8 bits donnée
bcf    STATUS,RP1        ; repointer sur banque0
return ; et retour

```

Quelques points sont à prendre en considération. En premier lieu, **l'incréméntation du pointeur s'effectue bien sur 2 registres**, EEADR et EEADRH. N'oubliez pas que l'accès à la mémoire programme complète nécessite 13 bits. On incrémentera donc le registre de poids fort si le registre de poids faible a « débordé », c'est à dire qu'il est passé de 0xFF à 0x00.

Si vous aviez pensé à utiliser pour ce faire l'instruction :

```

btfsc  STATUS,C          ; tester si débordé

```

L'idée n'était pas mauvaise en soi, malheureusement l'instruction « incf » précédente n'agit pas sur le bit « C ». Cela n'aurait donc pas fonctionné.

Ensuite, vous voyez que **le bit 8 de notre donnée correspond fort logiquement au bit 0 de EEDATH**. C'est donc ce dernier que nous testons pour constater que nous sommes arrivés en fin de notre zone de données.

Lancez l'assemblage et placez le fichier obtenu dans votre PIC. Aliménez, et regardez : la LED connectée sur RB0 s'allume et est la seule à s'allumer. quelque chose ne n'est donc pas passé comme prévu. A votre avis, à quel niveau ?

Il suffit de raisonner. Il y a eu allumage de la première LED, donc lecture de la première valeur en mémoire programme.

Si nous vérifions la fin de cette routine (puisque le début fonctionne), nous voyons que tout à l'air correct. On repasse bien en banque 0 avant de sortir, et l'instruction « return » est bien présente. Le problème provient sans doute d'ailleurs.

En regardant le programme principal, nous voyons que celui-ci appelle la routine de temporisation. C'est probablement dans cette routine que quelque chose « cloche ».

Regardez-la attentivement. Vous ne remarquez rien ? Non ? Et si je vous rappelle que cette routine attend durant 200ms, vous ne pensez toujours à rien ? Alors je vous rappelle que nous avons mis le watchdog en service dans notre configuration.

Alors la, ça doit faire « tilt ». En effet, le watchdog va vous provoquer un reset après une durée approximative de 18ms. Donc, bien avant que notre routine de temporisation soit terminée.

Il n'y a donc aucune chance d'arriver jusqu'à l'allumage suivant. Ajoutez donc une instruction de reset du watchdog :

```
tempo
    clrwdt          ; effacer watchdog
    btfss flagtempo ; tester si flag tempo positionné
    goto tempo      ; non, attendre
    bcf flagtempo    ; oui, effacer le flag
    return           ; et retour
```

Reprogrammez votre PIC, et contemplez l'effet obtenu. Vous pouvez alors à votre guise compléter les séquences d'allumage en ajoutant des données dans votre fichier « lumdat.inc ». N'oubliez pas que le seul bit 8 à « 1 » doit être celui de votre dernière donnée.

Vous pouvez également diminuer la valeur de RELOAD dans les assignations, de façon à accélérer le défilement. Une valeur de 4 devrait vous donner le « look K2000 », pour les nostalgiques de la série.

A ce stade, vous pouvez tenter d'interpeller votre épouse. Devant votre air admiratif pour ce jeu de lumière « intelligent », il est possible qu'elle vous gratifie d'un regard compatissant. N'en espérez guère plus, cependant, la bonté féminine a ses limites.

Par contre, c'est le moment de montrer le montage à votre jeune fils ou à votre petit frère. Pour sûr, il va vouloir installer le montage dans votre voiture.

Notes : ...

Notes : ...

14. Le timer 0

14.1 Généralités

Nous avons déjà parlé du timer 0 dans le premier livre. Ce timer fonctionne de façon identique à celui du PIC16F84. Inutile donc de revenir sur tout ce que nous avons déjà vu, toutes les explications restent d'application.

Je vais seulement vous donner quelques compléments d'information, compléments qui s'appliquent à tous les PICs de la gamme MID-RANGE.

Remarquez pour commencer que le timer 0 utilise, pour ses incréments le registre de 8 bits TMR0. Ce registre contient une valeur indéterminée à la mise sous tension, si vous désirez compter à partir d'une valeur précise (0 ou autre), placer explicitement cette valeur dans votre programme :

```
clrf    TMR0    ; commencer le comptage à partir de 0
```

J'ai testé sur quelques 16F876 en ma possession, et j'ai obtenu une valeur de démarrage constante et égale à 0xFF. Ceci ne peut cependant pas être généralisé, il s'agit au plus d'un peu de curiosité de ma part.

Ceci est valable pour la mise sous tension (reset de type P.O.R) ou pour la récupération après une chute de tension (reset de type B.O.R). Par contre, pour les autres types de reset, le contenu de TMR0 restera inchangé par rapport à sa dernière valeur.

Notez également qu'il est impossible d'arrêter le fonctionnement du timer. Une astuce que vous pouvez utiliser, si vous désirez suspendre le comptage du temps (en mode timer), est de faire passer le timer0 en mode compteur. Ceci, bien entendu, à condition que la pin RA4 ne soit pas affectée à une autre utilisation.

14.2 L'écriture dans TMR0

Tout d'abord, si nous nous trouvons en mode de fonctionnement « timer » du timer 0, nous pouvons être amenés à écrire une valeur (ou à ajouter une valeur) dans le registre TMR0. Ceci, par exemple, pour raccourcir le temps de débordement du timer avant ses 256 cycles classiques.

Voyons donc ce qui se passe au moment de **l'écriture d'une valeur dans TMR0.**

La première chose qu'il va se passer, est que **le contenu du prédiviseur**, s'il était affecté au timer 0 (souvenez-vous qu'il sert également pour le watchdog) **va être remis à 0.** Attention, je parle du contenu du prédiviseur, pas de la valeur du prédiviseur (je vais expliquer en détail).

Ensuite, **les 2 cycles d'instruction suivants seront perdus au niveau du comptage du timer.** La mesure de temps reprendra donc à partir du 3^{ème} cycle suivant.

Il importe de comprendre que le prédiviseur fonctionne de la façon suivante :

- Le prédiviseur compte jusque sa valeur programmée par PS0/PS2.
- Une fois cette valeur atteinte, TMR0 est incrémenté
- Le prédiviseur recommence son comptage à l'impulsion suivante

Supposons par exemple que TMR0 contienne la valeur 0x15, que le prédiviseur soit sélectionné sur une valeur de 8 affectée au timer 0. Supposons également que nous en sommes à 2 cycles d'instruction depuis la dernière incrémentation de TMR0. Il reste donc théoriquement 6 cycles d'instruction avant que TMR0 ne passe à 0x16. Nous avons donc la situation suivante :

Prédiviseur configuré à : 8
 Contenu du prédiviseur : 2
 Valeur de TMR0 : 0x15

Imaginons qu'à cet instant nous rencontrions la séquence suivante :

```
movlw 0x25 ; prendre la valeur 0x25
movwf TMR0 ; dans TMR0
```

Nous obtenons pour la suite de notre programme :

```
Instruction suivante : TMR = 0x25, contenu prédiviseur = 0, prédiviseur à 8.
Instruction suivante : Pas de comptage, contenu prédiviseur = 0
Instruction suivante : redémarrage du comptage : contenu prédiviseur = 1
```

Vous voyez donc que non seulement nous avons perdu 2 cycles, mais qu'en plus nous avons perdus les instructions qui avaient été comptabilisées dans notre prédiviseur.

Lorsque vous écrivez dans TMR0, il vous appartient, si nécessaire, de prendre en compte cette perte d'informations.

14.3 le timing en mode compteur

Il y a quelques précautions à prendre lorsqu'on travaille avec le timer 0 en mode compteur. En effet, les éléments à compter sont, par définition, asynchrones avec l'horloge du PIC.

Or la sortie du prédiviseur, et donc l'incrémentation du registre TMR0 est, elle, synchronisée sur le flanc montant de Q4 (Q4 étant le 4^{ème} clock de l'horloge du PIC).

Dit plus simplement, on peut dire que la sortie du prédiviseur passe à 1 au moment où la valeur de comptage est égale à la valeur du prédiviseur choisie (8 dans l'exemple précédent). Une fois cette valeur dépassée, la sortie du prédiviseur repasse à 0, jusqu'au multiple de 8 suivant (dans notre exemple).

Mais l'incrémentation de TMR0 se fait sur le test suivant : On a incrémenté si au moment du flanc montant de Q4, la sortie du prédiviseur est à 1.

On voit alors tout de suite que si le signal est trop rapide, nous risquons de rater le débordement du prédiviseur, celui-ci étant repassé à 0 avant d'avoir eu apparition du flanc montant de Q4.

On peut donc en déduire que pour être certain de ne rien rater, la sortie de notre prédiviseur devra être maintenue au minimum le temps séparant 2 « Q4 » consécutifs. Donc, l'équivalent de 4 T_{osc} (temps d'oscillateur), soit la durée d'un cycle d'instruction.

Nous aurons, de plus quelques limites dues à l'électronique interne. Ceci nous donne les règles simples suivantes :

Si on n'utilise pas de prédiviseur, la sortie du prédiviseur est égale à son entrée, donc :

- La durée totale du clock ne pourra être inférieure à 4 T_{osc}.
- De plus, on ne pourra avoir un état haut <2 T_{osc} et un état bas <2 T_{osc}
- De plus, l'état haut et l'état bas ne pourront durer chacun moins de 20ns.

Si on utilise un prédiviseur, la sortie est égale à l'entrée divisée par la valeur du prédiviseur, donc, fort logiquement :

- La durée totale du clock ne pourra être inférieure à 4 T_{osc} divisée par le prédiviseur
- De plus, on ne pourra avoir un état haut <2 T_{osc} divisé par le prédiviseur et idem pour l'état bas
- De plus, l'état haut et l'état bas ne pourront durer chacun moins de 10ns.

Calculons **quelles sont les fréquences maximales** que nous pouvons appliquer sur la pin TOCKI à un PIC cadencé à une fréquence de **10MHz** :

Sans prédiviseur, la contrainte en terme de T_{osc} nous donne :

$$T_{osc} = 1/f = 1 / 10^7 = 10^{-7} \text{ s} = 100 \text{ ns}$$

Temps minimum en terme de T_{osc} = 4 T_{osc} = 400 ns

Ceci respecte la contrainte de temps absolu qui nous impose un temps > 40ns (20+20).

Donc, nous aurons comme fréquence maximale :

$$F = 1/T_{osc} = 1/(4*10^{-7}) = 2,5 * 10^6 = \mathbf{2,5 \text{ MHz.}}$$

Il va de soi qu'avec notre **16F876 cadencé à 20 MHz**, la fréquence maximale utilisable sans prédiviseur est de **5MHz**.

Nous voyons donc que nous pouvons dire que la fréquence maximale (en signal carré) utilisable avec le timer 0 configuré en mode compteur sans prédiviseur, est égale au quart de la fréquence du quartz employé pour son horloge.

Voyons maintenant le cas du prédiviseur. Utilisons la valeur maximale, soit 256 :

Temps minimum en terme de T_{osc} = 4 T_{osc} divisé par le prédiviseur = 400 ns / 256 = 1,56 ns

Ce temps minimum ne respecte pas le temps minimum absolu imposé, qui est de 10 ns. C'est donc ce dernier temps dont nous tiendrons compte.

Notre fréquence maximale est donc de :

$$F = 1/T = 1/(20 \cdot 10^{-9}) = 50 \cdot 10^6 \text{ Hz} = 50 \text{ MHz.}$$

Donc, vous voyez que vos PICs peuvent, à condition d'utiliser le prédiviseur à une valeur suffisante, compter à des fréquences pouvant atteindre 50 MHz

14.4 Modification « au vol » de l'assignation du prédiviseur

Vous pouvez être amenés, en cours d'exécution de votre programme, à changer l'assignation du prédiviseur entre le watchdog et le timer 0. En effet, il vous suffit de changer le bit PSA du registre pour effectuer cette opération.

Il vous faudra cependant respecter les procédures suivantes, sous peine de provoquer un reset non désiré de votre PIC au moment de la modification de PSA.

Certains profitent d'ailleurs de cette anomalie pour provoquer des resets à la demande depuis le logiciel. Je vous le déconseille fermement :

- D'une part parcequ'un programme qui nécessite un reset de ce type est en général un programme mal structuré (vous imaginez un programme sur votre PC qui nécessite un reset ?)
- D'autre part, parce que ce fonctionnement est anormal, et donc, bien évidemment, non « garanti ». En passant sur une nouvelle révision de PICs, vous risquez que votre « astuce » ne fonctionne plus.

Nous avons donc 2 possibilités : soit nous affectons le prédiviseur au watchdog alors qu'il était affecté au timer 0, soit le contraire.

14.4.1 Prédiviseur du timer 0 vers le watchdog

Commençons par ce cas de figure. Nous allons devoir distinguer 2 sous-cas particuliers, en fonction de la valeur finale de notre prédiviseur (bits PS0/PS2).

Imaginons tout d'abord que la valeur de notre prédiviseur finale soit différente de 1. Voici alors la procédure à respecter scrupuleusement :

```
Modifpsa
    clrf    TMR0           ; effacer timer0, et donc le contenu du prédiviseur
    bsf     STATUS,RP0     ; passer en banque 1
    bsf     OPTION_REG,PSA ; prédiviseur sur watchdog SANS changer valeur
    clrwdt           ; effacer watchdog et prédiviseur
    movlw   B'xxxx1abc'    ; définir la nouvelle valeur prédiviseur(selon abc)
    movwf   OPTION_REG     ; dans registre
    bcf     STATUS,RP0     ; repasser en banque 0, fin du traitement
```


Si maintenant, nous désirons que la valeur finale du prédiviseur soit égale à 1, nous sommes obligés de passer par une valeur intermédiaire quelconque, mais différente de 1. Voici donc la procédure :

```
Modifpsa
    bsf    STATUS,RP0      ; passer en banque 1
    movlw  B'xx0x0111'    ; prédiviseur quelconque, tmr en compteur
    movwf  OPTION_REG      ; dans registre
    bcf    STATUS,RP0      ; repasser en banque 0
    clrf   TMR0            ; effacer timer0, et donc le contenu du prédiviseur
    bsf    STATUS,RP0      ; passer en banque 1
    movlw  B'xxx1111'      ; passer sur watchdog, mode tmr réel utilisé
    movwf  OPTION_REG      ; dans registre
    clrwdt                  ; effacer watchdog et prédiviseur
    movlw  B'xxx1000'      ; définir la nouvelle valeur prédiviseur(1)
    movwf  OPTION_REG      ; dans registre
    bcf    STATUS,RP0      ; repasser en banque 0, fin du traitement
```

14.4.2 Prédiviseur du watchdog vers le timer 0

Dans ce cas, la procédure est plus simple :

```
clrwdt                  ; commencer par effacer le watchdog et le prédiviseur
bsf    STATUS,RP0      ; passer en banque 1
movlw  B'xxx0abc'      ; passer sur tmr 0 et nouvelle valeur prédiviseur (abc)
movwf  OPTION_REG      ; dans registre
bcf    STATUS,RP0      ; repasser banque 0, fin du traitement
```

Je vous rappelle que le non respect de ces procédures risque de causer un reset imprévu de votre PIC.

Une dernière chose à prendre en compte, c'est que, puisque le comptage ne se fait qu'au moment de la montée de Q4, il y a un décalage entre l'apparition du clock et le moment où ce clock est pris en compte. Ce décalage est donc au maximum de 1 cycle d'instruction.

Voici donc quelques compléments d'information sur le timer 0, étant donné qu'au stade où vous en êtes, vous avez de plus en plus de chance de réaliser des programmes complexes.

Notes : ...

15. Le timer 1

Voici un sujet qui m'a valu bien du courrier électronique. J'ai constaté que beaucoup de mes correspondants avaient des difficultés à mettre en œuvre ce timer.

Nous allons voir, que dans ses fonctions de base, la mise en œuvre de ce timer n'est pas plus compliquée que celle du timer 0.

Je me limiterai dans ce chapitre au fonctionnement du timer en tant que tel. Je verrai ses modes de fonctionnement annexes dans les chapitres concernés (CCP).

15.1 Caractéristiques du timer 1

Le timer 1 fonctionne dans son ensemble comme le timer 0. La philosophie est semblable. Il y a cependant de notables différences.

Tout d'abord, ce timer est capable de compter sur 16 bits, à comparer aux 8 bits du timer 0. Il sera donc capable de compter de D'0' à D'65535'. Pour arriver à ces valeurs, le timer 0 devait recourir à ses prédiviseurs, ce qui, nous l'avons vu dans la première partie du cours, limitait de ce fait sa précision lorsqu'on était contraint de recharger son contenu (perte des événements comptés par le prédiviseur).

Le timer 1 compte donc sur 16 bits, ce qui va nécessiter 2 registres. Ces registres se nomment TMR1L et TMR1H. Je pense qu'arrivés à ce stade, il est inutile de vous préciser lequel contient la valeur de poids faible, et lequel contient celle de poids fort.

Souvenez-vous que le contenu de TMR1L et de TMR1H n'est pas remis à 0 lors d'un reset. Donc, si vous voulez compter à partir de 0, il vous incombe de remettre à 0 ces 2 registres vous-même.

Le timer 1 permet également, tout comme le timer 0, de générer une interruption une fois le débordement effectué.

Le timer 1 dispose, lui aussi, de la possibilité de mettre en service un prédiviseur. Mais ce prédiviseur ne permet qu'une division maximale de 8. Le résultat final est cependant meilleur que pour le timer 0, qui ne pouvait compter que jusque 256, multiplié par un prédiviseur de 256, soit 65536.

Pour notre nouveau timer, nous arrivons à une valeur maximale de 65536 multiplié par 8, soit 524288.

Vous allez me rétorquer que ceci vous permet d'atteindre des temps plus longs, mais ne vous permet pas d'utiliser des temps aussi courts que pour le timer 0, qui peut générer une interruption tous les 256 clocks d'horloge, notre timer 1 nécessitant 65536 clocks au minimum.

En fait, il suffit que lors de chaque interruption vous placiez la valeur 0xFF dans TMR1H, et vous vous retrouvez avec un compteur sur 8 bits. Vous pouvez naturellement utiliser toute autre valeur qui vous arrange.

La souplesse du timer 1 est donc plus importante que celle du timer 0. Et vous allez voir que c'est loin d'être terminé.

Tout comme pour le timer 0, les registres de comptage TMR1L et TMR1H contiennent une valeur indéterminée après un reset de type P.O.R ou B.O.R. De même, un autre reset ne modifie pas le contenu précédent de ces registres, qui demeurent donc inchangés.

15.2 Le timer 1 et les interruptions

Il est bien entendu qu'un des modes privilégiés de l'utilisation des timers, est la possibilité de les utiliser en tant que générateurs d'interruptions.

Je vous rappelle donc ici que le timer 1 permet, exactement comme le permet le timer 0, de générer une interruption au moment où timer « déborde », c'est-à-dire au moment où sa valeur passe de 0xFFFF à 0x0000 (souvenez-vous que nous travaillons sur 16 bits).

Quelles sont les conditions pour que le timer 1 génère une interruption ? Et bien, tout simplement que cette interruption soit autorisée. Donc :

- Il faut que le bit d'autorisation d'interruption du timer 1 (TMR1IE) soit mis à 1. Ce bit se trouve dans le registre PIE1 (banque 1).
- Pour que le registre PIE1 soit actif, il faut que le bit PEIE d'autorisation des interruptions périphériques soit positionné dans le registre INTCON.
- Il faut également, bien entendu, que le bit d'autorisation générale des interruption (GIE) soit positionné dans le registre INTCON.

Moyennant quoi, une interruption sera générée à chaque débordement du timer1. Cet événement sera indiqué par le positionnement du flag TMR1IF dans le registre PIR1.

Comme d'habitude, il vous incombe de remettre ce flag à 0 dans la routine d'interruption, sous peine de rester indéfiniment bloqué dans la dite routine. Mais, comme pour les autres interruptions, le fichier maquette que je vous fournis effectue déjà ce travail.

Voici une séquence qui initialise les interruptions sur le timer 1 :

```
bsf      STATUS,RP0    ; passer en banque 1
bsf      PIE1,TMR1IE   ; autoriser interruptions timer 1
bcf      STATUS,RP0    ; repasser banque 0
bsf      INTCON,PEIE    ; interruptions périphériques en service
bsf      INTCON,GIE     ; interruptions en service
```

15.3 Les différents modes de fonctionnement du timer1

Le timer 1 peut, tout comme le timer 0, fonctionner en **mode timer** (c'est-à-dire en comptage des cycles d'instruction), ou en **mode compteur** (c'est-à-dire en comptant des impulsions sur une pin externe).

Cependant, le timer 1 dispose de **2 modes différents de mode de comptage** : un mode de type **synchrone** et un mode de type **asynchrone**.

Vous avez déjà eu une idée de ce qu'est un mode synchrone au chapitre traitant du timer 0. Rappelez-vous que le comptage ne s'effectuait, en sortie du prédiviseur, qu'au moment de la montée de Q4, donc en synchronisme avec l'horloge interne.

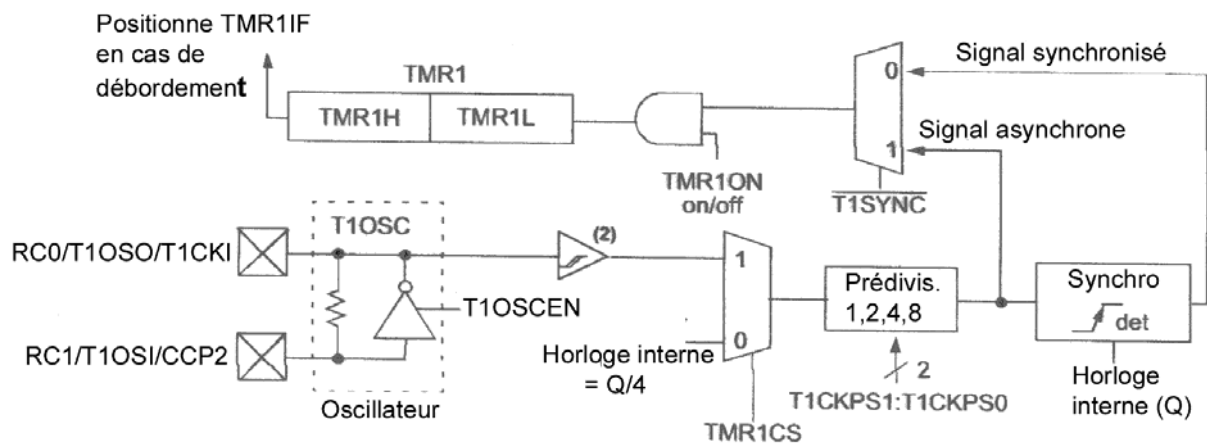
En plus, nous avons la possibilité, au niveau du timer 1, d'utiliser un quartz sur les pins T1OSI et T1OSO afin de disposer d'une horloge séparée de l'horloge principale.

En tout état de cause, souvenez-vous qu'un timer utilisé en mode « timer », n'effectue jamais qu'un simple comptage de cycles d'instruction.

Pour résumer, nous pouvons donc dire que le timer peut fonctionner en tant que :

- Timer basé sur l'horloge interne (compteur de cycles d'instruction)
- Timer basé sur une horloge auxiliaire
- Compteur synchrone
- Compteur asynchrone.

Les modes de fonctionnement sont définis en configurant correctement le registre **TICON**, que nous allons examiner maintenant. Mais tout d'abord, je vous donne le schéma-bloc du timer 1 :



15.4 Le registre TICON

Comme je le disais, c'est ce registre, situé en banque 0, qui va permettre de configurer le timer1 en fonction de nos besoins. Voici les bits qui le composent :

- b7 : Inutilisé : lu comme « 0 »
b6 : Inutilisé : lu comme « 0 »
b5 : T1CKPS1 : Timer 1 oscillator Clock Prescale Select bit 1
b4 : T1CKPS0 : Timer 1 oscillator Clock Prescale Select bit 0

b3 : T1OSCEN : Timer 1 OSCillator ENable control bit
 b2 : T1SYNC : Timer 1 external clock input SYNCronisation control bit
 b1 : TMR1CS : TiMeR 1 Clock Source select bit
 b0 : TMR1ON : TiMeR 1 ON bit

Je ne sais pas quel est votre sentiment, mais moi, la première fois que j'ai rencontré ces bits, j'ai trouvé leur nom particulièrement « barbare ». Mais bon, leur utilisation est heureusement plus simple à retenir que les noms. Voyons donc.

Tout d'abord, les bits **T1CKPS1** et **T1CKPS0** déterminent à eux deux la **valeur du prédiviseur** dont nous avons déjà parlé. Le prédiviseur, comme son nom l'indique, et comme pour le timer0, permet, je le rappelle, de ne pas compter chaque événement reçu, mais seulement chaque multiple de cet événement. La valeur de ce multiple est la valeur du prédiviseur.

T1CKPS1	T1CKPS0	Valeur du prédiviseur
0	0	1
0	1	2
1	0	4
1	1	8

Vous pouvez donc vérifier que les choix sont plus limités que pour le timer 0, mais ceci est compensé par le fait que le timer 1, je le rappelle encore, compte sur 16 bits au lieu de 8.

Nous trouvons ensuite **T1OSCEN**. Ce bit permet de **mettre en service l'oscillateur interne**, et donc permet d'utiliser un second quartz pour disposer d'un timer précis travaillant à une fréquence distincte de la fréquence de fonctionnement du PIC. Mais rassurez-vous, je vais vous détailler tout ça un peu plus loin.

Pour l'instant, reprenez que si vous placez ce bit à « 1 », vous mettez en service l'oscillateur interne, ce qui aura, nous le verrons plus loin, toute une série de conséquences.

Maintenant, voyons **T1SYNC**, qui permet de choisir si le comptage des événements sera effectuée de façon **synchrone ou asynchrone** avec l'horloge principale du PIC. Il est de fait évident que lorsque TMR1 est utilisé en mode « timer », T1SYNC n'a aucune raison d'être, le comptage étant déterminé par l'oscillateur principal du PIC, il est automatiquement synchrone.

Le bit TMR1CS permet de définir quel est le mode de comptage du timer 1 :

- Soit on place TMR1CS à « 0 », et dans ce cas, il compte les cycles d'instructions du PIC. Dans ce cas, on dira, tout comme pour le timer 0, que TMR1 travaille en mode « timer ».
- Soit, on place ce bit à « 1 », et TMR1 compte alors les flancs montants du signal appliqué sur la pin RC0/T1OSO/T1CKI (la pin porte 3 noms, en fonction de son utilisation).

Comme je le disais plus haut, si TMR1CS est à « 0 », alors T1SYNC est ignoré.

Reste le plus simple, le bit TMR1ON, qui, s'il est mis à « 1 », autorise le timer 1 à fonctionner, alors que s'il est mis à « 0 », fige le contenu du timer 1, et donc, le met à l'arrêt.

15.5 Le timer 1 en mode « timer »

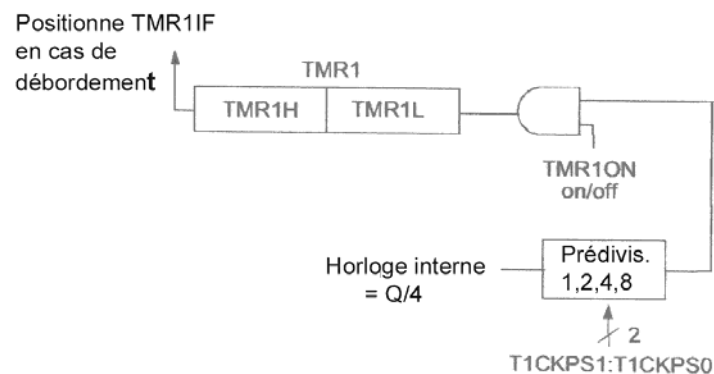
Nous allons maintenant voir avec plus de détails les différents modes de fonctionnement de ce timer, en les examinant chacun en particulier. Et tout d'abord, en commençant par le plus simple, le mode « timer ».

Dans ce mode, nous trouvons un fonctionnement tout ce qu'il y a de plus simple, puisque nous avons affaire à un **comptage des cycles d'instructions internes du PIC**, exactement comme nous avons pour le timer 0.

Pour choisir ce mode, nous devons configurer T1CON de la façon suivante :

T1CON : B'00ab0001'

Si vous avez suivi, vous avez déjà compris que « ab » représentent la valeur du prédiviseur choisie, et que TMR1CS devra être mis à « 0 ». TMR1ON permet de mettre le timer en service. Ceci configure le fonctionnement interne du timer 1 comme illustré ci-dessous :



Voici un exemple d'initialisation du timer 1, configuré avec un prédiviseur de 4

```
clrf    TMR1L        ; effacer timer 1, 8 lsb
clrf    TMR1H        ; effacer timer 1, 8 msb
movlw   B'0010000'   ; valeur pour T1CON
movwf   T1CON        ; prédiviseur = 4, mode timer, timer off
bsf     T1CON,TMR1ON ; mettre timer 1 en service
```

15.6 Le timer 1 en mode compteur synchrone

Pour travailler dans le mode « compteur synchrone », nous devons configurer T1CON de la façon suivante :

T1CON : B'00ab0011'

Evidemment, pour pouvoir compter des événements sur la pin T1CKI, il faut que cette pin soit configurée en entrée via le registre TRISC.

Donc, en reprenant le schéma, vous voyez que la pin RC1 n'est pas utilisée, pas plus que l'oscillateur et la résistance associée. Donc, vous ignorez la partie située dans le rectangle pointillé.

Nous entrons donc sur T1CKI (T1 ClOcK In) et nous arrivons sur un trigger de Schmitt (2). Nous arrivons ensuite sur la sélection du signal via TMR1CS. Comme nous avons mis « 1 » pour ce bit, nous validons donc l'entrée de T1CKI et nous ignorons le comptage des instructions internes (mode timer).

Ensuite, nous arrivons au prédiviseur, dont la valeur est fixée par T1CKPS1 et T1CKPS0. Vous constatez qu'à ce niveau, le comptage se fait de façon asynchrone. En effet, par synchrone il faut comprendre : « qui est synchronisé par l'horloge interne du PIC. ».

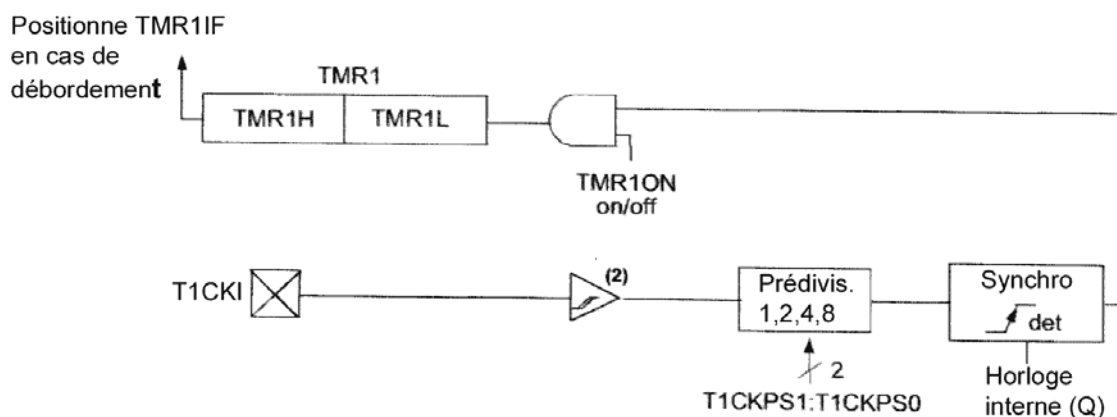
Or, ici, les événements sont comptés dans le prédiviseur indépendamment de l'horloge du PIC.

Par contre, la sortie du prédiviseur est envoyée également dans un module de synchronisation. Le bloc de sélection T1SYNC permet de définir si on utilise la sortie du prédiviseur avant la synchronisation (mode asynchrone) ou après la synchronisation (mode synchrone).

Nous avons mis notre T1SYNC à 0. Comme il est actif à l'état bas, ceci nous sélectionne le signal synchronisé. Le signal est ensuite validé ou non grâce à TMR1ON, signal qui va servir à incrémenter le mot de 16 bits constitué de TMR1H et TMR1L.

Si ce mot de 16 bits passe de 0xFFFF à 0x0000 (débordement), le flag TMR1IF est alors positionné, et peut générer une interruption, si c'est le choix de l'utilisateur.

Voici donc le schéma-bloc du mode « compteur synchrone » :



Vous voyez que c'est très simple. Reste à expliquer le rôle du synchronisateur. En effet, vous allez me dire : « A quoi sert de synchroniser ou de ne pas synchroniser la sortie du prédiviseur ? »

En fait, la différence principale est que si vous choisissez de synchroniser, alors, fort logiquement, vous ne comptez pas si le PIC est à l'arrêt. Or, quand le PIC est-il à l'arrêt ? Et bien quand vous le placez en mode « sleep ».

Donc, le timer 1 utilisé en mode « compteur synchrone » ne permet pas de réveiller le PIC sur une interruption du timer 1. C'est logique, car, puisqu'il n'y a pas de comptage, il n'y a pas de débordement.

Notez que la présence du synchronisateur retarde la prise en compte de l'événement, puisqu'avec ce dernier la sortie du prédiviseur ne sera transmise au bloc incrémenteur que sur le flanc montant suivant de l'horloge interne.

Il me reste à vous préciser que le comptage s'effectue uniquement sur un flanc montant de T1CKI, donc sur le passage de T1CKI de « 0 » vers « 1 ». Il ne vous est pas possible ici de choisir le sens de transition, comme le permettait le bit TOSE pour le timer 0.

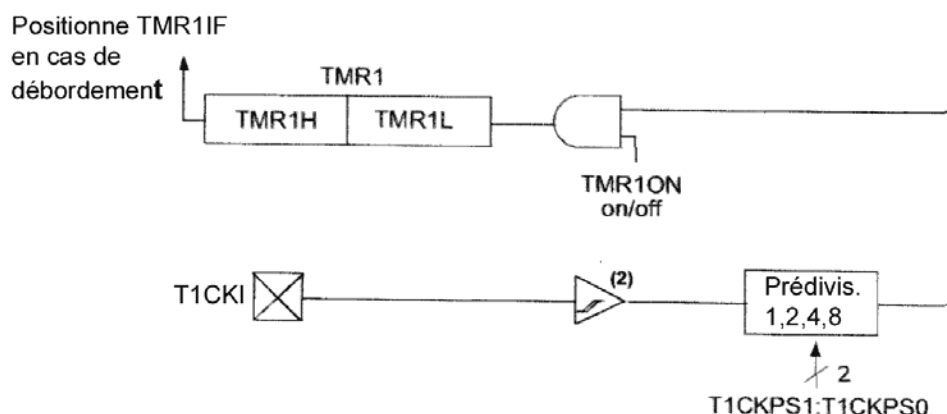
De plus, le flanc montant de T1CKI n'est pris en compte que s'il a été précédé d'au moins un flanc descendant.

Ce dernier point mérite un complément d'information. Si vous avez un signal connecté au PIC qui est au niveau 0, et dont vous devez compter les passages à « 1 », alors, lorsque votre PIC recevra son alimentation, le premier flanc montant de T1CKI ne sera pas comptabilisé, puisque pas précédé d'un flanc descendant.

Notez que pour éviter de « perdre » un comptage, les temps minimum des signaux appliqués répondent exactement aux mêmes critères que pour le timer 0. Je vous renvoie donc au chapitre précédent à ce sujet.

15.7 Le timer 1 en mode compteur asynchrone

Partant des explications précédentes, le schéma-bloc obtenu ne présente aucune difficulté, il suffit bien entendu de supprimer le bloc de synchronisation :



Donc, vous voyez qu'un événement est comptabilisé immédiatement, sans attendre une quelconque transition de l'horloge interne.

Les contraintes temporelles, concernant les caractéristiques des signaux appliqués sont toutefois toujours d'application (voyez chapitre précédent).

La plus grosse différence, c'est que le comptage s'effectue même si le PIC est stoppé, et que son horloge interne est arrêtée. Il est donc possible de réveiller le PIC en utilisant le débordement du timer1.

Par contre, cette facilité se paye au niveau de la facilité de lecture et d'écriture des registres TMR1H et TMR1L, mais je verrai ceci un peu plus loin.

La valeur à placer dans le registre T1CON pour travailler dans ce mode est bien évidemment :

T1CON : B'00ab0111'

Il faut de nouveau ne pas oublier de configurer T1CKI (RC0) en entrée, via le bit 0 de TRISC.

15.8 Le timer 1 et TOSCEN

Nous avons presque fait le tour des modes de fonctionnement du timer 1. Reste à utiliser l'oscillateur interne.

La valeur à placer dans T1CON est donc :

T1CON : B'00ab1x11'

Remarquez la présence d'un « x » qui vous permet de choisir si vous désirez travailler en mode synchrone ou asynchrone. Les remarques que j'ai faite à ce sujet pour l'utilisation en mode compteur restent d'application.

Bien qu'il s'agisse d'une mesure de temps, donc d'un timer, il faut bien évidemment configurer T1CON pour qu'il puisse compter les oscillations reçues sur la pin T1OSI. Ceci rejoint ce que je vous ai déjà signalé, à savoir qu'un timer n'est rien d'autre qu'un compteur particulier.

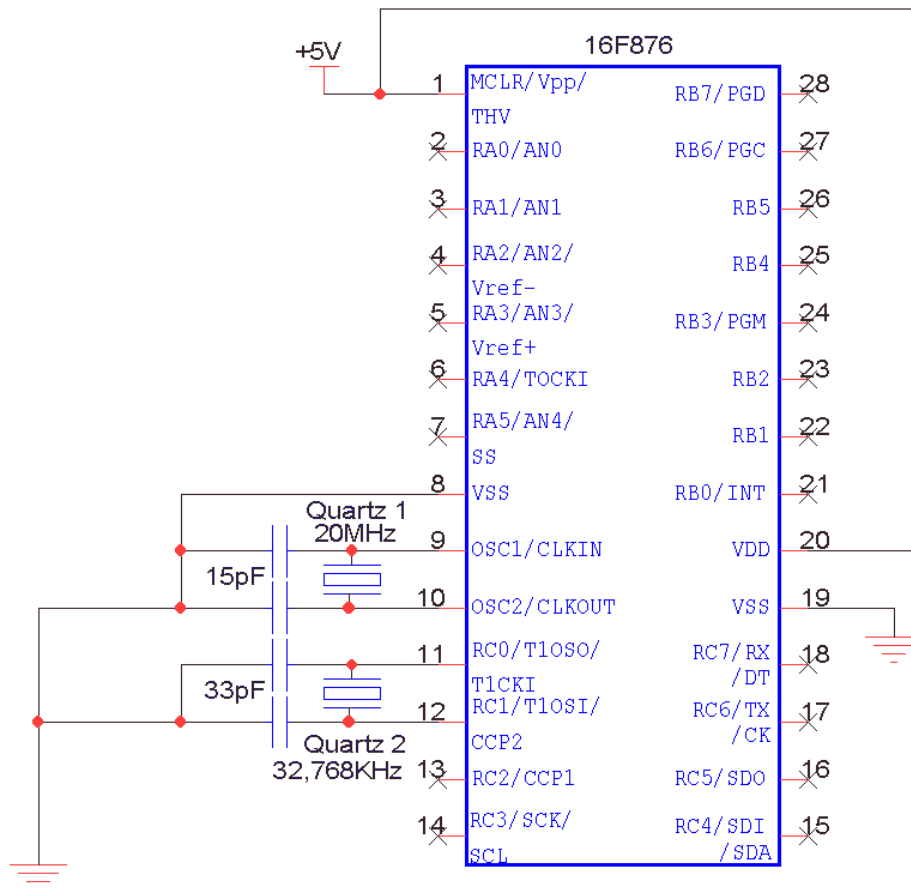
Arrivé ici, vous vous posez peut-être quelques questions concernant cet oscillateur. Je vais donc tenter d'anticiper :

A quoi sert cet oscillateur ?

Et bien, tout simplement à utiliser une base de temps différente pour le timer et pour le PIC. Ceci permet donc de choisir un quartz multiple exact de la fréquence à mesurer, tout en conservant une vitesse de traitement maximale du PIC.

Souvenez-vous que je vous ai déjà parlé, dans la première partie, dédiée au 16F84, de la solution « de luxe » consistant, pour améliorer la précision, à utiliser un second oscillateur pour réaliser des mesures de précision. L'avantage, ici, c'est que l'oscillateur est présent en interne, il ne suffit donc que de lui ajouter un quartz.

Voici un schéma typique de l'utilisation du timer 1 avec un quartz sur OSCEN :



Vous notez donc la présence de 2 quartz. Le quartz 1 vous permet d'utiliser le PIC à sa fréquence de travail maximale (ici, 20MHz). Le second vous donne une base de temps différente pour l'utilisation du timer 1.

Se pose donc une seconde question : Quelles sont les valeurs du second quartz que je peux utiliser ?

En fait, l'oscillateur a été conçu pour fonctionner efficacement à une valeur centrée sur 32KHz (attention KHz, pas MHz). Mais vous pouvez augmenter cette fréquence jusqu'à 200 KHz.

Cet oscillateur fonctionne donc à une vitesse plus lente que l'oscillateur principal. La table 6-1 du datasheet vous donne les valeurs des condensateurs à utiliser pour quelques fréquences typiques.

Vous voyez que j'ai choisi une fréquence de 32768 Hz pour mon second quartz. En fait, je n'ai pas choisi cette valeur par hasard. En effet, si vous vous souvenez que le timer 1 compte sur 16 bits (le contraire serait étonnant, vu le nombre de fois que je vous l'ai répété), nous aurons un débordement du timer 1 au bout du temps : $65536 / \text{fréquence}$. Soit tout juste 2 secondes. Voici qui nous donne une base de temps très précise (n'utilisant pas de prédiviseur), et qui ne provoque qu'une interruption toutes les 2 secondes.

Si vous voulez une interruption toutes les secondes, il vous suffira de positionner le bit 7 de TMR1H lors de chaque interruption. Ainsi, le timer 1 comptera de 0x8000 à 0xFFFF, soit

un comptage de 32768 cycles. Ceci nous donnera un temps de comptage de $32768/32768 = 1$ seconde « pile ». Il vous suffit donc dans ce cas d'ajouter la ligne

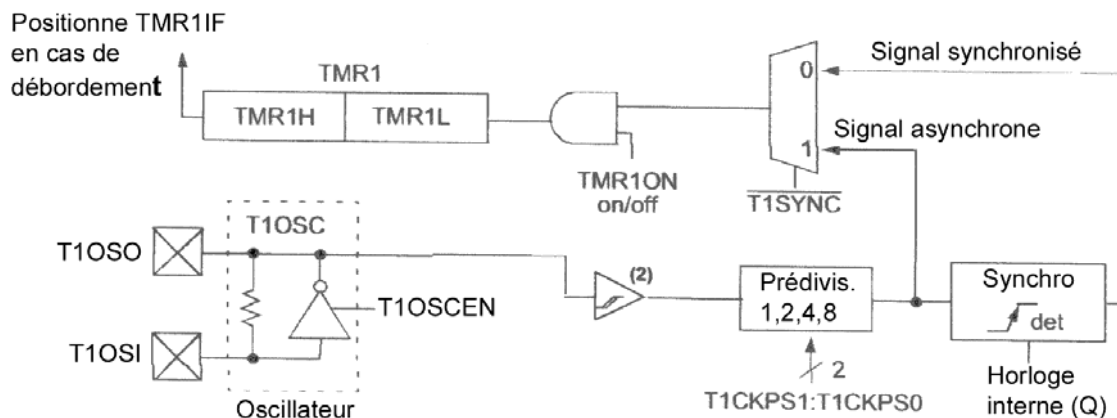
```
bsf    TMR1H,7    ; forcer bit 7 de TMR1H = bit 15 de TMR1
```

à votre routine d'interruption du timer 1.

Ce mode constitue donc le mode idéal pour créer une mesure de temps réel. Notez de plus que si vous avez configuré le mode asynchrone, le timer continuera de compter même si votre PIC est en mode « sleep ».

La précision obtenue est fonction du quartz, mais est typiquement de 20 parts par million. Ceci nous donne une erreur maximale, pour le cas de la réalisation d'une horloge, de 1,7 seconde par jour.

Je vous donne le schéma-bloc correspondant au mode OSCEN, schéma que vous aurez probablement trouvé vous-même :



Il me reste cependant à vous signaler que dans ce mode particulier, les pins RC0 (T1OSO) et RC1 (T1OSI) sont automatiquement configurées en entrée, indépendamment des bits correspondants de TRISC. Il est donc inutile de configurer les bits 0 et 1 de ce registre.

15.9 Utilisation du débordement

Programmer le timer1 et comprendre son fonctionnement ne suffit pas. Encore faut-il l'utiliser pour mesurer ou compter.

La première méthode d'utilisation de notre TMR1 est la plus courante. Je ne la répète que pour pouvoir faire la distinction avec les pièges qui se trouvent dans la seconde méthode d'utilisation.

Cette méthode consiste à utiliser le débordement de la valeur du timer pour positionner le flag TMR1IF, et, éventuellement, pour déclencher une interruption.

La philosophie utilisée est donc la suivante :

- On initialise le timer avec une valeur de départ (éventuellement 0)
- On attend son débordement
- A ce moment on détecte que le temps prédéfini est écoulé.

Imaginons donc la séquence d'initialisation de notre timer 1 utilisé en mode timer. Nous décidons pour cet exemple que nous avons besoin d'un prédiviseur d'une valeur de 4. Nous décidons également d'autoriser l'interruption du timer 1. Si nous nous souvenons qu'il nous incombe de remettre le timer à 0, nous aurons, par exemple, la séquence suivante :

```
clrf    TMR1L    ; effacer timer 1, 8 lsb
clrf    TMR1H    ; effacer timer 1, 8 msb
movlw   B'0010000' ; valeur pour T1CON
movwf   T1CON    ; prédiviseur = 4, mode timer, timer off
bsf     STATUS,RP0 ; passer en banque 1
bsf     PIE1,TMR1IE ; autoriser interruptions timer 1
bcf     STATUS,RP0 ; repasser banque 0
bsf     INTCON,PEIE ; interruptions périphériques en service
bsf     INTCON,GIE ; interruptions en service
bsf     T1CON,TMR1ON ; mettre timer 1 en service
```

Le raisonnement serait identique pour un fonctionnement en mode compteur.

Bien entendu, il ne s'agit que d'un exemple, plusieurs lignes peuvent être inversées, vous pouvez mettre GIE et PEIE en service en même temps etc. Certains d'entre vous, à l'œil acéré, ont peut-être remarqué que je n'avais pas remis le flag TMR1IF à « 0 » avant de lancer les interruptions.

C'est judicieusement pensé, mais ce n'est pas utile. En effet, au moment de la mise sous tension, PIR1 est automatiquement mis à 0. De plus, T1CON est également mis à 0, ce qui implique que le timer 1 est à l'arrêt tant qu'on ne le met pas en service intentionnellement.

Donc, il est impossible que le timer 1 ait provoqué un débordement de TMR1IF avant sa mise en service.

Si, par contre, on ne désirait pas compter de 0x0000 à 0xFFFF, on peut initialiser les registres du timer 1 pour diminuer le temps de comptage. Imaginons que nous désirions compter 3978 cycles d'instructions :

En fait, nous devons donc initialiser le timer1 pour qu'il démarre le comptage 3978 cycles avant sa valeur de débordement. Vous voyez, en raisonnant un peu, qu'il suffit de mettre dans les registres du timer1, la valeur 0x10000 de laquelle on soustrait la valeur à compter. Exemple, pour bien visualiser, si vous désirez compter 1 cycle, vous mettrez $0x10000 - 0x01 = 0xFFFF$, ce qui est logique, puisqu'au cycle suivant, nous aurons débordement.

Dans notre cas, il s'agira donc de mettre $0x10000 - D'3978$ dans notre timer1, constitué de TMR1H pour le poids fort, et TMR1L pour le poids faible. Plutôt que de calculer cette valeur, autant laisser faire MPLAB, il est là pour ça (entre autres). Définissons notre valeur sur 16 bits :

```
VALUE EQU    0x10000 - D'3978'    ; calcul de la valeur sur 16 bits
```

Il suffit alors de remplacer :

```
clrf    TMR1L    ; effacer timer 1, 8 lsb
clrf    TMR1H    ; effacer timer 1, 8 msb
```

par :

```
movlw   LOW  VALUE ; 8 bits faibles de la valeur calculée
movwf   TMR1L    ; dans TMR1L
movlw   HIGH VALUE ; 8 bits forts de la valeur calculée
movwf   TMR1H    ; dans TMR1H
```

Bien entendu, dans notre exemple, vous devrez remettre le prédiviseur à 1, sinon nous aurons interruption après $4 * 3978$, soit 15912 cycles.

De la sorte, notre compteur débordera 3978 cycles après avoir été lancé. Vous voyez donc que vous pouvez obtenir une bien meilleure précision qu'avec le timer 0, et, de plus, beaucoup plus facilement.

15.10 Utilisation d'une lecture

Nous pouvons aussi vouloir décider d'utiliser notre timer d'une autre façon. En effet, supposons que nous voulions construire un chronomètre. Nous aurons alors la structure de programme suivante :

- On presse le bouton « démarrer »
- On lance le timer 1 en mode timer, à partir de 0.
- On presse le bouton « lecture de temps intermédiaire »
- On sauve la valeur de TMR1H et TMR1L
- On presse le bouton « stop »
- On arrête le timer 1
- On lit la valeur dans TMR1H et TMR1L

Evidemment, il s'agit d'un exemple, on considère ici, pour simplifier, que le temps mesuré est compris entre 0 et 0xFFFF cycles. Si le temps était plus long, nous devrions en plus compter le nombre de fois que le timer1 a débordé. Nous aurions alors le temps total (en cycles) = (nombre de débordements * 0x10000) + valeur actuelle du mot de 16 bits : TMR1H TMR1L.

La philosophie utilisée dans ce mode est donc la suivante :

- On démarre le timer à partir de 0
- On détecte un événement
- On lit la valeur du timer pour calculer le temps écoulé

Le programme serait alors, par exemple (START, STOP et LAP sont des « define » qui pointent sur des pins utilisées en entrée) :

```
start
```

```

    clrf    TMR1L        ; effacer timer 1, 8 lsb
    clrf    TMR1H        ; effacer timer 1, 8 msb
attendre
    btfss   START        ; tester si le bouton « démarrer » est pressé
    goto    attendre     ; non, attendre
    bsf     T1CON,TMR1ON ; oui, lancer le timer
attendre2
    btfss   LAP          ; tester si temps intermédiaire pressé
    goto    attendre2    ; non, attendre
    movf    TMR1L,w       ; lire 8 lsb
    movwf   VALL          ; sauver dans variable
    movf    TMR1H,w       ; lire 8 msb
    movwf   VALH          ; sauver dans variable
attendre3
    btfss   STOP         ; tester si bouton « stop » est pressé
    goto    attendre3    ; non, attendre
    bcf     T1CON,TMR1ON ; oui, arrêter le timer 1
    movf    TMR1L,w       ; lire 8 lsb
    movwf   FINALL        ; sauver dans variable
    movf    TMR1H,w       ; lire 8 msb
    movwf   FINALH        ; sauver dans variable

```

Examinons ce programme. N'oubliez pas qu'il s'agit d'un exemple symbolique, donc, je sais que ce programme impose de presser « LAP » avant de presser « STOP », que les boucles introduisent une erreur, qu'il était préférable d'utiliser d'autres méthodes etc. En fait, ce n'est pas ce que je voulais vous montrer.

Regardez la dernière lecture, celle qui place le timer 1 dans 2 variables, pour obtenir une valeur sur 16 bits. Le résultat FINALH FINALL reflète effectivement le temps qui s'est écoulé depuis la mise en service du timer 1 jusqu'à son arrêt.

Imaginez qu'à la fin de l'exécution de cette séquence, nous ayons :

```

FINALL = 0x12
FINALH = 0xDE

```

Nous pouvons dire qu'il s'est écoulé 0xDE12 cycles, soit 56850 cycles entre le démarrage et l'arrêt du timer 1.

Regardez maintenant ce qui se passe au niveau de la mesure du temps intermédiaire. Supposons que nous avons après exécution de notre programme :

```

VALL = 0xFE
VALH = 0x45

```

Vous en concluez donc qu'il s'est écoulé 0x45FE cycles, soit 17918 cycles entre le démarrage du timer 1 et la mesure de temps intermédiaire (on néglige les temps de réaction de stockage des valeurs).

Vous êtes bien certain de ceci ? Oui ? Alors vérifions ce qui se passe lors de la lecture :

```

movf    TMR1L,w         ; lire 8 lsb

```

Comme notre valeur VALL vaut 0xFE, c'est donc que notre TMR1L vaut 0xFE. Nous ne connaissons pas encore, à ce stade, la valeur de TMR1H. Ensuite :

```
movwf VALL          ; sauver dans variable
```

Nous sauvons donc bien 0xFE dans VALL, mais un cycle d'instruction s'est exécuté, notre TMR1L continue de compter, et vaut donc maintenant 0xFF. Voyons la suite :

```
movf  TMR1H,w       ; lire 8 msb
```

A ce moment, une nouvelle instruction est exécutée, donc TMR1L a de nouveau été incrémenté, et est passé donc à 0x00, entraînant l'incrémentement de TMR1H. C'est donc cette valeur incrémentée que nous allons lire. En effet, $0x44FF + 1 = 0x4500$.

```
movwf VALH          ; sauver dans variable
```

Donc, nous sauvons 0x45, puisque c'est le résultat que nous avons obtenu.

Donc, nous avons déduit que la valeur du timer au moment de la lecture était de 0x45FE, alors que nous constatons en réalité que ce compteur valait 0x44FE, soit 17662. Nous nous sommes donc trompés de 256 cycles.

Vous allez alors me répondre qu'il suffit de vérifier que TMR1L soit inférieur à 0xFE. S'il est supérieur, on soustrait 1 de TMR1H.

En fait, votre raisonnement n'est valable que dans cet exemple précis. En effet, le comptage des cycles comme je viens de le faire ci-dessus inclut plusieurs conditions :

- Il faut que le prédiviseur utilisé soit égal à 1. En effet, dans le cas contraire, si vous avez un TMR1L égal à 0xFF, il vous est impossible de savoir s'il y a eu incrémentation ou pas de TMR1L. Donc, vous ne pouvez savoir si vous devez ou non rectifier TMR1H.
- Ensuite, si les interruptions sont en service, vous ne pouvez pas être certain qu'il n'y a pas eu interruption entre la lecture de TMR1L et celle de TMR1H. Donc, vous ne pouvez pas savoir combien de cycles séparent les 2 instructions.
- Si nous travaillons en mode compteur, nous ne pouvons pas savoir combien d'impulsions ont été comptabilisées entre les lectures.

Forts de tout ceci, nous pouvons en déduire qu'il est préférable, pour lire les 16 bits du timer 1 d'opérer en mettant le dit timer hors-service :

```
bcf    T1CON,TMR1ON ; arrêter le timer 1
movf    TMR1L,w      ; lire 8 lsb
movwf   FINALL       ; sauver dans variable
movf    TMR1H,w      ; lire 8 msb
movwf   FINALH       ; sauver dans variable
bsf     T1CON,TMR1ON ; remettre timer en service
```

Mais, bien entendu, ceci n'est pas toujours possible. En effet, dans notre exemple, la mesure du temps intermédiaire impose de ne pas arrêter le timer pour lire la valeur. Nous allons donc changer de stratégie.

Nous pouvons utiliser l'algorithme suivant :

- On lit le poids fort TMR1H et on le sauve
- On lit le poids faible
- On relit le poids fort : s'il n'a pas changé, c'est OK, sinon, on recommence.

Si on se dit que le TMR1L ne peut déborder 2 fois de suite, on peut se passer de recommencer le test. Mais, pour que cette condition soit vraie, il faut que le temps séparant les 2 lectures ne soit pas trop grand, donc il faut empêcher qu'une interruption s'intercale :

- On interdit les interruptions
- On lit le poids fort TMR1H et on le sauve
- On lit le poids faible TMR1L et on le sauve
- On relit le poids fort : s'il n'a pas changé, on va en « suite »
- On relit le poids fort
- On relit le poids faible

Suite :

- On réautorise les interruptions

Ca paraît plus long, mais si vous encodez la boucle précédente, vous verrez que le temps nécessaire est plus long (surtout en cas d'interruption). Voici la séquence tirée de ce pseudo-code :

```
bcf    INTCON,GIE    ; interdire les interruptions
movf   TMR1H,w        ; charger poids fort
movwf  TEMPOH          ; sauver valeur
movf   TMR1L,w        ; lire poids faible
movwf  TEMPOL          ; sauver valeur
movf   TMR1H          ; relire poids fort
xorwf  TEMPOH,w        ; comparer les 2 poids forts
btfsc  STATUS,Z        ; tester si identiques
goto   suite           ; oui, fin du traitement
movf   TMR1H,w        ; charger poids fort
movwf  TEMPOH          ; sauver valeur
movf   TMR1L,w        ; lire poids faible
movwf  TEMPOL          ; sauver valeur
suite
bsf    INTCON,GIE    ; réautoriser les interruptions
```

Notre programme précédent devient donc :

```
start
  clrf  TMR1L          ; effacer timer 1, 8 lsb
  clrf  TMR1H          ; effacer timer 1, 8 msb
attendre
  btfss START          ; tester si le bouton « démarrer » est pressé
  goto  attendre       ; non, attendre
  bsf   T1CON,TMR1ON   ; oui, lancer le timer
attendre2
  btfss LAP            ; tester si temps intermédiaire pressé
  goto  attendre2      ; non, attendre
  bcf   INTCON,GIE     ; interdire les interruptions
  movf  TMR1H,w        ; charger poids fort
  movwf VALH           ; sauver valeur
  movf  TMR1L,w        ; lire poids faible
  movwf VALL           ; sauver valeur
```

```

    movf   TMR1H           ; relire poids fort
    xorwf  VALH,w          ; comparer les 2 poids faibles
    btfsc  STATUS,Z        ; tester si identiques
    goto   suite           ; oui, fin du traitement
    movf   TMR1H,w         ; charger poids fort
    movwf  VALH            ; sauver valeur
    movf   TMR1L,w         ; lire poids faible
    movwf  VALL            ; sauver valeur
suite
    bsf    INTCON,GIE      ; réautoriser les interruptions
attendre3
    btfss  STOP            ; tester si bouton « stop » est pressé
    goto   attendre3       ; non, attendre
    bcf    T1CON,TMR1ON    ; oui, arrêter le timer 1
    movf   TMR1L,w         ; lire 8 lsb
    movwf  FINALL          ; sauver dans variable
    movf   TMR1H,w         ; lire 8 msb
    movwf  FINALH          ; sauver dans variable

```

Vous voyez qu'il faut toujours rester attentif, pour éviter de tomber dans toutes sortes de pièges. Je vous montre les plus classiques, mais il en existera toujours dans une application ou dans une autre.

15.11 Ecriture du timer 1

Nous venons de parler de lecture des 2 registres du timer 1. Vous vous doutez bien que les écritures comportent également des pièges.

La plus simple des solutions est évidemment **d'arrêter le timer lors d'une écriture**, de la façon suivante :

```

bcf    T1CON,TMR1ON      ; stopper le timer 1
movlw  VALL              ; charger valeur basse
movwf  TMR1L             ; dans registre
movlw  VALH              ; charger valeur haute
movwf  TMR1H             ; dans registre
bsf    T1CON,TMR1ON      ; remettre timer 1 en service

```

Mais, vous voudrez peut-être, pour une application particulière, inscrire une valeur dans le timer sans devoir l'arrêter. Quel problème allez-vous rencontrer ? Voyons donc le cas suivant :

- On écrit la valeur basse dans TMR1L
- On écrit la valeur haute dans TMR1H

Ca a l'air tout simple, mais vous devez raisonner de façon identique que pour la lecture, à savoir :

Il se peut que votre registre TMR1L déborde entre le moment de son écriture et le moment de l'écriture de TMR1H. Supposons que vous vouliez écrire la valeur 0x53FF.

- Vous commencez par écrire 0xFF dans TMR1L
- Vous écrivez 0x53 dans TMR1H, mais durant ce temps, TMR1L a débordé
- La valeur finale écrite est donc : 0x5300

Nous retrouvons donc le même type d'erreur que pour la lecture, et les mêmes remarques. Vous ne pouvez donc pas savoir à partir de quelle valeur vous aurez débordement, sauf pour le cas particulier du mode timer avec prédiviseur à 1.

A ce stade, j'en vois qui disent : « Eh, il suffit d'écrire dans l'autre sens, TMR1H suivi de TMR1L ».

C'est bien pensé, mais, en fait, alors se pose un autre problème. Etant donné que le timer est en train de tourner, vous ne connaissez pas la valeur qui se trouve dans TMR1L au moment de l'écriture de TMR1H. Il se peut donc que le TMR1L déborde avant que vous n'ayez terminé. Si on reprend l'exemple précédent :

- On écrit 0x53 dans TMR1H, TMR1L déborde à ce moment, et incrémente TMR1H
- On écrit 0xFF dans TMR1L
- Le résultat final est donc 0x54FF.

De nouveau, nous retrouvons notre erreur.

En fait, la solution est simple : si on veut éviter que le TMR1L déborde, il suffit de lui placer en premier lieu une valeur suffisamment basse pour être sûr d'éviter tout débordement (par exemple 0) . Bien entendu, il faudra également interdire les interruptions :

- On interdit les interruptions
- On écrit 0x00 dans TMR1L
- On écrit la valeur haute dans TMR1H
- On écrit la valeur basse dans TMR1L
- On réautorise les interruptions

Voici donc la séquence résultante :

```
bcf    INTCON,GIE    ; interdire les interruptions
clrf   TMR1L         ; 0 dans TMR1L
movlw  VALH          ; charger valeur haute
movwf  TMR1H         ; dans registre
movlw  VALL          ; charger valeur basse
movwf  TMR1L         ; dans registre
bsf    INTCON,GIE    ; réautoriser les interruptions
```

Vous voyez que ce n'est pas compliqué, à condition de prendre garde aux pièges qui vous sont tendus.

Il me reste à vous signaler qu'une écriture dans un des registres TMR1L ou TMR1H provoque l'effacement des éléments déjà comptabilisés dans le prédiviseur. Une écriture dans un de ces registres, si vous utilisez un prédiviseur différent de 1, provoquera donc une perte d'informations.

Attention, ne confondez pas « valeur du prédiviseur » et « contenu du prédiviseur ». La valeur est celle que vous avez placée via T1CKPS1 et T1CKPS0. Ces valeurs ne sont pas modifiées. Le contenu est le nombre d'événements déjà comptés par le prédiviseur. C'est donc ce contenu qui est perdu lors d'une écriture dans TMR1L ou TMR1H.

15.12 Exercice pratique

A la lecture de tout ce qui précède, vous devez être conscient qu'il m'est très difficile de vous proposer des exercices pour tous les cas de figure possibles. Je vais donc me limiter à un seul exemple qui mettra en pratique certains des points que nous avons abordés.

Puisque nous avons déjà pas mal parlé du mode timer lorsque nous avons étudié le timer 0 dans la première partie, je vais vous proposer un exercice qui utilise le timer 1 en mode compteur.

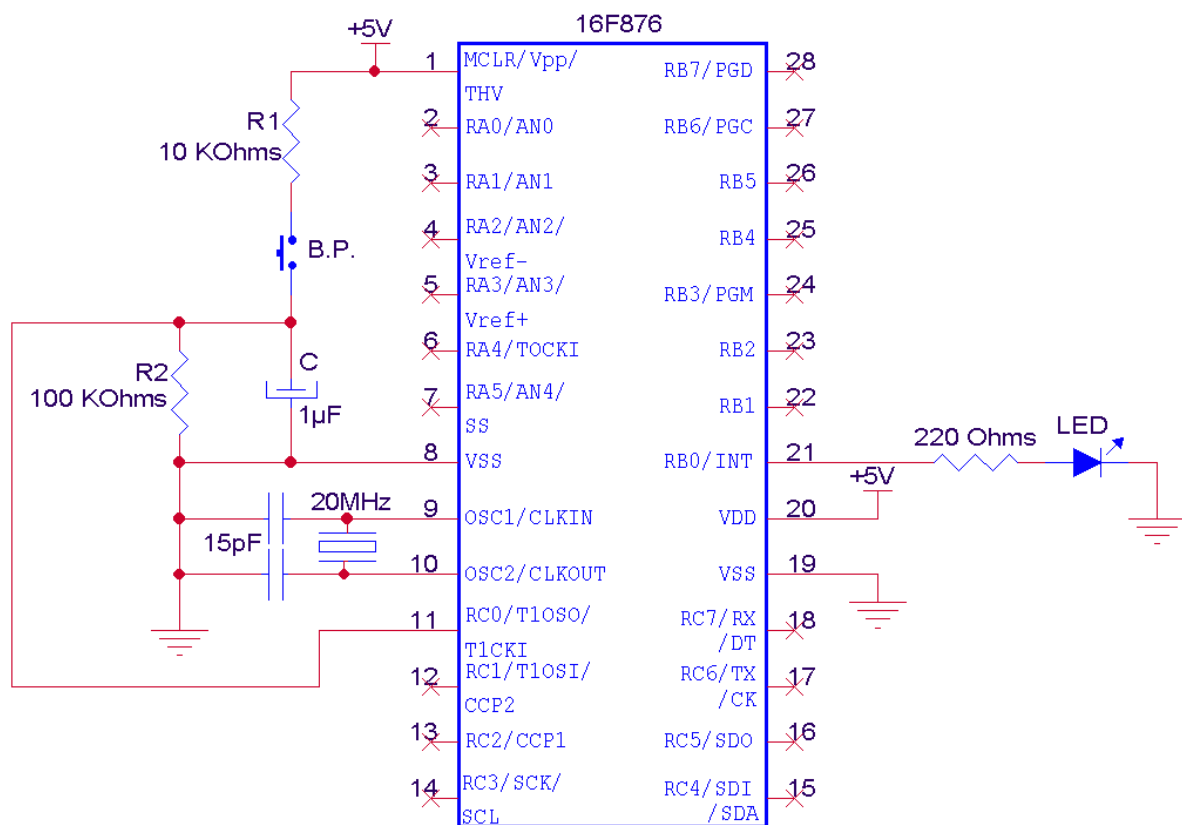
Nous allons donc construire un programme qui inverse l'état d'une LED chaque fois que 10 impulsions sont reçues sur la pin T1CKI. Ces impulsions seront créées par des pressions sur un bouton-poussoir.

Si vous vous souvenez de la première partie, j'avais parlé que les interrupteurs sont soumis à des rebonds. Il nous faudra donc, pour que notre montage fonctionne éliminer ces rebonds.

J'utiliserai pour ce faire la charge d'un petit condensateur au travers d'une résistance. Ceci permettra de ralentir (intégrer) la montée du niveau en tension, de façon à ce que le passage à l'état « 1 » intervienne après la fin des rebonds de l'interrupteur.

Afin de permettre l'utilisation de tous les types d'interrupteurs, j'ai choisi un temps de retard de 20ms.

Voici le schéma que nous allons utiliser :



15.12.1 Un peu de maths

Je pourrais vous donner les valeurs que j'ai choisies en vous donnant une formule approximative, pourtant largement suffisante. Seulement, j'ai remarqué qu'il y a des méticuleux parmi mes lecteurs (je n'ai pas dit « maniaques »). Si je ne donne pas plus de précision, je vais donc m'attirer de nouveau de nombreux courriers. Je prends donc les devants...

... Et puis, il faut bien que je montre de temps en temps que je sais calculer, et, de plus, ça va m'attirer la sympathie des professeurs de mathématiques (en espérant que je ne fasse pas d'erreur).

Plus sérieusement, je profite de cet exemple pour vous montrer comment calculer des montages particuliers. Ceux que ça n'intéresse pas n'ont qu'à exécuter le pseudo-code suivant :

Début

Si je n'aime pas les maths

Alors goto chapitre 15.12.2

Sinon

On poursuit :

Redevenons sérieux (ça change) :

Voyons donc les données :

- Le temps de charge : 20 ms
- La tension appliquée : 5V
- La tension détectée comme niveau haut par le PIC : 0,7 VDD (datasheet)

Les inconnues :

- R1
- R2
- C

Et la formule de charge d'un condensateur sous une tension constante, et via une résistance série :

$$U_c = U_1 (1 - e^{-t/rc})$$

Avec U_c = tension aux bornes du condensateur après le temps « t », U_1 = tension de charge, t = temps écoulé depuis le début de la charge, r = résistance de charge, et c = valeur du condensateur.

Donc, une équation avec 3 inconnues (R1,R2,C), il va falloir avoir de l'imagination (comme d'habitude en électronique).

Tout d'abord, si on regarde le montage, on constate que si on ferme l'interrupteur, en fin de charge du condensateur, la tension finale est déterminée par le pont diviseur formé par R1 et R2. Donc, il nous faut choisir R2 fortement supérieur à R1, sous peine de ne pouvoir jamais

atteindre notre tension de niveau haut égale à 0,7 Vdd. Nous prendrons arbitrairement $R2 = 10 \cdot R1$. Vous voyez en effet que si vous choisissez $R1 = R2$, par exemple, votre tension finale ne pourra dépasser $Vdd/2$.

Ensuite, il ne faut pas que le courant consommé par la pin T1CKI soit trop important par rapport au courant de charge, sous peine de ne pas arriver à charger le condensateur. Il faut aussi que l'impédance d'entrée de T1CKI soit assez supérieur à $R2$ pour ne pas fausser nos calculs.

Le datasheet (15.3) nous donne un courant d'entrée de $T1CKI = 5\mu A$.

Forts de tout ceci, nous déduisons la résistance d'entrée de notre T1CKI :

$$R = U/I$$

$$R = 5V / (5 \cdot 10^{-6}A) = 10^6 \text{ Ohms} = 1 \text{ MOhms}$$

Nous choisirons donc, pour éviter les interférences entre courant absorbé par T1CKI et charge du condensateur, $R2 = R_{tcki} / 10$

Donc : $R2 = 100 \text{ KOhms}$

D'où : $R1 = R2 / 10$, donc $R1 = 10 \text{ Kohms}$

Ne reste donc plus qu'une inconnue, C , qu'il faut tirer de l'équation précédente. Qui s'y colle ? Ben oui, les maths, tout compte fait, ça ne sert pas qu'à l'école. Si ça vous tente de faire comme moi, prenez un papier et calculez ce condensateur. Sinon, lisez la suite :

Bon, essayons de voir notre formule, il nous faut extraire C qui est dénominateur d'un exposant de « e ». Pas vraiment marrant. Posons :

$$-t/rc = x$$

Ceci rend la lecture plus claire, car nous avons maintenant :

$$U_c = U_1 \cdot (1 - e^x)$$

Nous ne connaissons pas « x », mais nous avons besoin, pour le trouver de connaître U_1 , qui est la tension réelle de charge.

5V me direz-vous ? Pas du tout ! La tension de charge est déterminée par le pont diviseur $R1/R2$. Encore des souvenirs de l'école allez-vous penser. C'est très simple pourtant, cette fois il suffit d'appliquer la loi d'Ohm (j'ai horreur de retenir des formules toutes faites, je préfère les retrouver par raisonnement).

En fin de charge du condensateur, et étant donné que nous avons choisi les résistances pour que le courant absorbé par T1CKI soit négligeable, nous pouvons dire que le courant circulant dans $R1$ (I_{R1}) est le même que celui circulant dans $R2$ (I_{R2}), et, évidemment le même que celui qui circule dans les deux (I_{R1R2}).

Donc, comme $I = U/R$,

$$I_{R2} = V_{dd} / (R1+R2)$$

Donc, la tension aux bornes de R2, qui est notre tension de charge finale, est égale à $I_{R2} * R2$, donc :

$$U_{R2} = U_1 = (V_{dd} / (R1+R2)) * R2$$

$$U_1 = (5V * 10^5 \text{ Ohms}) / (10^5 \text{ Ohms} + 10^4 \text{ Ohms}).$$

$$U_1 = (5V * 10^5) / (11*10^4)$$

$$U_1 = 4,54V$$

Le passage au niveau « 1 » se fait à $0,7V_{dd}$, soit $0,7 * 5V = 3,5V$. Donc, notre tension finale de 4,54V nous permet bien de faire passer T1CKI au niveau « 1 ».

Reprenons notre équation :

$$U_c = U_1 * (1-e^x)$$

Donc :

$$1 - e^x = U_c/U_1$$

D'où

$$e^x = 1 - (U_c/U_1)$$

Mais ce foutu « x » est toujours en exposant, il faudrait le faire « descendre ». En grattant dans vos souvenirs, vous vous souviendrez peut-être que le logarithme népérien d'un nombre à la puissance « n » est égal à « n ». Autrement dit : $\ln(e^x) = x$.

Pour vous en convaincre, ce n'est qu'un cas particulier des logarithmes. Un logarithme en base x de la base à la puissance y est égal à y. Simple exemple en base 10 :

$$\text{Log } 10^5 = 5$$

Le logarithme népérien n'étant qu'un logarithme en base « e », vous comprenez maintenant l'égalité précédente.

Il suffit donc, pour respecter l'égalité, de prendre le logarithme népérien des 2 côtés de l'identité. Ceci nous donne :

$$\ln(e^x) = \ln(1 - (U_c/U_1))$$

Ou encore :

$$x = \ln(1 - (U_c/U_1))$$

Uc représente la tension aux bornes du condensateur après le temps que nous avons défini (20ms). Or, ce temps est le temps du passage au niveau haut de T1CKI. Ce passage s'effectuant lorsque la tension est égale à 0,7 Vdd, nous avons donc que $U_c = 0,7V_{dd}$, soit $0,7 \times 5V$ (la tension d'alimentation du PIC) = 3,5V.

Donc, nous pouvons tirer « x » :

$$X = \ln(1 - (3,5V / 4,54V)) = -1,473$$

Ne reste plus qu'à remplacer de nouveau « x » par ce qu'il représente :

$$(-t/rc) = -1,473$$

donc,

$$t = 1,473 * r * c$$

Autrement dit (et écrit) :

$$C = t / (1,473 * r)$$

$$C = 20 * 10^{-3} F / (1,473 * 10^4)$$

On peut en effet estimer que la résistance de charge vaut R1.

$$C = 20 * 10^{-7} F / 1,473 = 2 * 10^{-6} F / 1,473 = 1,3 * 10^{-6} F$$

Donc,

$$C = 1,3 \mu F.$$

Nous prendrons donc un condensateur de 1 μF .

Si, pour vérifier, vous remplacez C par sa valeur dans la formule initiale, :

$U_c = U_1 * (1 - e^{-t/rc})$, vous trouverez $U_c = 3,56V$, ce qui nous rend bien les 3,5V de passage du niveau « 0 » vers le niveau « 1 » de notre pin T1CKI. C.Q.F.D.

Ca fait plaisir de voir qu'on n'est pas encore tout à fait rouillé, pas vrai ?

15.12.2 Le programme

Revenons à nos moutons, je veux dire à nos PICs. Nous allons construire le pseudo-code de notre programme :

Initialiser

Initialiser le timer 1 et le charger avec une valeur de « -10 »

Aller au programme principal

Interruption timer 1
 Inverser LED
 Recharger timer 1 avec « -10 »
 Fin d'interruption

Programme principal
 Rien faire du tout

Copiez comme d'habitude votre fichier maquette, et renommez la copie « cmpt1.asm ». Créez un nouveau projet, et créez l'en-tête du programme.

```

;*****
; Exercice d'utilisation du timer 1 en mode compteur. *
; On inverse une LED pour chaque série de 10 impulsions reçues sur T1CKI *
; *
;*****
; *
; NOM: Cmpt1 *
; Date: 07/05/2002 *
; Version: 1.0 *
; Circuit: platine d'expérimentation *
; Auteur: Bigonoff *
; *
;*****
; *
; Fichier requis: P16F876.inc *
; *
; *
; *
;*****
; *
; Notes: Entrée des impulsions sur T1CKI (RC0). *
; Impulsions générées par un bouton-poussoir équipé d'un système *
; anti-rebond par réseau RC. *
; LED de sortie connectée sur RB0 *
; *
;*****
;*****

LIST p=16F876 ; Définition de processeur
#include <p16F876.inc> ; fichier include

__CONFIG _CP_OFF & _DEBUG_OFF & _WRT_ENABLE_OFF & _CPD_OFF & _LVP_OFF &
_BODEN_OFF & _PWRTE_ON & _WDT_OFF & _HS_OSC

;_CP_OFF Pas de protection
;_DEBUG_OFF RB6 et RB7 en utilisation normale
;_WRT_ENABLE_OFF Le programme ne peut pas écrire dans la flash
;_CPD_OFF Mémoire EEprom déprotégée
;_LVP_OFF RB3 en utilisation normale
;_BODEN_OFF Reset tension hors service
;_PWRTE_ON Démarrage temporisé
;_WDT_OFF Watchdog hors service
;_HS_OSC Oscillateur haute vitesse (20Mhz)
  
```

Ensuite, les assignations système, dont on ne conserve que celles utiles. A noter que, vu qu'on n'a qu'un bit à positionner pour chacune, et que le niveau est connu au moment de la mise sous tension, on aurait pu se permettre de n'utiliser que le positionnement du bit en

question (bsf ou bcf) au niveau de la routine d'initialisation. Cependant, pour ne rien omettre, conservons ces assignations.

```
;*****
;                                     ASSIGNATIONS SYSTEME                                *
;*****

; REGISTRE OPTION_REG (configuration)
; -----
OPTIONVAL EQUB'10000000' ; Résistance rappel +5V hors service

; REGISTRE INTCON (contrôle interruptions standard)
; -----
INTCONVAL EQUB'01000000' ; autorisation générale périphériques

; REGISTRE PIE1 (contrôle interruptions périphériques)
; -----
PIE1VAL      EQUB'00000001' ; interrupt débordement tmr1
```

Ensuite, nous avons nos constantes. Ici, nous souhaitons que le timer 1 déborde au bout de 10 pressions sur le bouton-poussoir. Nous devons donc recharger le timer avec une valeur telle que TMR1H déborde après 10 événements. Donc, comme je vous l'ai expliqué :

```
;*****
;                                     ASSIGNATIONS PROGRAMME                            *
;*****

RELOADEQU    0x10000 - D'10'      ; valeur de recharge de TMR1
```

Concernant les macros, ne conservez que la macro de passage en banque 0

```
;*****
;                                     MACRO                                           *
;*****

BANK0 macro          ; passer en banque0
    bcf STATUS,RP0
    bcf STATUS,RP1
endm
```

Nous n'avons, à première vue, pas besoin de variable, vu que c'est notre timer 1 qui va compter lui-même les impulsions reçues. Profitons-en également pour supprimer les variables de sauvegarde dont nous n'aurons pas besoin. Notre programme principal ne fera rien du tout, il n'y a donc rien à sauvegarder lors d'une interruption.

```
;*****
;                                     VARIABLES BANQUE 0                             *
;*****
CBLOCK 0x20          ; Début de la zone (0x20 à 0x6F)
ENDC

;*****
;                                     VARIABLES ZONE COMMUNE                         *
;*****
CBLOCK 0x70          ; Début de la zone (0x70 à 0x7F)
ENDC
```

```
;*****
;
;                               DEMARRAGE SUR RESET                               *
;*****

    org 0x000      ; Adresse de départ après reset
    goto  init     ; Initialiser
```

Voyons notre routine d'interruption, très simple :

- Il n'y a qu'une seule interruption, donc, inutile de tester de quelle interruption il s'agit.
- Il n'y a rien à sauvegarder, donc rien à restaurer
- La routine se contente d'inverser la LED et de recharger le timer (compteur)
- Il ne faut pas oublier d'effacer le flag de l'interruption timer 1 (TMR1IF)

Nous pouvons donc écrire notre routine d'interruption, réduite à sa plus simple expression :

```
;*****
;
;                               ROUTINE INTERRUPTION TIMER 1                               *
;*****

    ORG 0x04

    movlw B'00000001'      ; pour bit 0
    xorwf PORTB,f          ; inverser LED
    clrf TMR1L             ; car écriture sans stopper compteur
    movlw HIGH RELOAD      ; octet fort valeur de recharge
    movwf TMR1H            ; dans timer poids fort
    movlw LOW RELOAD       ; octet faible valeur de recharge
    movwf TMR1L            ; dans timer poids faible
    bcf PIR1,TMR1IF        ; effacer flag interrupt
    retfie                 ; return from interrupt
```

Reste maintenant, dans notre routine d'initialisation, à initialiser notre timer 1 en mode compteur (nous choisirons asynchrone, mais cela n'a aucune importance ici), à le précharger avec le valeur « RELOAD », de façon à ce que le premier allumage de la LED s'effectue après 10 pressions sur notre bouton-poussoir, et à initialiser notre LED en sortie de RB0.

```
;*****
;
;                               INITIALISATIONS                               *
;*****

init

    ; initialisation PORTS (banque 0 et 1)
    ; -----
    BANK0                ; sélectionner banque0
    clrf PORTB           ; sorties PORTB à 0
    bsf STATUS,RP0       ; passer en banque1
    bcf TRISB,0           ; RB0 en sortie (LED)

    ; Registre d'options (banque 1)
    ; -----
    movlw OPTIONVAL       ; charger masque
    movwf OPTION_REG      ; initialiser registre option

    ; registres interruptions (banque 1)
    ; -----
    movlw INTCONVAL       ; charger valeur registre interruption
```

```

movwf INTCON          ; initialiser interruptions
movlw PIE1VAL         ; Initialiser registre
movwf PIE1            ; interruptions périphériques 1

                ; initialiser timer 1
                ; -----
bcf    STATUS,RP0     ; passer banque 0
movlw  LOW RELOAD     ; octet faible de recharge timer 1
movwf  TMR1L          ; dans registre poids faible
movlw  HIGH RELOAD    ; octet fort de recharge timer 1
movwf  TMR1H          ; dans registre poids fort
movlw  B'000000111'   ; timer en service en mode compteur asynchrone
movwf  T1CON          ; dans registre de contrôle timer 1
bsf    INTCON,GIE     ; valider interruptions

```

Reste notre programme principal, qui ne fait strictement rien :

```

;*****
;                               PROGRAMME PRINCIPAL                               *
;*****
start
    goto start          ; boucler
END                    ; directive fin de programme

```

Programmez votre 16F876, placez la sur votre circuit, et lancez l'alimentation. Pressez le bouton calmement une trentaine de fois. Votre LED doit s'inverser à chaque multiple de 10 pressions.

Si, à ce niveau, vous obtenez un comportement erratique (la LED s'allume n'importe quand, « frétille » etc.), c'est que vous avez travaillé avec des fils volants. Or, la résistance de rappel à la masse de T1CKI (R2) est de 100 Kohms, ce qui est une grosse valeur. Il faut savoir que lorsque vous travaillez avec des valeurs supérieures à 10 Kohms, les interférences (parasites) sont de plus en plus susceptibles d'affecter votre montage.

Dans ce cas, le remède est fort simple : vous remplacez R2 par une résistance de 10Kohms. Cependant, alors, pour respecter tous nos calculs, il vous faudra alors remplacer R1 par une résistance de 1Kohms, et C par un condensateur de 10µF. Vous voyez qu'il y a toujours plusieurs paramètres électroniques à prendre en compte pour la réalisation d'un montage. En règle générale, utilisez des résistances d'entrée inférieures ou égales à 10 Kohms.

Bon, à ce stade, tout le monde est à égalité, avec un montage opérationnel. Coupez l'alimentation, attendez 5 secondes, et relancez l'alimentation.

Pressez lentement le bouton-poussoir (1 pression par seconde), et comptez les impulsions avant le premier allumage de la LED. Vous vous attendiez à 10 impulsions, mais vous devez en trouver 11.

Que voilà un curieux phénomène, vous ne trouvez pas ?

En fait, l'explication est donnée par le fonctionnement interne du timer 1 du PIC.

La première impulsion n'est prise en compte que si le signal d'entrée a subi au moins un flanc descendant. Et oui, vous aviez déjà oublié ?

Ceux qui s'en sont souvenus sont très forts et méritent une mention « très bien ».

Pour résumer, à la mise sous tension de notre montage, le niveau sur T1CKI est bas. Lorsque vous pressez la première fois sur le bouton, vous avez un flanc montant. Mais ce flanc montant n'ayant pas été précédé d'un flanc descendant, ce flanc n'est tout simplement pas pris en compte.

Vous relâchez ensuite le bouton, ce qui fait passer votre niveau d'entrée de « 1 » à « 0 ». Ceci constitue votre premier flanc descendant. Donc, le comptage s'effectuera à partir de la prochaine pression du bouton-poussoir.

Vous pouvez vérifier ceci en constatant que si le premier allumage de la LED s'effectue après 11 pressions, tous les autres allumages et extinctions s'effectueront après 10 impulsions.

Vous pourriez corriger le programme en initialisant la valeur du timer1 dans la routine de d'initialisation à « -9 » au lieu de « -10 ». La valeur « -10 » devant demeurer dans la routine d'interruption. Si vous voulez vérifier, il suffit donc d'ajouter « 1 » dans TMR1L (pour les réticents en maths, pour passer de « -10 » à « -9 », il faut ajouter « 1 » et non soustraire ».

Votre routine d'initialisation du timer devient :

```
        ; initialiser timer 1
        ; -----
bcf     STATUS,RP0      ; passer banque 0
movlw   LOW RELOAD+1    ; octet faible de recharge timer 1
movwf   TMR1L           ; dans registre poids faible
movlw   HIGH RELOAD     ; octet fort de recharge timer 1
movwf   TMR1H           ; dans registre poids fort
movlw   B'000000111'    ; timer en service en mode compteur asynchrone
movwf   T1CON           ; dans registre de contrôle timer 1
bsf     INTCON,GIE      ; valider interruptions
```

15.13 Errata : Fonctionnement non conforme

Je termine sur une remarque importante qui concerne plusieurs versions de 16F87x., quand le timer 1 est configuré en mode « compteur », que ce soit synchrone ou asynchrone :

La lecture du registre TMR1L peut empêcher TMR1H d'être incrémenté durant le temps de la lecture. Inversement, la lecture de TMR1H peut empêcher TMR1L d'être incrémenté durant le temps de la lecture.

Ceci peut être particulièrement gênant, si le passage de TMR1L de 0xFF à 0x00 ne provoque pas l'incrément de TMR1H, donnant de ce fait une erreur non négligeable.

Microchip indique que ce point sera prochainement corrigé, mais cela fait plusieurs versions de suite que cette correction n'est pas effective.

A vous, dans le cas où vous utilisez ces possibilités, soit de gérer ce phénomène, soit de vous renseigner chez Microchip pour savoir si votre propre version de PIC intègre toujours ce bug.

La solution de Microchip pour contourner le problème est pour le moins radicale : il recommande, si vous utilisez le timer 1 avec le bit TMR1CS positionné (mode compteur) et que vous devez lire la valeur 16 bits du compteur, d'utiliser tout simplement un autre timer ou une autre méthode pour votre application.

Précisons, pour ceux qui n'auraient pas bien compris, que ça ne concerne que la lecture ou l'écriture des registres TMR1L/TMR1H sans arrêter le timer.

Les solutions que j'ai proposées pour éviter les erreurs ne fonctionneront donc que sur les versions debuggées des PICs.

J'ai cependant donné ces solutions, car ce sont des grands classiques sur toutes les familles de microcontrôleurs utilisant des compteurs sur 16 bits. Connaître ces techniques vous sera toujours utile. De plus, lorsque vous passerez à des PICs plus performants, il faudra espérer que Microchip aura résolu ce problème.

Notes : ...

Notes : ...

16. Le debuggage « pin-stimulus »

Et oui, en cas de problème, la première question qui vient à l'esprit est : «comment puis-je simuler un programme qui fait intervenir des modifications de niveaux sur les pins » ?

J'introduis ce chapitre ici, car vous vous posez peut-être la question suite à vos éventuels déboires concernant l'exercice précédent.

En fait, MPLAB dispose de la possibilité de créer un fichier qui contient les événements à envoyer sur une ou plusieurs pins, en fonction du nombre de cycles écoulés depuis le début du programme.

Prenez votre projet « **cmpt1** », que nous allons faire fonctionner en mode simulation.

- Allez dans le menu : « **file -> New** ». Une fenêtre s'ouvre alors.
- Dans cette fenêtre, vous commencez par introduire le mot « **CYCLE** » qui sert à MPLAB (anciennes versions) à déterminer qu'il s'agit d'un fichier d'événements.
- Tapez une tabulation, puis entrez le numéro de la pin à simuler. Pour notre part, il s'agit de T1CKI. Mais comme MPLAB ne reconnaît pas ce nom, nous utiliserons son autre nom : « RC0 ».
- Vous pouvez ensuite taper d'autres noms séparés par des tabulations, chaque nom correspondant à une pin à simuler.
- Tapez « return »
- Sous la colonne « CYCLE », vous tapez le numéro du cycle auquel se rapporte la modification. Il s'agit ici de cycles d'instructions. Pour rappel ,chaque instruction utilise un cycle, sauf les sauts qui en nécessitent 2.
- Ensuite, sous la colonne « RC0 », donc après avoir tapé une nouvelle tabulation, entrez l'état que vous désirez pour la pin en question.
- Procédez de même pour les autres pins, et vous pouvez terminer par un commentaire séparé par un « ; »

Remarque

Pour savoir quels sont les noms de pins que vous pouvez utiliser, allez dans « **Debug>Simulator Stimulus>Asynchronous** ». Une fois la fenêtre ouverte, allez sur « Stim 1 (P) » et cliquez avec le bouton de droite. Choisissez « assign pin... » et vous avez la liste des noms de pins autorisés.

Observez que cette fenêtre vous donne accès à MCLR, donc vous permet de simuler des opérations de « reset ».

Voici le fichier que nous allons créer pour la circonstance :

```
CYCLE  RC0
00      0      ; au début, RC0 = 0
30      1      ; 30 cycles après le début, mettre RC0 à « 1 »
40      0      ; puis, tous les 10 cycles, on inverse RC0
50      1
60      0
70      1
80      0
90      1
100     0
110     1
120     0
130     1
140     0
150     1
160     0
170     1
180     0
190     1
200     0
210     1
220     0
230     1
240     0
250     1
260     0
270     1
280     0
```

A titre d'information, si vous aviez voulu utiliser 2 pins (RC0 et RC1) , voici ce que vous auriez pu créer :

```
CYCLE  RC0    RC1
00      0      0      ; au temps 0, RC0 et RC1 sont à « 0 »
30      1      0      ; après 30 cycles, RC0 = 1 et RC1 = 0
40      0      1      ; 10 cycles plus loin, RC0 = 0 et RC1 = 1
```

Notez que la valeur des cycles représente le nombre de cycles écoulés depuis le début du programme.

Maintenant, sauvez votre fichier (file -> save as...) en entrant un nom terminé par l'extension « .sti » pour « STImulus ». Ne tenez donc pas compte des suffixes indiqués par défaut.

Vous sauvez alors votre fichier sous « cmpt1.sti ». Vérifiez impérativement que la case « unix format » ne soit pas cochée.

Nous allons ensuite indiquer à MPLAB qu'il doit utiliser ce fichier :

- Allez dans le menu : « debug -> simulator stimulus »
- Choisissez « pin-stimulus -> enable... »

- Un requester s'ouvre alors, vous demandant de choisir votre fichier. Choisissez « **cmpt1.sti** ».

A ce stade, allez dans « **windows -> Stopwatch...** »

Une fenêtre s'ouvre alors, qui vous indique le nombre de cycles que MPLAB estime écoulé depuis le début de votre simulation. Si la fenêtre « cycles » n'indique pas « 0 », alors pressez « **Zero** ».

Vous pouvez maintenant lancer la simulation de façon classique. Pressez F6, puis F7 pour chaque pas.

Remarquez qu'au cycle 30, RC0 passe à 1 et que TMR1L est incrémenté. Continuez vos actions sur F7 jusqu'à ce que une interruption soit générée.

La dernière action de votre fichier « sti » se termine au cycle 280. Une fois arrivé à ce cycle, il vous suffit de presser « zero » dans votre fenêtre « stopwatch » pour que la lecture de votre fichier « sti » reprenne au début, sans influencer le déroulement de votre programme.

Vous constaterez également que MPLAB incrémente votre registre « TMR1L » au premier flanc montant de RC0. Le programme ne présuppose pas des états précédents de RC0. Il ne peut donc savoir s'il s'agit ou non d'un premier flanc montant.

Ceci doit attirer votre attention sur les limites des simulations. Une simulation reste une simulation, et ne peut donc prendre toutes les contraintes réelles en compte.

Un autre exemple est que, en mode simulation, toutes les variables en zone RAM sont initialisées à 0, alors que dans la réalité, leur contenu est aléatoire. Idem concernant le contenu des timers.

Nous verrons dans le livre suivant une méthode de debuggage beaucoup plus puissante.

Notes :

17. Le timer 2

Dans ce chapitre, nous allons maintenant étudier le dernier des trois timers de notre PIC. Celui-ci dispose, comme vous pouvez déjà vous en douter, de caractéristiques différentes des 2 autres.

Cette approche de Microchip permet à l'utilisateur d'avoir un panaché des modes d'utilisations possibles, tout en conservant, pour chaque timer, une facilité d'utilisation et une complexité, donc un coût, abordables.

Votre démarche, lorsque vous choisirez un timer, sera donc fonction de l'utilisation envisagée. J'y reviendrai.

17.1 Caractéristiques du timer 2

Le timer 2 est un compteur sur 8 bits, donc nous ne rencontrerons pas les difficultés inhérentes à la lecture et à l'écriture de ses registres.

Le timer 2, comme les précédents, possède un prédiviseur. Celui-ci peut être paramétré avec une des 3 valeurs suivantes : 1, 4, ou 16. Nous sommes donc pauvres à ce niveau.

Cependant, le timer 2 dispose également d'un postdiviseur, qui effectue une seconde division après l'unité de comparaison, que nous allons voir. Ce postdiviseur peut prendre n'importe quelle valeur comprise entre 1 et 16, ce qui donne un grand choix possible à ce niveau.

La valeur du diviseur total, vue par l'utilisateur, est bien entendu obtenue en multipliant la valeur du prédiviseur par celle du postdiviseur.

Moyennant ceci, il est possible, avec le timer 2, d'obtenir les valeurs de diviseur suivantes :

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60, 64, 80, 96, 112, 128, 144, 160, 176, 192, 208, 224, 240, 256

Vous voyez qu'avec ce timer, vous disposez d'un large éventail de diviseurs effectifs.

Le timer 2 incrémente pour sa part le registre TMR2, registre unique puisque comptage sur 8 bits.

Les valeurs de division minimale et maximale sont donc identiques à celles du timer 0, qui disposait également d'un comptage sur 8 bits, avec prédiviseur de 1 à 256.

Le registre TMR2 est remis automatiquement à 0 lors d'un reset, contrairement au timer 1, pour lequel vous deviez vous en charger. Ceci peut être un avantage ou un inconvénient, suivant le type de réaction que vous attendez du timer. C'est clair qu'un reset inopiné et non désiré du pic remettra dans ce cas votre timer 2 dans son état initial.

Il faut également tenir compte que ce timer ne dispose d'aucune entrée extérieure via une pin du PIC. Il ne peut donc fonctionner qu'en mode « timer » pur.

17.2 Le timer 2 et les interruptions

Le timer 2 fonctionne, à ce niveau, comme le timer1. Le flag d'interruption se nomme **TMR2IF**, en toute logique, tandis que le bit d'autorisation s'appelle **TMR2IE**.

La principale différence provient de l'événement qui cause le positionnement de **TMR2IF**, donc qui cause l'interruption. Je vais en parler un peu plus loin.

Tout comme pour le timer 1, il s'agit d'une interruption périphérique, donc, la procédure pour autoriser les interruptions du timer 2 se fera en 3 étapes :

- Autorisation des interruptions périphériques via le bit **PEIE** du registre **INTCON**
- Autorisation de l'interruption timer 2 via **TMR2IE** du registre **PIE1**
- Autorisation générale des interruptions via le bit **GIE** du registre **INTCON**

Je vous renvoie donc aux chapitres précédents pour des renseignements plus précis.

17.2 Le timer 2 et les registres PR2 et T2CON

Le principe de fonctionnement des 2 précédents timers était le suivant :

- On incrémente le contenu du TMR (sur 1 ou 2 octets) suivant l'événement choisi et la valeur du prédiviseur.
- Une fois que le timer « déborde », ceci déclenche le positionnement du flag associé

Le principe du timer 2 est différent, dans le sens que l'événement détecté n'est pas le débordement « ordinaire » du timer (c'est-à-dire le passage de 0xFF à 0x00), mais le débordement par rapport à une valeur prédéfinie.

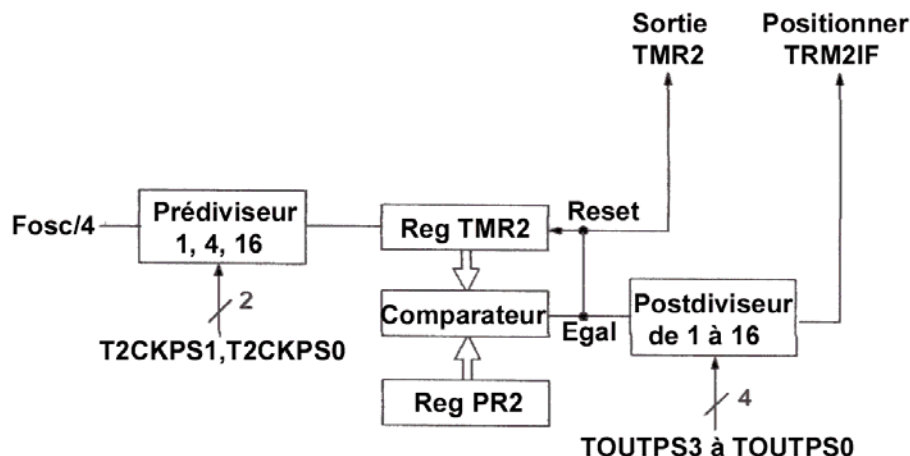
Cette valeur étant mémorisée dans le registre PR2 (banque 1). Nous pouvons donc avoir, par exemple, débordement de 0x56 à 0x00, en plaçant la valeur 0x56 comme valeur maximale dans le registre PR2.

On peut donc dire que le fonctionnement du timer est le suivant :

- On incrémente le contenu du prédiviseur à chaque cycle d'instruction
- Chaque fois que ce contenu correspond à un multiple de la valeur du prédiviseur, on incrémente TMR2 (contenu du timer 2)
- Chaque fois que **le contenu de TMR2 dépasse le contenu de PR2**, on remet TMR2 à 0, et on incrémente le contenu du postdiviseur.

- Chaque fois que le contenu du postdiviseur correspond à un multiple de la valeur du postdiviseur, on positionne le flag TMR2IF.

Pour clarifier la compréhension, je vous donne le schéma-bloc du timer 2.



Vous constatez qu'il n'y a pour ce timer qu'une seule source de comptage, à savoir l'horloge principale du PIC divisée par 4, autrement dit le compteur d'instructions. Nous sommes donc bien en présence d'un timer « pur ».

Une prédivision est paramétrée par **T2CKPS0** et **T2CKPS1**. La sortie du prédiviseur incrémente le registre **TMR2**. Cette valeur est comparée avec la valeur contenue dans **PR2**.

Chaque fois que les contenus de TMR2 dépasse celle de PR2, la sortie du comparateur incrémente la valeur contenue dans le postdiviseur. Cette sortie effectue également un reset de TMR2, qui redémarre donc à 0.

En fait ce schéma est trompeur, bien que d'origine Microchip, car il risque d'induire en erreur. En effet, la sortie du comparateur ne donne pas une impulsion lorsque les 2 registres sont égaux, mais lorsque le registre TMR2 dépasse la valeur de PR2.

Donc, le nombre de cycles réellement comptés est, abstraction faite des diviseurs, la valeur de PR2 incrémentée de 1.

En effet, tout comme pour le timer0, vous aviez une impulsion à chaque débordement de 0xFF vers 0x00, pour le cas du timer2, vous aurez « débordement » de la valeur de PR2 vers 0x00. Le timer 0 donnait bien, pour une valeur maximale de 0xFF, un nombre d'impulsions comptées de 0x100 (D'256'), donc valeur maximale + 1. Il s'agit donc ici du même phénomène et du même calcul.

Chaque fois que le contenu du postdiviseur est égal à un multiple de la valeur de ce celui-ci, paramétrée par **TOUTPS0** à **TOUTPS3**, le flag **TMR2IF** est forcé à 1, et une interruption est éventuellement générée.

Ne vous préoccupez pas de la flèche « sortie de TMR2 », le timer2 est utilisé en interne pour d'autres fonctions que nous étudierons plus tard.

Une écriture dans le registre TMR2 efface le contenu du prédiviseur et du postdiviseur.

Pour rappel, ne pas confondre contenu (nombre d'événements comptés) et valeur (déterminée par les bits de configuration).

Forts de tout ceci, vous avez maintenant compris que la spécificité du timer 2, et donc son principal avantage, est qu'il permet de configurer le « débordement » sur n'importe quelle valeur de TMR2, associé à un large éventail de valeurs de diviseur.

Inutile donc d'attendre le passage de 0xFF à 0x00, quoique cela reste possible, simplement en plaçant 0xFF dans PR2.

Cet avantage, combiné à la grande flexibilité de l'ensemble prédiviseur/postdiviseur, permet d'obtenir très facilement des durées d'interruption précises sans complications logicielles.

Voici à présent le contenu du registre T2CON, qui permet de paramétrer prédiviseur et postdiviseur, ainsi que d'autoriser ou non le fonctionnement du timer2.

T2CON (en banque 0)

- b7 : non utilisé, laisser à 0
- b6 : TOUTPS3 : Timer2 Output PostScale bit 3
- b5 : TOUTPS2 : Timer2 Output PostScale bit 2
- b4 : TOUTPS1 : Timer2 Output PostScale bit 1
- b3 : TOUTPS0 : Timer2 Output PostScale bit 0
- b2 : TMR2ON : Timer2 ON
- b1 : T2CKPS1 : Timer2 Clock PreScale bit 1
- b0 : T2CKPS0 : Timer2 Clock PreScale bit 0

Vous constatez que les bits TOUTPSx permettent de configurer la valeur du postdiviseur. Il y a 16 valeurs possibles (0 à 15). Comme une valeur de diviseur de 0 n'a aucun sens, le nombre formé par les 4 bits de TOUTPSx est incrémenté de 1 pour obtenir la valeur effective du postdiviseur.

Voici donc les valeurs utilisables :

b6	b5	b4	b3	Postdiviseur
0	0	0	0	1
0	0	0	1	2
0	0	1	0	3
0	0	1	1	4
0	1	0	0	5
0	1	0	1	6
0	1	1	0	7
0	1	1	1	8
1	0	0	0	9
1	0	0	1	10
1	0	1	0	11
1	0	1	1	12
1	1	0	0	13
1	1	0	1	14
1	1	1	0	15
1	1	1	1	16

Quant au prédiviseur, on n'a le choix qu'entre 3 valeurs :

b1	b0	Prédiviseur
0	0	1
0	1	4
1	0	16
1	1	16

Il me reste à vous donner la formule de la durée séparant 2 positionnements consécutifs du flag TMR2IF. Vous pourriez retrouver cette formule vous-même, en suivant les explications précédentes :

Durée totale = temps d'une instruction * prédiviseur * postdiviseur * (PR2 +1)

La valeur maximale est donc bien, comme pour le timer 0 de $16*16*256 = 65536$.

17.3 Utilisation pratique de notre timer 2

Supposons que nous désirions réaliser une interruption toutes les secondes. Nous avons vu qu'avec nos autres timers, ceci nous posait problème, car nous n'avons aucune valeur multiple exacte possible, d'où une erreur au final (sauf à utiliser un second quartz sur le timer1 ou à diminuer la fréquence de fonctionnement du PIC).

Nous calculons qu'avec notre quartz de 20MHz, nous avons 5.000.000 de cycles d'instruction par seconde. Nous allons donc devoir compter le plus précisément possible jusque 5.000.000.

Vous vous rendez bien compte que compter jusqu'à 5.000.000 ne peut être réalisé en un seul passage dans la routine d'interruption, le maximum étant de 65536. Nous allons donc passer un certain nombre de fois dans la routine d'interruption.

Cherchons tout d'abord le plus grand nombre de 8 bits permettant une division exacte de 5.000.000. Sans être expert en mathématiques, le nombre D'250' convient très bien.

Nous pouvons donc décider qu'après 250 passages dans notre routine d'interruption, nous aurons atteint une seconde. Ceci implique que nous devons compter jusqu'à $5.000.000/250 = 20.000$ entre 2 passages.

La valeur maximale pouvant être comptée par notre timer 2, je le rappelle, est de 256 (si PR2 = 0xFF) multiplié par 256 (prédiviseur * postdiviseur), soit 65536. Nous sommes donc en dessous, cela reste donc possible.

Nous allons maintenant essayer de trouver une combinaison « prédiviseur/postdiviseur » qui nous donne un résultat entier, en commençant par la plus forte valeur, soit 256. Souvenez-vous que les valeurs possibles de toutes les combinaisons sont :

1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,20,24,28,32,36,40,44,48,52,56,60,64,80,96,112,128,144,160,176,192,208,224,240,256

- Une valeur de division de 256 nous donne $20.000 / 256 = 78,125$
- Une valeur de division de 240 nous donne $20.000/240 = 83,33$
- Nous continuons de la sorte, toujours des résultats fractionnés
- Une valeur de division de 160 nous donne $20.000/160 = 125$

Nous voici donc avec une valeur entière. Nous devons donc configurer PR2 avec une valeur de (125 – 1), soit 124, ce qui est tout à fait dans nos possibilités.

Pour réaliser notre programme, nous devons donc :

- Configurer le prédiviseur à 16
- Configurer le postdiviseur à 10 ($10*16 = 160$)
- Placer D'124' dans PR2

Moyennant quoi, pour chaque 250^{ème} passage dans la routine d'interruption, nous aurons une durée exacte d'une seconde.

Ceci nous donne donc une précision égale à celle de notre quartz, sans utiliser de seconde horloge, qui aurait été nécessaire avec le timer1. Vous voyez qu'avec ce timer2, vous pouvez obtenir des temps configurables de façon très précise, et dans un grand éventail. C'est là la puissance du timer 2.

Mais bon, je ne vais pas vous laisser tomber ici, nous allons donc réaliser réellement un programme qui permet une fois de plus de faire clignoter notre LED à une fréquence de 1 Hz.

1 Hz, vous avez dit ? Alors ne tombons pas dans le piège, 1Hz c'est 0.5 seconde allumée et 0.5 seconde éteinte, donc nous devons compter non pas 1 seconde, mais 0.5 seconde.

Qu'à cela ne tienne, **il nous suffit de passer 125 fois dans notre routine d'interruption** au lieu de 250 fois. Simple, non ? Vous auriez besoin du dixième de seconde ? Et bien, comptez jusque 25.

Effectuez une copie de votre fichier maquette, et nommez-la « **led_tmr2.asm** ». Commençons par écrire l'en-tête et la directive config.

```
;*****
; Exercice d'utilisation du timer2 : réalisation d'une LED qui clignote *
; à une fréquence exacte de 1Hz. *
; *
;*****
;
; NOM: Led_Tmr2 *
; Date: 21/05/2002 *
; Version: 1.0 *
; Circuit: platine d'expérimentation *
; Auteur: Bigonoff *
; *
;*****
;
; Fichier requis: P16F876.inc *
; *
;*****
;
; Notes: La LED est connectée sur RB0 *
; On utilise les interruptions timer2 *
; 1 interruption toutes les 4 ms *
; *
;*****
LIST p=16F876 ; Définition de processeur
#include <p16F876.inc> ; fichier include

__CONFIG __CP_OFF & __DEBUG_OFF & __WRT_ENABLE_OFF & __CPD_OFF & __LVP_OFF &
__BODEN_OFF & __PWRTE_ON & __WDT_OFF & __HS_OSC

; __CP_OFF Pas de protection
; __DEBUG_OFF RB6 et RB7 en utilisation normale
; __WRT_ENABLE_OFF Le programme ne peut pas écrire dans la flash
; __CPD_OFF Mémoire EEprom déprotégée
; __LVP_OFF RB3 en utilisation normale
; __BODEN_OFF Reset tension hors service
; __PWRTE_ON Démarrage temporisé
; __WDT_OFF Watchdog hors service
; __HS_OSC Oscillateur haute vitesse (4Mhz<F<20Mhz)
```

Ensuite, les assignations système, et plus particulièrement celles qui nous intéressent, à savoir les valeurs à placer dans les registres de configuration et d'interruptions.

```
;*****
; ASSIGNATIONS SYSTEME *
;*****

; REGISTRE OPTION_REG (configuration)
; -----
OPTIONVAL EQU B'10000000' ; Résistances rappel +5V hors service
```

```
; REGISTRE INTCON (contrôle interruptions standard)
; -----
INTCONVAL EQUB'01000000' ; autorisation générale périphériques

; REGISTRE PIE1 (contrôle interruptions périphériques)
; -----
PIE1VAL EQUB'00000010' ; interrupt TMR2
```

Maintenant, les assignations du programme, soit la valeur à utiliser pour le compteur de passages dans la routine d'interruption, et la valeur à utiliser pour le registre PR2.

```
;*****
;
; ASSIGNATIONS PROGRAMME
;*****

PRVAL EQU D'124' ; le tmr2 compte jusque (124+1) * 160 * 0,2µs = 4ms
COMPTVAL EQU D'125' ; pour 125 passages dans tmr2 = 125 * 4ms = 500ms
```

Un petit define pour la position de la LED sur RB0 :

```
;*****
;
; DEFINE
;*****
#DEFINE LED PORTB,0 ; LED de sortie
```

La zone des variables se réduit à sa plus simple expression : une seule variable est nécessaire pour compter les passages dans la routine d'interruption. Aucune sauvegarde de registre nécessaire, puisque notre programme principal n'exécute rien.

```
;*****
;
; VARIABLES BANQUE 0
;*****
; Zone de 80 bytes
; -----

CBLOCK 0x20 ; Début de la zone (0x20 à 0x6F)
compteur : 1 ; compteur de passages dans tmr2
ENDC ; Fin de la zone
```

On arrive sur l'adresse de démarrage après reset :

```
;*****
;
; DEMARRAGE SUR RESET
;*****
org 0x000 ; Adresse de départ après reset
goto init ; Initialiser
```

La routine d'interruption n'a nul besoin des sauvegardes, ni de la gestion des interruptions multiples. Donc, elle ne comportera que la routine d'interruption timer2 en elle-même.

```
*****
;
; ROUTINE INTERRUPTION TMR2
;*****
;-----
; Un passage dans cette routine tous les 160*125*0,2µs = 4ms.
; Pas de programme principal, donc pas de sauvegarde à effectuer
; Durée d'allumage de la LED = durée d'un cycle * prédiviseur * postdiviser *
; valeur de comparaison du timer * nombre de passages dans cette routine =
```

; 0,2µs * 16 * 10 * 125 * 125 = 500.000 µs = 500ms = 0.5s

```

;-----
    org 0x004                ; adresse d'interruption

    decfsz    compteur,f      ; décrémente compteur
    goto      intend         ; pas 0, fin interruption
    movlw     COMPTVAL        ; valeur de recharge
    movwf     compteur        ; dans compteur de passage
    movlw     B'00000001'     ; valeur pour inverser LED
    xorwf     PORTB,f         ; inverser LED
intend
    bcf       PIR1,TMR2IF     ; effacer flag interrupt tmr2
    retfie                    ; retour d'interruption

```

Vous voyez que cette routine décrémente le compteur de passages, et, lorsque ce dernier atteint 0, il inverse l'état de la LED.

Nous trouvons ensuite la routine d'initialisation, qui commence par initialiser le PORTB et les registres d'interruption.

```

;*****
;                               INITIALISATIONS                               *
;*****
init
    ; Initialiser portB
    ; -----
    bsf      STATUS,RP0      ; passer banque1
    bcf      LED              ; passer RB0 en sortie

    ; Registre d'options (banque 1)
    ; -----
    movlw    OPTIONVAL       ; charger masque
    movwf    OPTION_REG      ; initialiser registre option

    ; registres interruptions (banque 1)
    ; -----
    movlw    INTCONVAL       ; charger valeur registre interruption
    movwf    INTCON          ; initialiser interruptions
    movlw    PIE1VAL         ; Initialiser registre
    movwf    PIE1            ; interruptions périphériques 1

```

Il est temps maintenant d'initialiser notre timer 2, ce qui consiste à placer les valeurs de pré et de postdiviseur, et de le mettre en service, après avoir initialisé le registre PR2 avec la valeur de « débordement ». Notez que j'utilise à dessin la notion de débordement, car celle de comparaison induit en erreur, comme je vous l'ai déjà expliqué.

```

    ; initialiser Timer 2
    ; -----
    movlw    PRVAL           ; valeur de "débordement" de tmr2
    movwf    PR2             ; dans PR2
    bcf      STATUS,RP0      ; repasser banque 0
    movlw    B'01001110'     ; postdiviseur à 10,prédiviseur à 16,timer ON
    movwf    T2CON           ; dans registre de contrôle

```

Ne reste donc plus qu'à initialiser notre variable, puis à autoriser les interruptions.

```

                ; initialiser variable
                ; -----
movlw  COMPTVAL    ; valeur de recharge
movwf  compteur    ; dans compteur de passage interruption

                ; autoriser interruptions (banque 0)
                ; -----
bsf    INTCON,GIE  ; valider interruptions

```

Quant à notre programme principal, il ne fait rien, et donc se contente de boucler sur lui-même.

```

;*****
;                                PROGRAMME PRINCIPAL                                *
;*****

start
    goto start        ; boucler
    END               ; directive fin de programme

```

Voici notre programme terminé, lancez l'assemblage, placez le fichier « .hex » dans votre PIC, et observez la LED. Elle clignote à la fréquence de 1Hz, avec la précision du quartz.

Voici donc une solution de base pour réaliser une horloge ou d'autres applications basées sur la mesure du temps en secondes. Nul besoin de quartz spécial avec notre timer2.

Si vous décidez de simuler ce programme, vous pouvez, dans la fenêtre des registres spéciaux, remarquer les registres « t2pre » et « t2post » (tout en bas) qui indiquent le contenu du pré et du post diviseur du timer. Ces registres ne sont pas accessibles par votre programme, ils sont simulés par MPLAB pour vous montrer le déroulement de votre programme.

Remarquez en mode « pas à pas » que la valeur D'124' dans TMR2 ne provoque ni le reset de TMR2, ni le positionnement du flag TMR2IF. C'est au moment où TMR2 devrait passer à D'125' que ces opérations s'effectueront.

18. Récapitulatif sur les timers

18.1 Le choix du timer

Vous vous demandez peut-être comment choisir votre timer dans une application particulière. En fait, **il s'agit de rester logique**. Prenons quelques cas concrets :

18.1.1 Vous désirez mesurer un temps compris entre 256 et 65536 cycles.

Dans ce cas, **tout dépend de la précision demandée, et de la valeur à mesurer**. S'il s'agit d'une valeur multiple de 256 (par exemple 512), vous pouvez utiliser le timer0, 1, ou 2 sans aucun problème et avec une précision maximale.

Si, par contre, vous devez mesurer des nombres de cycles non divisibles par des puissances de 2, et qu'une grande précision est demandée, vous avez 3 options :

- Soit vous utilisez le timer2, en utilisant la technique précédemment décrite.
- Soit vous utilisez le timer1, qui dispose de la possibilité de comptage jusque 65536 sans utiliser de prédiviseur
- Soit vous utilisez le timer0, mais au prix d'une perte de précision ou d'une programmation plus délicate.

18.1.2 Vous désirez mesurer des temps allant jusque 524288

Vous pouvez alors utiliser le timer 1, avec un prédiviseur. Naturellement, si la valeur à atteindre n'est pas un multiple exact du prédiviseur, votre précision en sera altérée.

18.1.3 Vous désirez mesurer des temps quelconques avec une grande précision

Dans ce cas, si on admet que le nombre de cycles à mesurer n'est pas un multiple exact d'une valeur de diviseur, vous disposez de 2 méthodes principales et simples pour obtenir ces mesures de temps :

- Soit vous utilisez le timer1 avec un second quartz calculé de façon à obtenir des multiples exacts. Ceci présente l'avantage de moins utiliser de temps de PIC, puisque la seconde horloge aura une fréquence maximale de 200KHz, ce qui permet d'espacer les interruptions. Par contre, ceci nécessite de sacrifier 2 pins et d'utiliser plus de matériel.
- Soit vous utilisez le timer2, en calculant des valeurs multiples de pré et postdiviseur et en plaçant le résultat dans PR2. Une longue durée sera prise en compte en comptant les passages dans la routine d'interruption. Cette méthode est plus simple et plus économique, mais nécessite plus de temps d'exécution, surtout pour les longues durées pour lesquelles des interruptions seront « sacrifiées » dans le seul but de les compter. Le temps doit bien entendu être multiple de la durée d'une instruction.

18.1.4 Vous désirez compter des événements

Dans ce cas le timer2 est inutilisable, reste donc :

- Soit le timer 0 qui permet de choisir entre détection des flancs montants ou descendants, mais limite le comptage à 256 (multiplié par le prédiviseur)
- Soit le timer 1 qui permet de compter jusqu'à 65536 sans prédiviseur, mais qui impose de détecter uniquement les flancs montants de votre signal.

Je pourrais multiplier les exemples, mais le but était simplement de vous montrer que chaque timer est plus ou moins bien adapté au but recherché. Dans la grande majorité des applications, le choix du timer nécessitera une petite réflexion.

Prenez garde que l'électronique est liée au logiciel, les deux devront donc être étudiés en parallèle. En effet, pour simple exemple, si vous avez besoin des caractéristiques du timer1 pour compter des événements, vous serez alors dans l'obligation de connecter le signal sur T1CKI et non sur T0CKI. De plus, si vous désiriez compter des flancs descendants, vous devrez ajouter un montage qui inverse votre signal d'entrée.

18.2 Les bases de temps multiples

Il vous arrivera probablement de devoir utiliser plusieurs bases de temps dans le même programme.

La première solution qui vient à l'esprit est d'utiliser plusieurs timers. Ceci peut se révéler la bonne solution, mais ce n'est pas toujours nécessaire. En effet, si vos bases de temps disposent d'un dénominateur commun, vous pouvez utiliser ce dénominateur comme base de temps générale, et compter les passages dans la routine d'interruption.

Imaginons par exemple que vous ayez besoin de gérer une impulsion sur une broche toutes les 200ms, et que vous désirez faire clignoter une LED à une fréquence de 1Hz. Vous pouvez alors décider d'utiliser un timer pour la LED, un autre pour les impulsions.

Mais vous pouvez aussi vous dire : j'ai besoin d'une durée de 500ms pour ma LED, et d'une autre de 200ms pour les impulsions. Je décide de créer une durée d'interruption de 100ms.

Ma routine d'interruption sera donc du type :

- Je décompte compteur 1
- Compteur 1 = 0 ?
- Non, je ne fais rien
- Oui, je gère mon impulsion, et je recharge compteur 1 avec 2 (200ms)
- Je décrémente compteur 2
- Compteur 2 = 0 ?
- Non, fin d'interruption
- Oui, je gère ma LED et je recharge compteur 2 avec 5 (500ms)
- Fin d'interruption

Vous voyez donc qu'avec cette méthode, je gère 2 temps différents avec le même timer. Dans la pratique, c'est une méthode que vous utiliserez probablement souvent.

18.3 Possibilités non encore abordées

Arrivé à ce stade, vous ne disposez pas encore de toutes les possibilités des performances et des utilisations possibles des timers.

Pour compléter vos connaissances, et pour appréhender toutes les remarquables diversités d'application, il vous reste encore à étudier les modules CCPx abordés dans un chapitre ultérieur.

Vous verrez alors que vous pouvez disposer de fonctions supplémentaires, et même vaincre ce que vous pensiez être les limites des timers concernés.

Notes : ...

19. Le convertisseur analogique/numérique

19.1 Préambule

Tout d'abord, je vais vous demander de ne pas paniquer. En effet, je vais détailler le fonctionnement du convertisseur analogique/numérique, afin de permettre de l'exploiter dans ses moindres ressources.

Ce faisant, vous allez trouver ici des formules mathématiques pour le moins barbares. Mais, rassurez-vous, dans l'immense majorité des cas vous n'aurez pas besoin de ces formules. En effet, je vous donnerai les valeurs sûres à employer pour les éviter, au prix d'une légère perte de performance au niveau du temps de conversion, perte qui n'a généralement pas la moindre importance.

Beaucoup de personnes me demandent comment effectuer une conversion analogique/numérique, mais je m'aperçois que peu savent ce que cette conversion représente réellement et quelle sont ses limites. Je crains donc qu'il ne faille commencer par un peu de théorie.

De plus, cette approche présente l'avantage de vous permettre de vous sortir de toutes les situations futures qu'il m'est impossible de prévoir à l'heure actuelle, par exemple, sortie d'une nouvelle version de 16F87x avec des caractéristiques temporelles différentes de celles du PIC étudié ici.

Bien entendu, si tout ceci ne vous intéresse pas, ou que vous jugez que cela ne peut rien vous apporter, vous êtes libres de tourner les pages à votre convenance.

19.2 Nombres numériques, analogiques et conversions

Mais commençons donc par le commencement. Qu'est-ce qu'un Convertisseur Analogique/Digital (CAD), de préférence nommé Convertisseur Analogique/Numérique (CAN), ou Analogic to Digital Converter (ADC) ?

Bien qu'on utilise souvent la notion de « convertisseur analogique/digital », la bonne expression française serait plutôt « convertisseur analogique/numérique ». Mais bon, je ne suis pas ici pour vous donner une leçon de français, j'en suis du reste incapable.

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, c'est-à-dire, en fait, connaître la valeur de la tension présente sur une pin de votre PIC. Il est des tas d'applications où vous aurez besoin d'une telle possibilité, par exemple si vous voulez mesurer la tension de votre batterie à l'aide d'un PIC.

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.

Il importe à ce niveau de rappeler ce qu'est un signal analogique. Commençons par examiner une tension, par exemple la tension de votre alimentation. Cette tension peut s'exprimer par une infinité de valeur, et vous pouvez faire varier cette tension de façon continue, sans qu'il y ait un « trou » entre 2 valeurs.

Un nombre numérique, on contraire, dispose d'un nombre fini de valeurs, on parlera de valeurs « discrètes ». Par exemple, dans un octet, vous pouvez coder 0x01 ou 0x02, mais il n'y a pas de valeur intermédiaire, au contraire de votre valeur analogique, pour laquelle vous pouvez toujours insérer une telle valeur.

Le problème est donc en premier lieu de savoir comment passer mathématiquement d'une représentation à l'autre, donc comment convertir du numérique vers l'analogique et réciproquement.

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 (ne me dites pas 0,1) ? En fait, si vous réfléchissez un peu, vous voyez que 128,13 sera converti en 128,1, tandis que 128,16 sera converti en 128,2. L'erreur maximal obtenue est donc de la moitié du plus faible digit. 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 représente la plage de valeurs de la grandeur analogique divisée par la plus grande valeur représentée par notre nombre numérisé.

Ceci, bien entendu, en partant du principe que nos nombres numériques commencent à « 0 ». Dans le cas contraire, « la plus grande valeur » sera remplacée par « la différence entre la plus grande et la plus petite valeur possible ».

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.

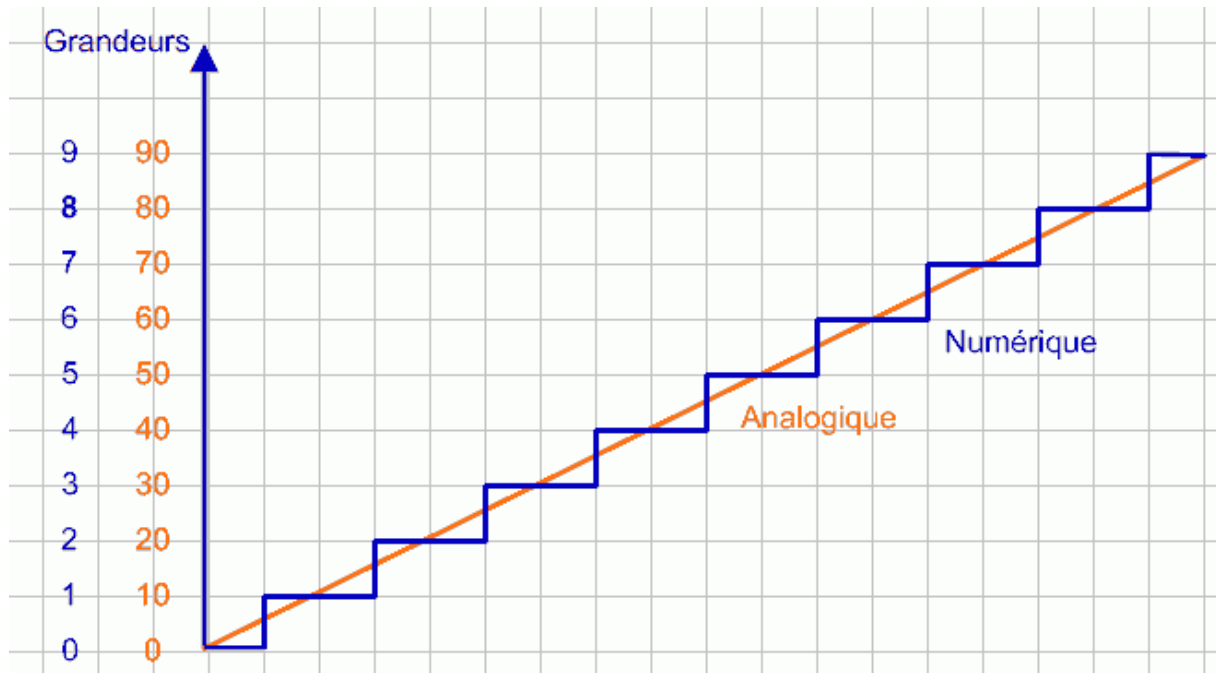
Si maintenant, nous désirons effectuer l'opération inverse, c'est-à-dire savoir à quelle valeur analogique correspond une valeur numérique, nous pourrions dire que :

La valeur analogique typique est égale à la valeur analogique minimale représentée à laquelle on ajoute le pas multiplié par la valeur numérique correspondante.

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 certains valeurs analogiques (par exemple 15), nous avons 2 valeurs numériques possibles. La même situation se retrouve dans la vie réelle. Si vous donnez 10 euros à un commerçant pour payer un montant de 7,565 euros, libre à lui de considérer que vous lui devez 7,57 plutôt que 7,56 euros. 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.**

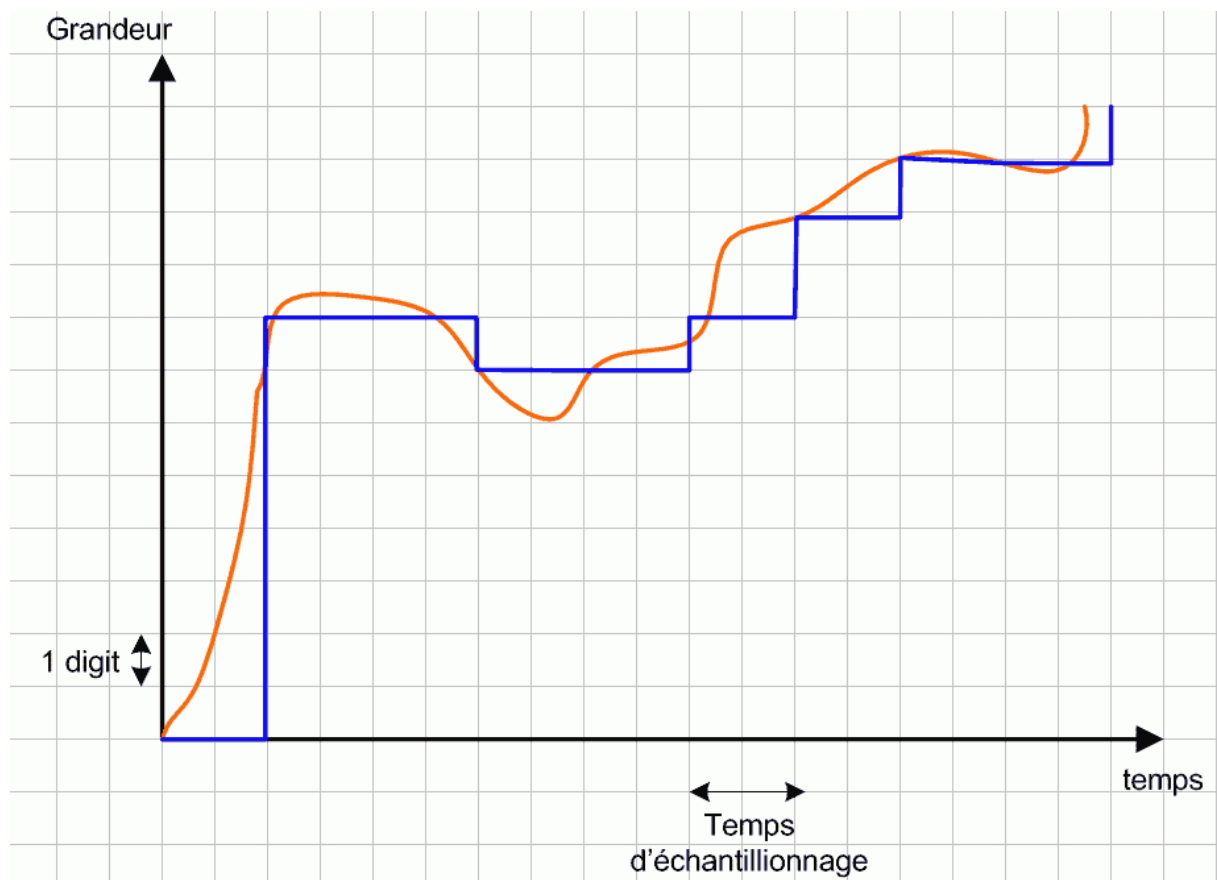
C'est comme si, par exemple, vous preniez une série de photos successives avec un appareil rapide. Lorsque vous visualisez l'animation, vous aurez une succession d'images fixes qui restitueront l'animation.

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 Fourier et autres, pour simplement vous dire que pour un signal sinusoïdal, on admet que la fréquence d'échantillonnage doit être au minimum le double de la fréquence du signal à mesurer.

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 44100 Hz, ce qui nous donne, suite au théorème précédent, une fréquence maximale sinusoïdale digitalisée 22 KHz (l'oreille humaine parvient en général à capter des fréquences maximales de 16KHz, 20KHz pour certaines personnes).

Anecdote en passant, ceci vous démontre l'inutilité des enceintes qui sont annoncées avec des fréquences maximales de plus de 50 KHz. D'une part, votre CD n'a pas enregistré de telles fréquences, d'autre part votre oreille est incapable de les interpréter. Quand on voit un chiffre, il faut toujours se demander quelle est la réalité qui se cache derrière (autre que commerciale).

19.3 Principes de conversion sur les 16F87x

Jusqu'à présent, nous venons de raisonner en décimal. Les pics, 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 (cours-part1).

Notre 16F87x 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 pics 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 :

- 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.

- La pin est déconnectée du condensateur, et ce dernier est connecté sur le convertisseur analogique/numérique interne.
- 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.

19.4 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, les formules sont donc dérivées de celles que nous avons calculées lors de la réalisation de notre circuit anti-rebond du chapitre sur le timer 1.

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 :

$$Tamp = 2\mu s$$

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 t° du pic ne pouvant pas, en effet, excéder 50°C.

$$0 \leq T_{\text{coff}} \leq 1,25 \mu\text{s}$$

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 16F876.

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 à une valeur de 120pF pour les versions actuelles de PIC (16F876), les formules que je vous ai données pour le calcul du temps de chargement d'un condensateur restant valables, la formule du temps de charge du condensateur est :

$$T_c = -C * (R_{\text{interne}} + R_{\text{source}}) * \ln(1/2047)$$

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 2046/2047^{ème} de la tension à mesurer.

Comme « C » est fixe et « ln(1/2047) » également, je vais vous calculer la constante une fois pour toutes (n'est-ce pas que je suis bon avec vous ?) :

$$-C * \ln(1/2047) = 0,914895 * 10^{-9}$$

La formule devient donc :

$$T_c = 0,914895 * 10^{-9} * (R_{\text{interne}} + R_{\text{source}})$$

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

$$T_c = 0,914895 * 10^{-9} * (10 * 10^3 + 12 * 10^3)$$

$$T_c \text{ maximal} = 20,12 \mu\text{s}$$

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

$$T_c = 0,914895 * 10^{-9} * (0 * 10^3 + 6 * 10^3)$$

$$T_c \text{ minimal} : 5,48 \mu\text{s}.$$

Vérifiez dans le datasheet actuel les tensions maximales autorisées pour les PICs. Ces dernières sont sujettes à fréquentes modifications. A l'heure actuelle, la tension maximale autorisée est de 7,5V. CONSULTEZ LES DATASHEETS LES PLUS RECENTS S'IL ENTRE DANS VOS INTENTIONS D'UTILISER CETTE TENSION MAXIMALE.

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 :

$$T_c = 0,914895 * 10^{-9} * (10 * 10^3 + 8 * 10^3)$$

$$T_c \text{ typique} = 16,46 \mu\text{s}.$$

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 :

$$T_{acq} = T_{amp} + T_c + T_{coff}$$

$$T_{acq} \text{ maximum} = 2\mu\text{s} + 20,12\mu\text{s} + 1,25\mu\text{s} = 23,37 \mu\text{s}$$

Le cas le plus favorable, une température inférieure ou égale à 25°C et un Tc minimal, nous donne :

$$T_{acq} \text{ minimum} = 2\mu\text{s} + 5,48\mu\text{s} = 7,48 \mu\text{s}.$$

Maintenant, pour nos utilisations classiques, sous 5V, nous aurons dans le pire des cas :

$$T_{acq} \text{ sous 5V} = 2\mu\text{s} + 16,46\mu\text{s} + 1,25\mu\text{s} = 19,71\mu\text{s}.$$

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.

19.5 La conversion

Arrivé à ce stade, après le temps T_{acq} , 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.**

C'est une méthode de type dichotomique, c'est un bien grand mot pour exprimer une méthode somme toutes assez intuitive. Il s'agit tout simplement de couper l'intervalle dans lequel se trouve la grandeur analogique en 2 parties égales, et de déterminer dans laquelle de ces 2 parties se situe la valeur à numériser. Une fois cet intervalle déterminé, on le coupe de nouveau en 2, et ainsi de suite jusqu'à obtenir la précision demandée.

Prenons un exemple pratique : vous avez un livre de 15 pages, vous en choisissez une au hasard, supposons la numéro 13. Voici comment vous allez procéder pour trouver de quelle page il s'agit.

- On coupe l'intervalle en 2, arrondi à l'unité supérieure, soit 8.
- Le numéro de page est-il supérieur, inférieur ou égal à 8 ?
- Le numéro est supérieur, donc dans l'intervalle 8-15
- On coupe cet intervalle en 2, soit 12
- Le numéro est-il supérieur, inférieur, ou égal à 8 ?
- Le numéro est supérieur, donc dans l'intervalle 12-15
- On coupe l'intervalle en 2, soit 14
- Le numéro est-il supérieur, inférieur, ou égal à 14 ?
- Le numéro est inférieur, donc dans l'intervalle 12-14
- Le numéro est donc 13, puisqu'il n'était égal ni à 14, ni à 12

Cette méthode peut paraître curieuse, mais elle est d'une grande efficacité en terme de temps. Chaque question se traduisant par une opération à effectuer au niveau électronique, on peut dire que moins de question il y a, moins de temps l'opération de conversion prendra. Si vous choisissez une page au hasard parmi un livre énorme de 65535 pages, vous verrez que vous pouvez trouver le bon numéro de page en moins de 16 questions.

Appliquée à la numérotation binaire, cette méthode est de plus particulièrement bien adaptée, puisque couper un intervalle en 2 revient à dire qu'on force simplement un bit à 1.

Reprenons notre exemple précédent, mais en raisonnant en binaire.

Notre nombre de pages maximum est de $B'1111'$, soit $D'15'$, la page choisie est $B'1101'$, soit $D'13'$. Effectuons notre conversion.

- On coupe l'intervalle en 2, soit $B'1000'$.
- **Le numéro de page est-il inférieur ?**
- Non, donc compris entre $B'1000'$ et $B'1111'$
- On coupe l'intervalle en 2, soit $B'1100'$

- Le numéro de page est-il inférieur?
- Non, donc compris entre B'1100' et B'1111'
- On coupe l'intervalle en 2, soit B'1110'
- Le numéro de page est-il inférieur?
- Oui, donc compris entre B'1100' et B'1101'
- On coupe l'intervalle en 2, soit B'1101'
- Le numéro de page est inférieur ?
- Non, donc le numéro de page est B'1101'

Vous voyez qu'en fait vous avez une question (approximation) par bit du résultat. Autrement dit, en raisonnant en terme de bits :

- On place le bit 3 à 1, donc B'1000'
- Le résultat est-il inférieur?
- Non, alors le bit 3 vaut effectivement « 1 »
- On place le bit 2 à 1, donc B'1100'
- Le résultat est inférieur?
- Non, alors le bit 2 vaut effectivement « 1 »
- On place le bit 1 à 1, donc B'1110'
- Le résultat est-il inférieur ?
- Oui, alors le bit 1 ne valait pas « 1 », mais « 0 »
- On place le bit 0 à 1, donc B'1101'
- Le résultat est inférieur ?
- Non, alors le bit 0 valait bien « 1 », le résultat est donc B'1101'

Pour résumer, le temps nécessaire à la conversion est égal au temps nécessaire à la conversion d'un bit multiplié par le nombre de bits désirés pour le résultat.

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 $T_{osc}/4$.

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 16F876, 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 **contraints** en utilisant cette horloge de placer votre PIC en mode sleep jusque 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 T_{ad} ne peut descendre, pour des raisons électroniques, en dessous de $1,6\mu s$ pour les versions classiques de 16F87x, et en dessous de $6\mu s$ 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 PICs de type classique. La formule d'obtention des temps T_{ad} est simple, puisqu'il s'agit tout simplement, comme expliqué ci-dessus, du temps d'instruction (T_{osc}) divisé par le diviseur donné. Exemple, à 20Mhz, le temps d'instruction est de $1/20.000.000$, soit 50ns. Donc, avec un diviseur de 2, on aura 100ns.

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

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 T_{ad} serait inférieur à $1,6\mu s$. Quand 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 remarquerez 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 T_{ad} avant le démarrage effectif de la conversion, et un temps supplémentaire T_{ad} à la fin de la conversion. Donc, le temps total de conversion est de :

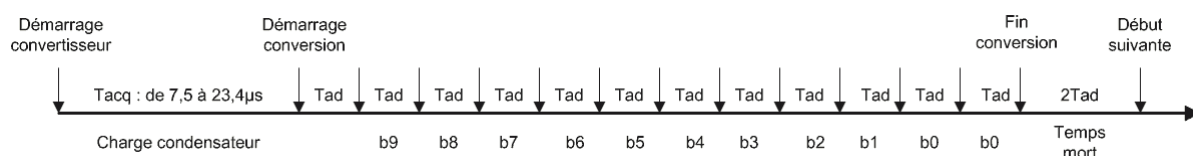
- T_{ad} : avant le début de conversion (le temps de connexion du condensateur est inclus)
- $10 * T_{ad}$ pour la conversion des 10 bits du résultat
- T_{ad} supplémentaire pour la fin de la conversion de b_0

Soit, au total, un temps de $12 T_{ad}$, soit dans le meilleur des cas, un temps de $12 * 1,6\mu s = 19,2 \mu s$.

Notez qu'un temps équivalent à $2 * T_{ad}$ 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 T_{acq})
- On effectue la conversion (nécessite le temps $12 * T_{ad}$)
- On doit attendre $2 * T_{ad}$ 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 T_{acq} 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 T_{ad} de $1,6 \mu s$** (attention, je ne travaille que sur les PICs 16F87x standards, pour les modèles différents, je vous ai donné toutes les méthodes de calcul) :

T entre 2 échantillonnages = $T_{acq} + 12 T_{ad} + 2 T_{ad} = T_{acq} + 14 T_{ad}$, donc :

$$T = 19,71 \mu s + 14 * 1,6 \mu s = 42,11 \mu s.$$

Ceci correspond donc à une fréquence de :

$$F = 1/T = 1 / 42,11 * 10^{-6} = 23747 \text{ Hz}.$$

Cette fréquence vous permet donc d'échantillonner des signaux sinusoïdaux d'une fréquence maximale de 11874 Hz (la moitié de la fréquence d'échantillonnage).

Si vous vous placez **dans le meilleur des cas**, vous ramenez T_{acq} à $7,5 \mu s$, ce qui vous donne :

$$T = 7,5 \mu s + 14 * 1,6 \mu s = 29,9 \mu s, \text{ soit une fréquence de } F = 33445 \text{ Hz}.$$

Soit la possibilité de numériser des signaux sinusoïdaux d'une fréquence maximale de 16722 Hz (si ce n'était la résolution sur 10 bits au lieu de 16, on serait dans le domaine de la hi-fi).

19.6 Compromis vitesse/précision

Nous avons vu que la conversion s'effectue sur 10 bits, avec une fréquence maximale possible de 33445 Hz. Si vous avez besoin d'une vitesse plus grande, il est possible d'utiliser quelques trucs.

La première chose à savoir, c'est que **l'augmentation de vitesse au delà de ces limites ne peut se faire qu'au détriment de la précision en terme de nombre de bits significatifs du résultat**. Cette augmentation se paye également par une plus grande difficulté au niveau de la programmation.

Voici le raisonnement employé. Si on accepte de limiter la précision à un nombre inférieur à 10 bits, et en constatant que la numérisation s'effectue en commençant par les bits les plus significatifs, on peut se dire : N bits me suffisent, donc en regardant notre figure précédente, on décide de stopper la conversion après que les bits en question aient été numérisés.

Donc, on se dit que pour conserver N bits significatifs, on aurait :

- Tacq pour charger le condensateur
- Tad pour le démarrage
- $N * Tad$ pour la conversion

Soit, pour une conversion en conservant 4 bits au lieu de 12, cela ramènerait notre temps total à $Tacq + Tad + 4 Tad$, soit $Tacq + 5 Tad$. Nous économisons donc 7 Tad.

Malheureusement, l'arrêt de la numérisation n'est pas possible, car le résultat ne serait pas enregistré dans les registres concernés. Il faut donc améliorer cette astuce. En fait, on ne peut arrêter la numérisation, mais on peut changer la valeur du diviseur en cours de digitalisation.

Les bits numérisés après la diminution du diviseur ne seront pas valides, mais ce n'est pas important, puisque nous avons décidé de ne pas les utiliser.

Supposons donc que nous travaillions avec un PIC à 20MHz. Nous utilisons donc le diviseur par 32. Chaque Tad vaut donc 32 Tosc.

Notre temps de conversion effectif prend donc normalement 12 Tad, soit $12 * 32 Tosc$, soit $19,2\mu s$. Si nous avons besoin de 4 bits valides, nous pouvons donc décider de réaliser les opérations suivantes :

- Tacq pour charger le condensateur
- Tad pour le démarrage = 32 Tosc
- $N * Tad$ pour la conversion = $N * 32 Tosc$
- Modification du diviseur pour passer à 2
- $(11 - N) Tad$ restants pour le reste de la conversion (qui sera erronée), donc $(11-N) * 2Tosc$

Donc, nous épargnons en réalité $(11-N) * (32-2)Tosc$, soit pour notre numérisation sur 4 bits :

$$(11-4) * 30 Tosc = 210 Tosc, \text{ soit } 210 * 50ns = 10,5\mu s.$$

La formule générale du temps de conversion pour numériser sur N bits est donc :

$$\text{Temps de conversion sur N bits} = Tad + N * Tad + (11-N) (2Tosc)$$

Autrement dit :

$$\text{Temps de conversion} = (N+1) Tad + (11-N) (2Tosc).$$

Ce qui nous donnera, pour notre conversion sur 4 bits pour un PIC à 20Mhz:

$$\text{Temps de conversion} = (4+1) * 1,6\mu s + (11-4) * 100ns = 8\mu s + 0,7\mu s = 8,7\mu s.$$

Donc le temps total de numérisation sera :

$$\text{Temps numérisation} = Tacq + \text{Temps de conversion} = 7,48\mu s + 8,7\mu s = 16,18\mu s.$$

A ceci nous ajoutons $2T_{ad}$, soit 200ns, ce qui nous donne un temps séparant 2 échantillonnages de $16,38\mu s$

Ceci nous donnera une fréquence d'échantillonnage de :

Fréquence d'échantillonnage = $1/16,38\mu s = 61050$ Hz. avec 16 valeurs différentes possibles.

La seconde astuce utilisable, est de constater que si on ne désire pas utiliser tous les bits, il est également inutile de charger le condensateur au maximum. On peut donc reprendre les calculs de charge du condensateur, afin de diminuer également T_{acq} , ce qui permet encore d'augmenter légèrement la fréquence d'échantillonnage.

Remarquez que l'utilisation de ces techniques vous donnera toujours un résultat sur 10 bits, mais dont seuls les N premiers représenteront encore une valeur utilisable. Les autres bits devront donc être ignorés.

Bien entendu, il vous incombera de déterminer le temps qui sépare le début du processus de conversion de celui de modification du diviseur. Cette modification interviendra logiquement un temps T_{ad} (N+1) après lancement de la conversion. T_{ad} étant un multiple exact (généralement 32) de l'oscillateur principal, une simple petite boucle, ou l'utilisation d'un timer pourra faire l'affaire.

Par exemple, pour digitaliser sur 6 bits avec un PIC à 20MHz :

- Vous lancez l'acquisition
- Après le temps T_{acq} , vous lancez la conversion
- Après le temps $(6+1) * T_{ad}$ vous modifiez le diviseur
- Vous attendez la fin de la conversion

Le temps d'attente $(6+1) T_{ad}$ vaut $7 * T_{ad}$, donc, $7 * 32 T_{osc}$. Sachant que le temps d'exécution d'une instruction est de $T_{osc}/4$, le temps d'attente sera de $7 * 32 / 4 = 56$ cycles d'instruction.

19.7 Les valeurs représentées

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 :

$$Val = ((V_{IN} - V_{REF-}) / (V_{REF+} - V_{REF-})) * 1023)$$

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

$$V_{IN} = ((Val/1023) * (V_{REF+} - V_{REF-})) + V_{REF-}$$

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 :

$$Val = (V_{IN} / V_{REF+}) * 1023)$$

$$V_{IN} = (Val/1023) * V_{REF+}$$

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 :

$$Val = (V_{IN} / 5) * 1023)$$

$$V_{IN} = (Val/1023) * 5$$

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. Les formules sont tirées de la loi d'ohm, et ont déjà été utilisés dans le calcul du circuit anti-rebond dont nous nous sommes servis dans le cadre du circuit de comptage de notre timer1.

19.8 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, comme expliqué plus haut.
- 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 :

$$\text{Val numérisée} = ((V_{\text{IN}} - V_{\text{REF-}}) / (V_{\text{REF+}} - V_{\text{REF-}})) * 1023$$

$$V_{\text{IN}} \text{ analogique} = ((\text{Val}/1023) * (V_{\text{REF+}} - V_{\text{REF-}})) + V_{\text{REF-}}$$

$$\text{Temps de conversion sur N bits} = T_{\text{ad}} + N * T_{\text{ad}} + (11-N) (2T_{\text{osc}})$$

$$T_{\text{ad}} = T_{\text{osc}} * \text{diviseur} \geq 1,6 \mu\text{s}$$

$$\text{Temps de conversion sur 10 bits} = 12 T_{\text{ad}}$$

$$T_{\text{acq}} = 2\mu\text{s} + T_{\text{c}} = 0,914895 * 10^{-9} * (R_{\text{interne}} + R_{\text{source}}) + 0,05(T^{\circ} - 25^{\circ}\text{C}) \text{ avec } T^{\circ} \geq 25^{\circ}\text{C}$$

Et également, les valeurs que vous pouvez utiliser dans la majorité des cas :

Tacq courant : 19,7μs

Temps de conversion courant : 19,2μs.

Temps entre 2 numérisation successives : 3,2μs

19.9 La théorie appliquée aux PICs : 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 ou nos signal (signaux) analogique(s) 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 5 canaux d'entrée analogique. Vous pouvez donc échantillonner successivement jusqu'à 5 signaux différents avec ce composant. Les pins utilisées sont les pins AN0 à AN4 (qui sont en fait les dénominations analogiques des pins RA0 à RA3 + RA5).

Le 16F877, quant à lui, dispose de 8 canaux d'entrée analogique. Vous pourrez donc échantillonner jusqu'à 8 signaux différents sur 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.

19.10 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 (V_{ref-}) et une tension de référence maximale (V_{ref+}).

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

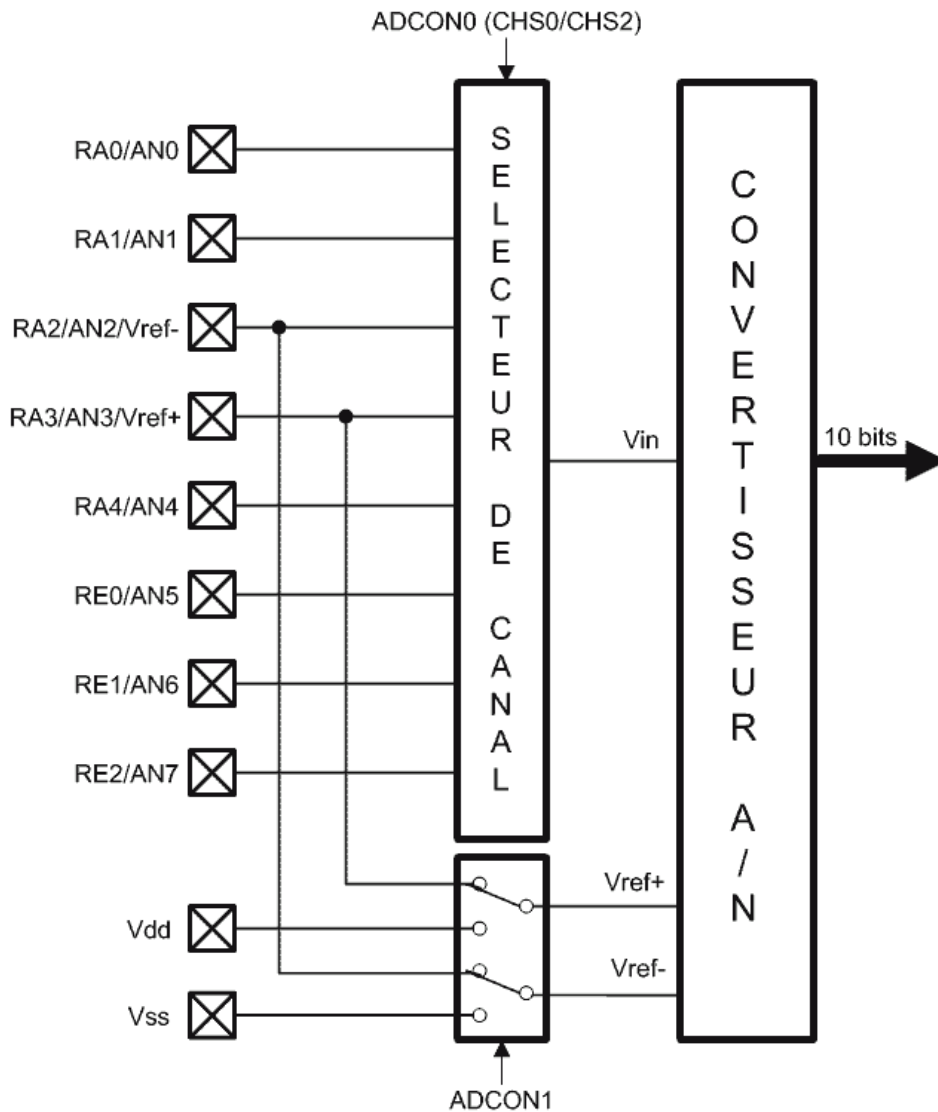
- Utilisation de V_{ss} (masse du PIC) comme tension V_{ref-} et de V_{dd} (alimentation positive du PIC) comme tension V_{ref+} . Dans ce mode, les tensions de références sont tirées en interne de la tension d'alimentation. Il n'y a donc pas besoin de les fournir.
- Utilisation de la pin V_{ref+} pour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

Notez que la broche V_{ref+} est une dénomination alternative de la broche RA3/AN3, tandis que la broche V_{ref-} 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 V_{ref+} et V_{ref-} 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 convertisseur numérique/analogique dont n'est pas pourvu notre PIC.

Nous pouvons maintenant dessiner le schéma symbolique des entrées de notre convertisseur analogique/numérique. Ce schéma correspond à un 16F877, pour le 16F876, les canaux AN5 à AN7 n'existent bien entendu pas.



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) :

- On choisit le canal à numériser, et on met en route le convertisseur
- On attend T_{acq}
- On lance la numérisation
- On attend la fin de la numérisation
- On attend $2 T_{ad}$
- 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 :

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 $Val = (V_{IN} / V_{REF+}) * 1023$, que de :

$$(2/5) * 1023 = 409 \text{ pour une tension de 2V à}$$

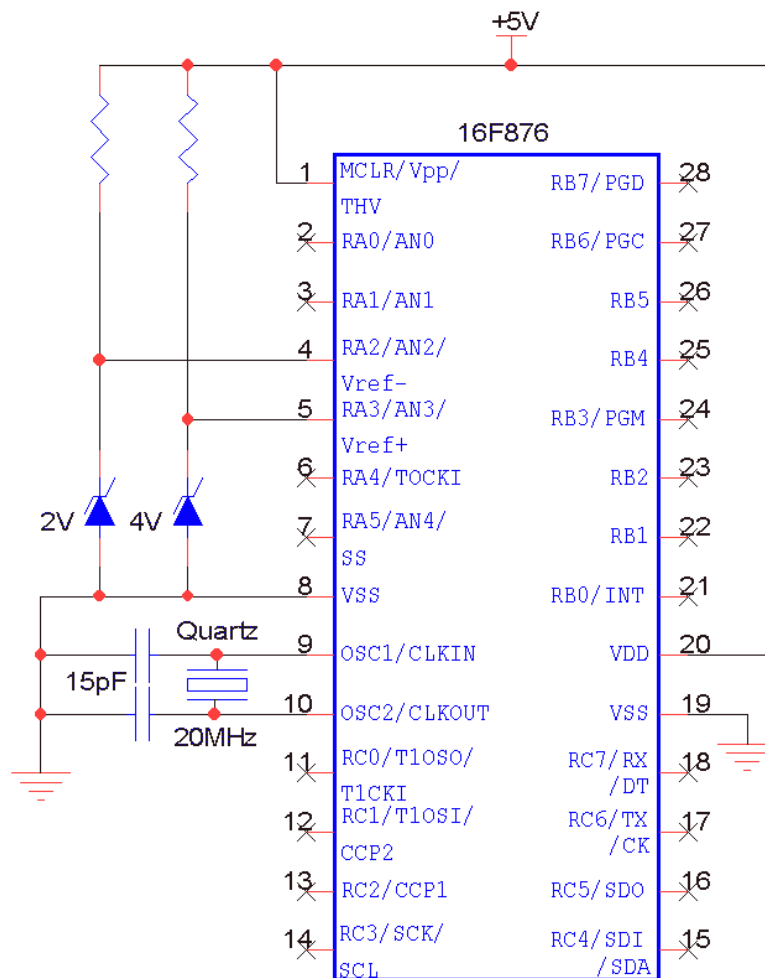
$$(4/5) * 1023 = 818 \text{ pour une tension de 4V.}$$

Votre précision sera donc de 409 pas sur les 1023 possibles, les autres valeurs étant inutilisés. Vous avez donc une perte de précision. Pour faire une équivalence avec le monde de la photo, vous réalisez ici un zoom numérique sur la plage 2/4V.

Par contre, si vous utilisez une tension de référence de 2V comme V_{ref-} et une de 4V comme V_{ref+} , 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.

Dans notre analogie avec les appareils photographiques, nous avons réalisé par cette méthode, un zoom optique sur la plage de tension 2/4V. Comme avec le zoom optique, le reste du cliché (le reste des tensions) n'est pas capturé par le PIC, mais, en contrepartie, la précision reste maximale dans la zone cadrée.

Voici le schéma que vous devrez utiliser, après avoir paramétré ADCON1 en conséquence :



Les résistances seront calculées en fonction des diodes zener et de la loi d'ohm : la résistance est égale à la tension au borne de la résistance ($5V - \text{tension de la zener}$) divisée par le courant qui doit traverser la zener (généralement quelques mA.).

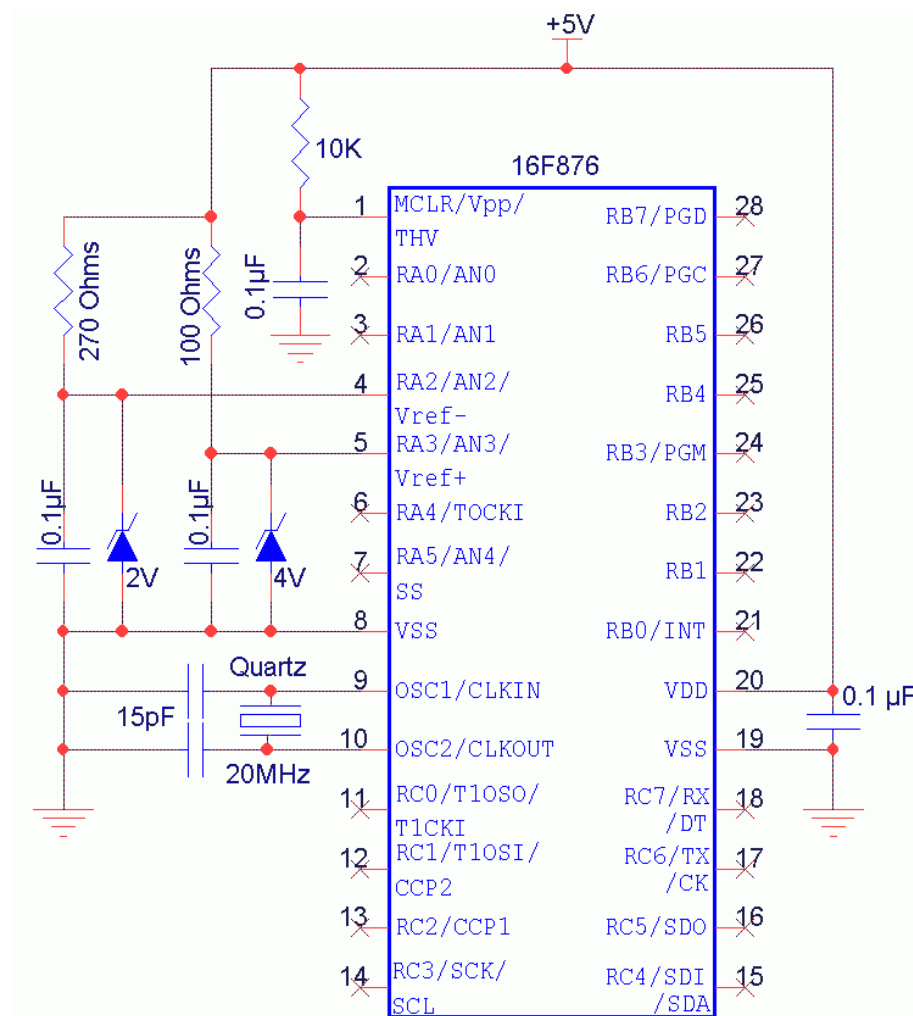
Vous constatez que ceci vous permet d'avoir des tensions de référence qui ne varient pas en fonction de la tension d'alimentation. Il peut donc être pratique, si votre alimentation n'est pas stable, d'utiliser des tensions de références fortement stabilisées.

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

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



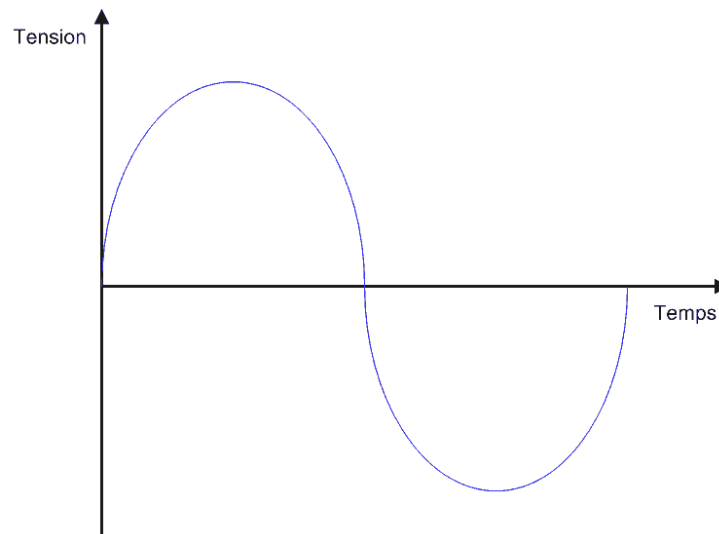
Notez que si vous désirez mesurer, par exemple, la position d'un potentiomètre, l'erreur s'annulera d'elle-même si vous n'utilisez pas les tensions de référence externes. En effet, si la tension d'alimentation varie, la référence interne +Vdd variera également. Comme le potentiomètre pourra être alimenté 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.

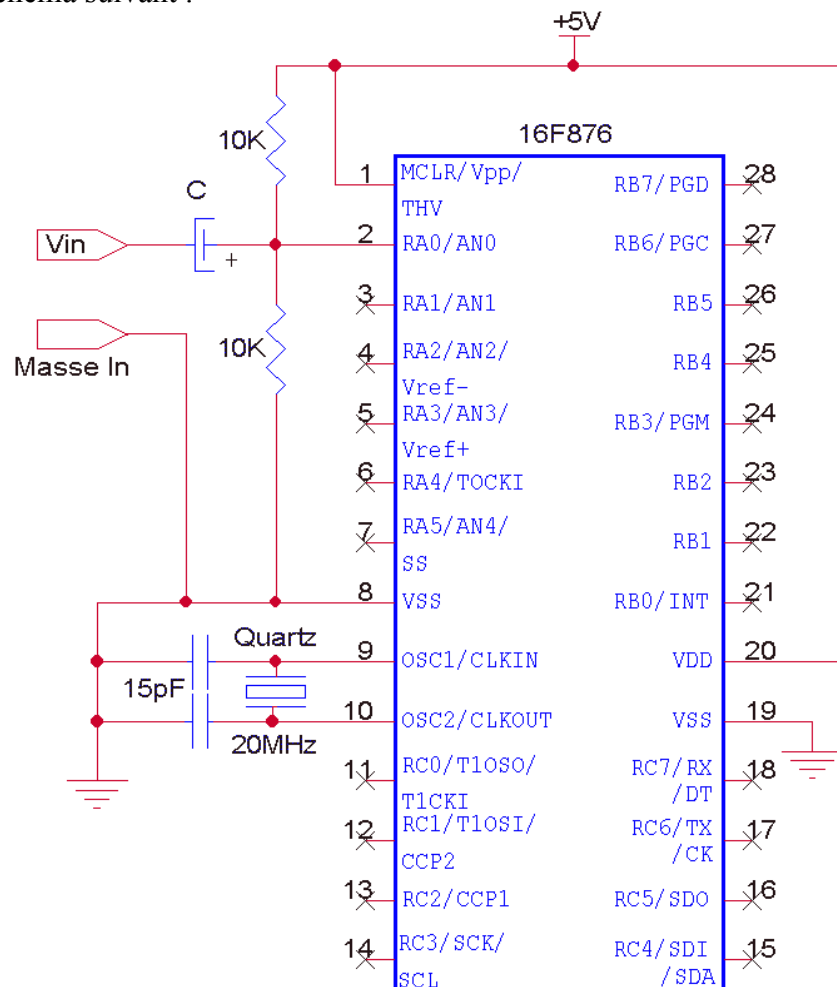
Nous allons poursuivre l'étude de notre convertisseur par l'étude des registres dont nous venons de dévoiler le nom. Mais, auparavant :

19.11 Mesure d'une tension alternative

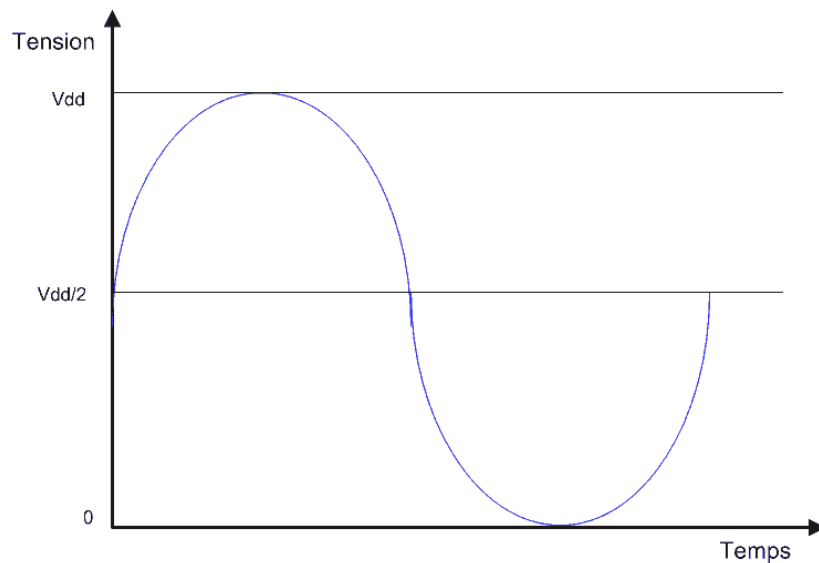
Nous avons vu que notre tension pouvait varier entre Vss et Vdd. Nous ne pouvons donc pas échantillonner directement une tension alternative, comme un signal audio, par exemple. Celui-ci est en effet de la forme :



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 $V_{dd}/2$, et en amenant la tension alternative via un condensateur, de façon à effectuer une addition des 2 tensions. Regardez le schéma suivant :



Ceci ramènera la tension à mesurer vue par le PIC à :



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 (je vous passe la théorie), 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 :

$$Z_c = 1 / (2 * \pi * f * C), \text{ donc}$$

$$C = 1 / (2 * \pi * f * Z_c)$$

Avec :

Z_c = impédance

$\pi = 3,1415$

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 Z_c doit être 10 fois plus petite que 5 Kohms

On aura donc un condensateur au moins égal à :

$$C = 1 / (2 * \pi * 50\text{Hz} * 500\text{Ohms}) = 6,37 \text{ F} * 10^{-6} = 6,37 \mu\text{F}.$$

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.

19.12 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	0	0	b9	b8	b7	b6	b5	b4	b3	b2

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. Merci qui ? Merci Microchip.

Le choix de la méthode s'effectue à l'aide du bit 7 de **ADCON1**, registre dont je vais maintenant vous parler.

ATTENTION : Le registre ADRESH se situe en banque 0, alors que ADRESL se trouve en banque 1.

19.13 Le registre ADCON1

Je vais parler maintenant, pour des raisons de commodité, de ce registre. Il 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- Vss	Vref + Vdd	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

Avant de vous donner le tableau concernant le 16F876, et qui est le même sans les 3 pins AN7/AN5, je vais vous donner un petit mot d'explication sur l'interprétation de celui que je viens de vous donner.

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.

Je vous donne maintenant le tableau équivalent pour le 16F876 :

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

Ce tableau est identique à celui du 16F877, excepté la disparition du PORTE. Du fait de cette disparition, vous trouverez plusieurs configuration de PCFGx qui donneront le même résultat. Dans ce cas, choisissez celle que vous voulez.

Bien évidemment, la somme des chiffres de la dernière colonne donnera la valeur 5 pour chacune des lignes, et non 8, comme c'était le cas sur le 16F877.

19.14 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 »

Souvenez-vous que:

- 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 PICs 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 la 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 en 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 :

- 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.
- 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 $2T_{ad}$ avant que ne redémarre automatiquement l'acquisition suivante.

19.15 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.

19.16 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) Paramétrez 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 T_{acq} (typiquement $19,7\mu s$ 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 $2T_{ad}$ (typiquement $3,2\mu s$)
- 11) Recommencez au point 4

Notez que puisque l'acquisition redémarre automatiquement après le temps « $2 T_{ad}$ », vous pouvez relancer l'acquisition directement, à votre charge d'attendre non pas le temps T_{acq} pour la fin de l'acquisition, mais le temps $T_{acq} + 2T_{ad}$. Ceci vous épargne une temporisation. En effet, 2 temporisations qui se suivent peuvent être remplacées par une temporisation unique de temps cumulé.

Si donc, nous prenons notre PIC cadencée à 20Mhz, sous une tension d'alimentation de 5V, nous aurons :

- 1) Configurez ADCON1 en fonction des pins utilisées en mode analogique, ainsi que les registres TRISA et TRISE si nécessaire.
- 2) Validez, si souhaité, l'interruption du convertisseur (PEIE, ADIE, GIE)
- 3) Paramétrez le diviseur 32 sur ADCON0 (B'10000000')
- 4) Choisissez le canal en cours de digitalisation sur ADCON0 et lancez le convertisseur (B'10xxx001')
- 5) Attendez le temps ($T_{acq}+2T_{ad}$), soit $19,7\mu s + 3,2\mu s = 22,9\mu s$
- 6) Démarrez la conversion en positionnant le bit GO du registre ADCON0
- 7) Attendez la fin de la conversion
- 8) Lisez les registres ADRESH et si nécessaire ADRESL
- 9) Recommencez au point 4

Notez que vous pouvez, comme montré, réaliser plusieurs opérations en même temps.

Cependant, fort logiquement, vous ne pouvez pas positionner les bits ADON et GO/DONE en même temps, puisque le temps T_{acq} doit impérativement les séparer.

Remarque :

Lorsque vous disposez de beaucoup de temps entre 2 lectures de la valeur analogique, je vous conseille d'effectuer plusieurs mesures intermédiaires, et d'effectuer la moyenne de ces mesures. Ainsi, un parasite éventuel, ou une légère fluctuation de votre tension sera fortement atténuée.

Il est pratique dans ce cas d'effectuer un nombre de mesures qui est une puissance de 2 (2 mesures, 4, 8, 16...). Ainsi, pour effectuer votre moyenne, il suffira d'effectuer la somme de toutes les valeurs, la division s'effectuant par simple décalage.

Exemple : vous faites 4 mesures successives.

Le résultat moyen sera donc (somme des 4 valeurs) / 4, donc, au niveau programmation, somme des 4 valeurs, suivie de 2 décalages vers la droite (division par 4).

Seconde remarque, vous verrez, dans le chapitre concernant les modules CCP, que des mécanismes sont prévus pour vous faciliter la vie au niveau de l'automatisation des séquences. J'y reviendrai à ce moment, mais je vous conseille de suivre dans l'ordre, pour ne pas être submergé d'informations parfois assez lourdes à digérer.

19.17 Exercice pratique sur le convertisseur A/D

Oui, oui, je sais. Vous devez vous dire : « ce n'est pas trop tôt ». En effet, je vous ai un peu assommé de théorie durant toute cette partie. Cependant, ce n'est pas de l'information perdue, elle vous sera nécessaire si vous décidez d'utiliser le convertisseur à la limite de ses possibilités.

Certes, ce ne sera pas toujours le cas, mais qui peut le plus peut le moins, et, de toutes façons, je vous ai donné des valeurs « standard » qui vous permettent d'ignorer toute la partie calcul si cela ne vous est pas utile.

Vous allez donc voir que paradoxalement, la mise en application est plus simple que la théorie relative à la conversion. Mais assez de discours, commençons notre exercice par l'électronique associée.

Nous allons nous servir de notre petit chenillard « lum1 », et l'améliorer en y ajoutant 2 potentiomètres destinés à régler, d'une part le temps d'allumage de chaque LED, et d'autre part, le temps d'extinction entre 2 allumages successifs.

Le matériel nécessaire supplémentaire consistera donc en 2 potentiomètres de type « linéaire », et d'une valeur peu critique, comprise entre 1Kohms (recommandé), et 10 Kohms. Etant donné que nous avons effectué nos calculs de temps sur la base d'une alimentation de 5V et d'une résistance de source égale à 10Kohms, ces valeurs permettront une numérisation suffisamment rapide de notre tension d'entrée. L'achat ne vous ruinera pas, le prix d'un potentiomètre avoisinant 1 Euro. Vous pouvez même utiliser des résistances ajustables pour circuit imprimé, qui vous fera encore baisser le prix de façon significative.

Nous allons monter nos potentiomètres en mode « diviseur de tension », ce qui fait que la tension présente sur le curseur sera proportionnelle à la position du bouton sur son axe de rotation. En clair, plus vous tournerez le bouton du potentiomètre vers la droite, plus la tension sur le curseur augmentera.

Comme nous alimentons ces potentiomètres à partir de Vdd et de Vss, la tension minimale présente sur le curseur sera de Vss, tandis que la tension maximale sera de Vdd. Nous devons donc choisir un mode de fonctionnement comportant 2 pins configurées en entrées analogiques, et aucune tension de référence externe (2/x/0).

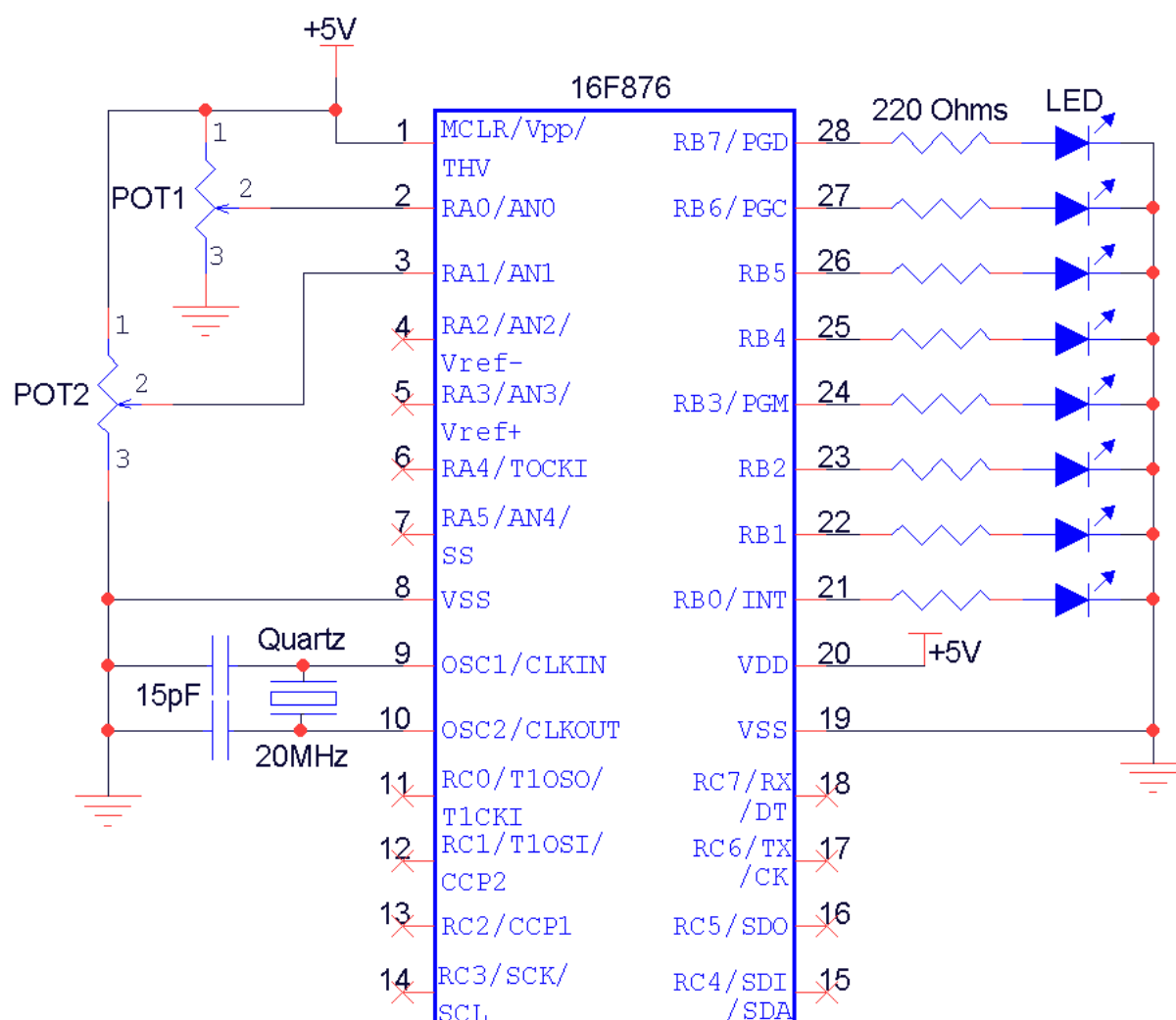
Ceci, en consultant notre tableau ADCON1, nous indique que cette possibilité n'existe pas. Par contre, nous avons la possibilité d'utiliser 3 entrées analogiques sans tension de

référence externe. Nous choisirons donc cette possibilité, quitte à ne pas utiliser un des canaux analogiques alors disponibles.

PCFG 3 à 0	AN4 RA5	AN3 RA3	AN2 RA2	AN1 RA1	AN0 RA0	Vref- Vss	Vref+ Vdd	A/D/R
0100	D	A	D	A	A	Vss	Vdd	3/2/0

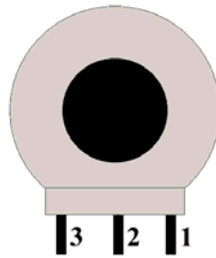
Nous devons de ce fait connecter nos 2 potentiomètres sur 2 des 3 pins suivantes : RA3, RA1, RA0. Pour rester logiques, nous choisissons RA0 pour le potentiomètre 1, et RA1 pour le potentiomètre 2.

Voici donc notre schéma (remarquez que nous avons été obligé de consulter les options possibles au niveau du logiciel avant de pouvoir le dessiner) :



Il me reste à vous donner le brochage des potentiomètres. En fait, si vous placez celui-ci bouton face à vous, et les 3 pins de connexion dirigées vers le bas, la pin 1 se trouve à votre droite, la 2 au centre, et la 3 à votre gauche. L'inversion des broches 1 et 3 inversera le sens

de fonctionnement du potentiomètre, l'inversion de la broche 2 avec une des 2 autres se traduira par.... un joli petit nuage de fumée.



Maintenant, nous allons calculer les temporisations. Le potentiomètre 1 va nous servir pour établir la durée d'allumage des LEDs de sortie, le potentiomètre 2 va régler la durée de l'extinction. Nous allons décider que la durée minimale d'éclairage sera de $1/10^{\text{ème}}$ de seconde, la durée maximale de 2,5S. La durée d'extinction variera dans le même registre.

Nous aurons donc un algorithme du type :

```
Allumage des LEDs
Attente suivant position potentiomètre 1 de 0,1 à 2,5s.
Extinction des LEDs
Attente suivant position du potentiomètre 2 de 0,1 à 2,5S
Allumage suivant : on recommence.
```

On va décider, pour des raisons de facilité, d'utiliser 256 valeurs intermédiaires pour nos potentiomètres. Ceci nous permet de coder la valeur sur 1 octet. Un pas de variation vaudra donc plus ou moins : $2500\text{ms} / 256 = 9,77 \text{ ms}$.

Notre timer 0 nous donne un temps de débordement, sans prédiviseur, de $0,2 * 256 = 51,2\mu\text{s}$.

Pour obtenir 9,77ms, nous avons besoin d'un prédiviseur de $9770 / 51,2 = 190,8$.

Cette valeur n'existe pas, mais comme notre application ne requiert pas de temps critiques, et que nous avons choisi ces valeurs « au pif », nous allons prendre un prédiviseur de 256.

De ceci, nous tirons :

- Le temps séparant 2 passages successifs dans notre routine d'interruption sera de : $0,2\mu\text{s} * 256 * 256 = 13,1072 \text{ ms}$
- Pour attendre la durée minimale de 100ms, nous devons passer $100 / 13,1072 =$ approximativement 8 fois dans notre routine d'interruption.
- Pour attendre la durée maximale de 2500ms, nous devons passer $2500 / 13,1072 = 190$ fois dans cette même routine.

Donc, notre potentiomètre devra faire varier notre compteur de passage entre 8 et 190 selon sa position. Notre potentiomètre devra donc pouvoir prendre $190 - 8 = 182$ valeurs, ce qui n'est pas pratique. Nous utiliserons donc 256 valeurs, pour rester dans les puissances de 2.

Nous pourrions donc faire varier le nombre de passages entre 8 (potentiomètre au minimum) et $(8 + 255)$, soit 263 (potentiomètre au maximum).

Ceci nous donne un temps minimum de 104 ms et un temps maximum de 3,45 secondes. Ceci est tout à fait valable pour notre petit chenillard. D'autant que nous pourrions régler la vitesse avec des pas de 10 ms.

Nous allons commencer par écrire notre pseudo-code sans nous préoccuper de la conversion analogique/numérique. Nous avons déjà étudié notre programme principal. Nous voyons que nous avons besoin d'une routine d'interruption.

Nous pouvons nous dire que pendant que nous attendons, nous n'avons rien d'autre à faire, inutile donc d'utiliser une interruption, une simple boucle fera l'affaire. Nous utiliserons cependant notre timer 0 pour faciliter le calcul de la durée.

On attend 8 fois le débordement du timer0 ($8 * 13,10\text{ms}$)
Tant que le compteur est supérieur à 0,
On attend 13,10ms supplémentaires
On décrémente le compteur

On doit alors intercaler la lecture des différents potentiomètres. Comme nous n'utilisons qu'un potentiomètre à la fois (on allume, on attend en fonction du potentiomètre 1, on éteint, on attend en fonction du potentiomètre 2), et que le temps d'attente minimum est de $8 * 13,10\text{ms}$, on remarque qu'on a largement le temps d'effectuer plusieurs lectures de chaque valeur.

En effet, pour lire un potentiomètre, il nous faut approximativement $20\mu\text{s}$ de temps d'acquisition, et à peu près le même temps ($19,2\mu\text{s}$) pour effectuer la conversion. Ceci nous prend donc de l'ordre de $40\mu\text{s}$, à comparer aux $13,10\text{ms}$ ($13100\mu\text{s}$) de chacun des débordements de notre timer 0.

Comme nous avons au minimum 8 temps d'attente, pourquoi donc ne pas en profiter pour réaliser 8 lectures du potentiomètre concerné ? Notre routine de temporisation devient donc :

Pour chacun des 8 premiers passages

On lance une conversion A/D

On attend que le timer 0 déborde ($13,10\text{ms}$)

Passage suivant

On calcule la moyenne des 8 conversions effectuées

Tant que le résultat est > 0

On attend que le timer 0 déborde

On décrémente le résultat

Il nous faut maintenant écrire la routine de conversion A/D. Pour bien appliquer ce que nous avons appris, nous allons effectuer les opérations suivantes :

- On choisit le canal à numériser (canal 0 = potentiomètre 1, canal1 = pot2)
- On lance l'acquisition (charge du condensateur) en mettant ON le convertisseur
- On attend 20 μ s
- On lance la conversion en mettant GO à 1 dans ADCON0

Bien entendu, nous avons besoin de mesurer nos 20 μ s. Ceci correspond à 100 cycles d'instructions. Nous pouvons donc utiliser notre timer 2, sans diviseur, et avec une valeur de 100-1 dans PR2. Il nous suffira donc d'attendre le positionnement du flag TRM2IF dans le registre PIR1.

Le temps d'attente se réalisera donc comme suit, PR2 étant initialisé à D'99' dans la routine d'initialisation.

- On efface TMR2
- On efface TMR2IF
- On lance le timer 2
- On attend le positionnement de TMR2IF
- On arrête le timer 2

Ne reste donc plus qu'à sauvegarder notre résultat (sur 10 bits) dans 2 variables. Ceci pourra se faire, par exemple, dans notre routine d'interruption AD, qui sera déclenchée automatiquement à la fin de la conversion. Cette routine se bornera à écrire les 2 registres du résultat dans les variables pointées.

Donc, pour résumer, voici comment les événements vont se succéder chronologiquement pour chacune des 8 premières durées de 13,10ms.

- On met en service le convertisseur
- On attend 20 μ s
- On lance la conversion
- 19,2 μ s plus tard, l'interruption A/D est générée
- On sauve le résultat et on arrête le convertisseur
- On attend le temps qui reste de nos 13,10ms du départ
- On recommence

L'arrêt du convertisseur et du timer 2 ne sont pas obligatoires, mais diminuent la consommation(et donc l'échauffement) du PIC. C'est donc une bonne habitude de procéder de la sorte.

Donc, le temps qui sépare 2 conversions est d'approximativement 13,10ms – 20 μ s – 19,2 μ s, soit bien plus que les 2Tad nécessaires (3,2 μ s). Donc, aucun soucis de ce côté.

Voyons maintenant la réalisation pratique de notre programme. Commencez par copier le fichier « lum1.asm », et renommez la copie en « lum2.asm ». Editez l'en-tête du programme :

```

;*****
;   Réalisation d'un mini-chenillard à 8 LEDs avec 2 potentiomètres de
;   réglage.
;
;*****
;
;   NOM: Lum2
;   Date: 31/05/2002
;   Version: 1.0
;   Circuit: Circuit maquette
;   Auteur: Bigonoff
;
;*****
;
;   Fichier requis: P16F876.inc
;                   lumdat.inc
;
;
;*****
;
;   Notes: les 8 sorties sont sur le PORTB. Un niveau haut allume la LED
;           correspondante.
;           Exercice sur les conversions A/D, à l'aide de 2 potentiomètres
;           Le potentiomètre 1 est sur RA0
;           Le potentiomètre 2 est sur RA1
;           Le potentiomètre 1 règle le temps d'allumage
;           Le potentiomètre 2 règle le temps d'extinction
;           La fréquence du quartz est de 20Mhz
;
;*****

```

Ensuite, la configuration, qui ne pose aucun problème :

```

LIST      p=16F876      ; Définition de processeur
#include <p16F876.inc>    ; fichier include
#include <lumdat.inc>     ; données d'allumage

__CONFIG  _CP_OFF & _DEBUG_OFF & _WRT_ENABLE_OFF & _CPD_OFF & _LVP_OFF &
_BODEN_OFF & _PWRTE_ON & _WDT_ON & _HS_OSC
; _CP_OFF          Pas de protection
; _DEBUG_OFF       RB6 et RB7 en utilisation normale
; _WRT_ENABLE_OFF  Le programme ne peut pas écrire dans la flash
; _CPD_OFF         Mémoire EEprom déprotégée
; _LVP_OFF         RB3 en utilisation normale
; _BODEN_OFF       Reset tension hors service
; _PWRTE_ON        Démarrage temporisé
; _WDT_ON          Watchdog en service
; _HS_OSC          Oscillateur haute vitesse (20Mhz)

```

On trouve ensuite la valeur pour le registre OPTION

```

;*****
;
;           ASSIGNATIONS SYSTEME
;*****

; REGISTRE OPTION_REG (configuration)
; -----
OPTIONVAL EQUB'10000111' ; RBPu b7 : 1= Résistance rappel +5V hors service
                        ; PSA  b3 : 0= Assignation prédiviseur sur Tmr0

```

```
; PS2/PS0 b2/b0 valeur du prédiviseur = 256
```

Nous n'aurons qu'une seule source d'interruption, le convertisseur A/D, qui est une interruption périphérique. Ceci nous impose donc de programmer INTCON et PIE1 :

```
; REGISTRE INTCON (contrôle interruptions standard)
; -----
INTCONVAL EQU B'01000000' ; PEIE b6 : masque autorisation générale périphériques

; REGISTRE PIE1 (contrôle interruptions périphériques)
; -----
PIE1VAL EQU B'01000000' ; ADIE b6 : masque interrupt convertisseur A/D
```

Comme il n'existe pas de mode « 2 canaux analogiques sans tension de référence externe », nous utiliserons le mode « 3 canaux ». Nous choisissons une justification à droite, pour conserver, avant calcul de la moyenne, l'intégralité des 10 bits de résultat. Ce n'est qu'après avoir calculé la moyenne, que nous ne conserverons que les 8 bits les plus significatifs.

```
; REGISTRE ADCON1 (ANALOGIQUE/DIGITAL)
; -----
ADCON1VAL EQU B'10000100' ; 3 Entrées analogiques, résultat justifié
; à droite
```

Le PORTA sera configuré en entrée, le PORTB en sortie. Nous aurions pu nous abstenir de configurer TRISA, celui-ci étant positionné en entrée au moment de la mise sous tension. Cependant, je laisse cette configuration pour vous rappeler que TRISA intervient dans l'initialisation du convertisseur.

```
; DIRECTION DES PORTS I/O
; -----

DIRPORTA EQU B'00111111' ; Direction PORTA (1=entrée)
DIRPORTB EQU B'00000000' ; Direction PORTB
```

Les macros ne posent aucun problème. J'en profite pour rappeler que les macros non utilisées ne consomment aucune place dans la mémoire du PIC, puisqu'elles ne sont tout simplement pas traduites.

```
*****
;
; MACRO
; *****

; Changement de banques
; -----

BANK0 macro ; passer en banque0;
    bcf STATUS,RP0
    bcf STATUS,RP1
endm

BANK1 macro ; passer en banque1
    bsf STATUS,RP0
    bcf STATUS,RP1
endm
```



```

BANK2 macro                ; passer en banque2
    bcf    STATUS,RP0
    bsf    STATUS,RP1
endm

BANK3 macro                ; passer en banque3
    bsf    STATUS,RP0
    bsf    STATUS,RP1
endm

```

Nous arrivons à notre zone de variables. Nous avons besoin d'une variable sur 8 bits pour compter les boucles de notre routine de temporisation. Les 8 lectures successives du potentiomètre concerné nécessitera 16 emplacements (8 * 2 octets).

Pour pouvoir calculer la moyenne de ces valeurs, nous devrons en faire l'addition, ceci nécessitera donc une variable de 2 octets.

Ne reste plus que le flag qui va nous informer si nous sommes en train de nous occuper de notre potentiomètre 1 ou de notre potentiomètre 2. Tout ceci nous donne :

```

;*****
;                               VARIABLES BANQUE 0                               *
;*****

; Zone de 80 bytes
; -----

    CBLOCK 0x20                ; Début de la zone (0x20 à 0x6F)
    cmpt : 1                   ; compteur de boucles
    flags : 1                  ; flags divers
                                ; b0 : 0 = potar1, 1=potar2

    potar : 16                 ; valeurs du potentiomètre lu(msb,lsb)
    result : 2                 ; résultat de la somme des valeurs
    ENDC                      ; Fin de la zone

#define numpotar flags,0 ; flag de temporisation

```

Notre zone commune se borne aux registres à sauvegarder :

```

*****
;                               VARIABLES ZONE COMMUNE                               *
;*****

; Zone de 16 bytes
; -----

    CBLOCK 0x70                ; Début de la zone (0x70 à 0x7F)
    w_temp : 1                 ; Sauvegarde registre W
    status_temp : 1            ; sauvegarde registre STATUS
    ENDC

```

Le démarrage du programme, comme toujours en adresse 0x00 :

```

; //////////////////////////////////////
;                               I N T E R R U P T I O N S                               *
; //////////////////////////////////////

```

```

;*****
;
;                               DEMARRAGE SUR RESET                               *
;*****

    org 0x000      ; Adresse de départ après reset
    goto init      ; Initialiser

```

Puis notre routine d'interruption, qui ne contient que l'interruption du convertisseur. Le test de l'interruption est donc inutile :

```

;*****
;
;                               ROUTINE INTERRUPTION                               *
;*****

    ;sauvegarder registres
    ;-----
    org 0x004      ; adresse d'interruption
    movwf w_temp    ; sauver registre W
    swapf STATUS,w  ; swap status avec résultat dans w
    movwf status_temp ; sauver status swappé
    BANK0           ; passer en banque0

    ; Interruption AD
    ; -----
    bcf ADCON0,ADON ; éteindre convertisseur
    movf ADRESH,w   ; charger poids fort conversion
    movwf INDF      ; sauver dans zone de sauvegarde
    incf FSR,f      ; pointer sur poids faible
    bsf STATUS,RP0  ; passer banque 1
    movf ADRESL,w   ; charger poids faible conversion
    bcf STATUS,RP0  ; passer banque 0
    movwf INDF      ; sauver dans zone de sauvegarde
    incf FSR,f      ; pointer sur poids fort suivant
    bcf PIR1,ADIF   ; effacer flag interrupt

    ;restaurer registres
    ;-----
restorereg
    swapf status_temp,w ; swap ancien status, résultat dans w
    movwf STATUS        ; restaurer status
    swapf w_temp,f      ; Inversion L et H de l'ancien W
                        ; sans modifier Z
    swapf w_temp,w      ; Réinversion de L et H dans W
                        ; W restauré sans modifier status
    retfie              ; return from interrupt

```

Une petite remarque, à ce sujet. Nous utilisons FSR dans le programme principal et dans la routine d'interruption, pourtant nous ne le sauvons pas. Ceci est dû au fait que la modification de FSR effectuée dans la routine d'interruption est utilisée dans le programme principal.

Ceci est possible uniquement parce que la routine d'interruption intervient à un endroit connu de notre programme, et n'est donc pas réellement asynchrone. C'est le signe que le traitement de ce morceau de code pouvait être effectué dans le programme principal, une routine d'interruption n'était donc pas indispensable.

Néanmoins, j'ai utilisé une interruption à dessin pour montrer son utilisation de façon didactique. Je rappelle qu'il n'est nullement question d'optimisation ici, bien qu'on pouvait

simplifier facilement le programme, mais plutôt de vous montrer la façon générale de procéder. Vous aurez plus facile supprimer ce qui est inutile que d'ajouter ce que je ne vous aurai pas montré.

La routine d'initialisation est semblable à celle de «lum1 », à ceci près qu'il a fallu initialiser le timer 2 et son registre PR2.

```
; ////////////////////////////////////////
;                                     P R O G R A M M E
; ////////////////////////////////////////

;*****
;                                     INITIALISATIONS
;*****
init
    ; initialisation PORTS (banque 0 et 1)
    ; -----
    BANK0                ; sélectionner banque0
    clrf PORTA            ; Sorties PORTA à 0
    clrf PORTB            ; sorties PORTB à 0
    bsf STATUS,RP0       ; passer en banque1
    movlw ADCON1VAL       ; PORTA en mode digital/analogique
    movwf ADCON1          ; écriture dans contrôle A/D
    movlw DIRPORTA        ; Direction PORTA
    movwf TRISA           ; écriture dans registre direction
    movlw DIRPORTB        ; Direction PORTB
    movwf TRISB           ; écriture dans registre direction

    ; Registre d'options (banque 1)
    ; -----
    movlw OPTIONVAL       ; charger masque
    movwf OPTION_REG      ; initialiser registre option

    ; registres interruptions (banque 1)
    ; -----
    movlw INTCONVAL       ; charger valeur registre interruption
    movwf INTCON          ; initialiser interruptions
    movlw PIE1VAL         ; Initialiser registre
    movwf PIE1            ; interruptions périphériques 1

    ; timer 2 (banque 1)
    ; -----
    movlw D'99'           ; 99 + 1 passages = 20µs
    movwf PR2             ; dans comparateur

    ; initialiser variables
    ; -----
    BANK2                ; passer en banque 2
    movlw low mesdata     ; adresse basse des data
    movwf EEADR           ; dans pointeur bas FLASH
    movlw high mesdata    ; adresse haute des data
    movwf EEADRH          ; dans pointeur haut FLASH
    bcf STATUS,RP1       ; repasser en banque 0
    movlw potar           ; adresse de la zone de stockage
    movwf FSR             ; dans pointeur
    clrf T2CON            ; pré = postdiviseur = 1

    ; autoriser interruptions (banque 0)
    ; -----
    clrf PIR1             ; effacer flags 1
```

```

bsf    INTCON,GIE    ; valider interruptions
goto   start         ; programme principal

```

Rien de changé pour la routine de lecture d'un octet en mémoire programme :

```

;*****
;                               LIRE UN OCTET EN FLASH                               *
;*****
;-----
; Lit un octet en mémoire programme, puis pointe sur le suivant
; retourne l'octet lu dans w
; si le bit 8 du mot lu vaut 1, on pointe sur la première donnée
;-----
readflash
    BANK3                ; pointer sur banque3
    bsf    EECON1,EEPGD   ; accès à la mémoire programme
    bsf    EECON1,RD      ; lecture
    nop                     ; 2 cycles d'attente
    nop
    bcf    STATUS,RP0     ; pointer sur banque2
    incf   EEADR,f        ; incrémenter pointeur bas sur données
    btfsc  STATUS,Z       ; tester si débordé
    incf   EEADRH,f       ; oui, incrémenter pointeur haut
    btfss  EEDATH,0       ; tester bit 0 data haute = bit 8 donnée
    goto   readfsuite     ; si 0, sauter instructions suivantes
    movlw  low mesdata    ; si 1, adresse basse des data
    movwf  EEADR          ; dans pointeur bas FLASH
    movlw  high mesdata   ; adresse haute des data
    movwf  EEADRH         ; dans pointeur haut FLASH
readfsuite
    movf   EEDATA,w       ; charger 8 bits donnée
    bcf    STATUS,RP1     ; repointer sur banque0
    return                ; et retour

```

La routine de temporisation est conforme à ce que nous avons défini, avec dans l'ordre, les 8 premiers passages, avec lancement de la numérisation

```

;*****
;                               Temporisation                               *
;*****
;-----
; Base de temps de 13,1072ms.
; On attend d'office 8 * 13,10 ms. Durant cette attente, on lance 8
; numérisations du potentiomètre concerné.
; Ensuite, on calcule la moyenne des 8 conversions, et on garde le résultat
; sur 8 bits. Cette valeur indique le nombre de multiples de 13,10ms
; supplémentaires à attendre.
;-----
tempo
    ; gestion des 8 premiers passages
    ; -----
    movlw  0x08           ; pour 8 * 13,10ms
    movwf  cmpt           ; dans compteur
tempol
    call   acquire        ; lancer acquisition
    call   wait           ; attendre 13,1072ms
    decfsz cmpt,f         ; décrémenter compteur de boucles
    goto   tempol         ; boucle suivante

```

Ensuite, la somme des 8 valeurs obtenues :

```

; calcul de la somme des valeurs
; -----
movlw 0x08      ; pour 8 boucles
movwf cmpt      ; dans compteur de boucles
clrf result     ; effacer résultat
clrf result+1   ; idem pour poids faible
tempo2
decf FSR,f      ; pointer sur poids faible
movf INDF,w     ; charger poids faible
addwf result+1,f ; ajouter au poids faible résultat
btfsc STATUS,C  ; tester si débordement
incf result,f   ; oui, poids fort +1
decf FSR,f      ; pointer sur poids fort
movf INDF,w     ; charger poids fort
addwf result,f  ; ajouter au poids fort résultat
decfsz cmpt,f   ; décrémenter compteur de boucles
goto tempo2     ; pas dernière, suivante

```

Remarquez l'astuce utilisée : la routine d'interruption du convertisseur incrémente FSR depuis le premier octet de poids fort des valeurs numérisées, jusqu'à l'octet qui suit le dernier octet de poids faible. Il suffit donc de décrémenter successivement FSR pour additionner toutes les valeurs en commençant par le dernier poids faible.

La méthode est celle de l'addition classique sur 2 octets : on additionne les 2 poids faibles, s'il y a débordement, on incrémente le poids fort. On additionne ensuite les 2 poids forts. Comme nous avons 8 valeurs à additionner, et que chaque valeur ne comporte que 10 bits valides, le résultat tiendra sur 13 bits. Donc, il n'y aura pas débordement en additionnant les poids forts.

Nous procédons ensuite au calcul de la moyenne. Comme nous voulons un résultat sur 8 bits, et que nous en avons 13, on pourrait se dire qu'on doit décaler le résultat 5 fois vers la droite. Le résultat final se trouvant dans le poids faible du résultat.

En fait, il est plus simple de tout décaler de 3 rangs vers la gauche, le résultat se trouvant alors dans le poids fort du résultat. Faites le test sur papier pour vous en convaincre.

```

; calcul de la moyenne sur 8 bits
; -----
rlf result+1,f  ; décaler poids faible vers la gauche
rlf result,f    ; idem poids fort avec b7 poids faible
rlf result+1,f  ; décaler poids faible vers la gauche
rlf result,f    ; idem poids fort avec b7 poids faible
rlf result+1,f  ; décaler poids faible vers la gauche
rlf result,f    ; idem poids fort avec b7 poids faible
; on avait 5 + 8 bits, on a 8 + 5 bits

```

De nouveau, la méthode est classique : on décale le poids faible vers la gauche, le bit 7 tombe dans le carry. On décale ensuite le poids fort, le carry devient le bit0. On a donc décalé les 16 bits vers la gauche. Notez que nous aurions dû mettre le carry à 0 avant le décalage du poids faible, mais ceci est inutile ici, car nous n'utiliserons pas le poids faible du résultat, seuls comptent les 8 bits finaux du poids fort. Donc peut importe si nous avons fait entrer un « 1 » indésirable dans le poids faible.

Il reste à attendre (result * 13,10ms). Nous devons tester result avant la boucle, de façon à ce qu'une valeur de 0 ne provoque pas 256 boucles, ce qui aurait été le cas en utilisant une boucle de type « decfsz ».

```

; attendre result * 13,10ms
; -----
tempo3
    movf    result,f        ; tester résultat
    btfsc   STATUS,Z        ; tester si 0
    return  ; oui, fin de l'attente
    call    wait            ; attendre
    decfsz  result,f        ; décrémenter durée restante
    goto    tempo3         ; boucle suivante

```

Il faut écrire maintenant la petite sous-routine d'attente du débordement du timer 0 :

```

;*****
;                               Attendre fin timer 0                               *
;*****
wait
    clrf    TMR0            ; effacer timer0
    bcf     INTCON,T0IF     ; effacer flag débordement
waitl
    btfss   INTCON,T0IF     ; tester si timer a débordé
    goto    waitl          ; non, attendre
    return  ; fin d'attente

```

Passons maintenant à la routine de démarrage de la numérisation. Cette dernière, comme prévu, se compose de 3 parties : Lancement de l'acquisition, attente du temps Tacq, démarrage du convertisseur :

```

;*****
;                               Acquisition de la valeur analogique                               *
;*****
;-----
; Si numpotar vaut 0, on travaille sur le potar1, sinon, c'est le potar2
; On sélectionne le canal, et on lance l'acquisition
; on attend 20 µs (Tacq), puis on lance la numérisation
; la fin de numérisation sera détectée par interruption
;-----
acquire
    ; lancer l'acquisition
    ; -----
    movlw   B'10000001'     ; diviseur 32, canal 0, convertisseur ON
    btfsc   numpotar        ; tester si c'est potentiomètre 2
    movlw   B'10001001'     ; oui, alors canal 1
    movwf   ADCON0          ; paramétrer convertisseur, lancer acquisition

    ; attendre 20 µs
    ; -----
    clrf    TMR2            ; effacer timer2
    bcf     PIR1,TMR2IF     ; effacer flag débordement
    bsf     T2CON,TMR2ON    ; lancer timer 2
acquirel
    btfss   PIR1,TMR2IF     ; tester si temps écoulé
    goto    acquirel        ; non, attendre
    bcf     T2CON,TMR2ON    ; oui, arrêt du timer 2

```

```

; démarrage du convertisseur
; -----
bsf    ADCON0,GO      ; lancer conversion A/D
return                               ; fin de l'acquisition

```

Le programme principal est tout simple :

```

;*****
;                               PROGRAMME PRINCIPAL                               *
;*****

start
    clrwdt          ; effacer watch dog
    call readflash  ; lire un octet en flash
    movwf PORTB     ; le mettre sur le PORTB (allumage LEDs)
    bcfnumpotar     ; pour potentiomètre 1
    call tempo      ; attendre en fonction du potentiomètre
    clrf PORTB      ; éteindre LEDs
    bsfnumpotar     ; pour potentiomètre 2
    call tempo      ; attendre en fonction du potentiomètre
    goto start      ; boucler
mesdata             ; emplacement des données (lumdat.inc)
END                 ; directive fin de programme

```

Lancez l'assemblage, chargez le programme, et lancez l'alimentation. Votre chenillard est maintenant réglable avec les 2 potentiomètres.

Tiens, le vôtre reste obstinément bloqué sur la première LED ? Que se passe-t-il donc ?

J'attribue 10 points avec mention du jury à ceux qui ont déjà compris, les autres, je vous conseille de réfléchir un peu avant de poursuivre la lecture.

Pour rechercher la cause, passons dans MPLAB en mode simulation en pas à pas. Après l'assemblage, pressez <F6>, puis des pressions successives de <F7> jusqu'à la ligne :

```
call readflash ; lire un octet en flash
```

du programme principal. Continuez ensuite les pressions successives de <F7> jusqu'à ce que vous arriviez à la ligne suivant l'étiquette « tempo » de votre routine de temporisation. Pressez alors successivement <F7> jusque la ligne :

```
call acquire ; lancer acquisition
```

Pressez alors sur <F8>. Ceci vous permet d'exécuter la sous-routine en une seule fois, sans devoir entrer manuellement à l'intérieur. Vous êtes maintenant sur la ligne :

```
callwait ; attendre 13,1072ms
```

Pressez de nouveau <F8>. La barre inférieure passe en jaune, preuve que le simulateur travaille. En effet, cette routine dure très longtemps du point de vue du PIC. En effet, 13,10ms représentent plus de 65000 instructions à exécuter.

Au bout d'un moment, votre programme devrait se retrouver à la ligne suivante, à savoir :

```
decfsz cmpt,f ; décrémenteur compteur de boucles
```

En fait, vous vous retrouvez de façon incompréhensible à :

```
org 0x000 ; Adresse de départ après reset  
goto init ; Initialiser
```

Pour ceux qui disent « mais bon sang, c'est normal », je donne 8 points. Pour les autres, je donne un indice pour 5 points :

L'adresse 0x00 est l'adresse de reset, notre PIC a donc effectué un reset durant notre routine de temporisation.

Vous avez trouvé ? Sinon, reste à savoir quel reset, parmi les types possibles. En procédant par élimination, on trouve assez facilement qu'il s'agit d'un reset provoqué par le watchdog. En effet, on va attendre au minimum $8 * 13,10\text{ms}$ dans la routine de temporisation, alors que le temps minimal de reset par watchdog sans prédiviseur se situe sous cette valeur.

Donc, ne tombez pas dans le piège, pensez que vous devez intercaler des instructions « clrwdt » avant que ne soit provoqué un reset par le mécanisme du watchdog. Le meilleur emplacement pour le faire est bien entendu dans la boucle d'attente du débordement du timer 0, dans la sous-routine « wait ».

```
wait  
    clrf TMR0 ; effacer timer0  
    bcf INTCON,T0IF ; effacer flag débordement  
wait1  
    clrwdt ; effacer watchdog  
    btfss INTCON,T0IF ; tester si timer a débordé  
    goto wait1 ; non, attendre  
    return ; fin d'attente
```

Vous pouvez maintenant relancer l'assemblage et reprogrammer votre PIC. Notez alors que le potentiomètre 1 vous permet de régler le temps durant lequel les LEDs restent allumées, alors que le potentiomètre 2 permet de régler le temps durant lequel elles restent éteintes.

19.18 Conclusion

Nous en avons maintenant terminé avec notre convertisseur A/D. Tout ceci a du vous paraître compliqué et laborieux, mais vous avez vu dans notre application pratique qu'on pouvait en général se passer des calculs.

Ces calculs vous seront par contre utiles pour les applications qui nécessitent d'exploiter le convertisseur au maximum de ses possibilités, ce qui rendait impératif les explications théoriques.

Grâce à ce convertisseur, vous allez pouvoir mesurer des résistances, des tensions continues ou alternatives, des températures, et toute autre grandeur analogique. Ceci ouvre donc les portes de l'analogique à votre composant numérique.

Notes : ...

Notes : ...

20. Les modules CCP1 et CCP2

20.1 Généralités

Les 16F87x disposent de 2 modules **CCP**. CCP signifie **C**apture, **C**ompare, and **P**WM. Ceci vous indique déjà que nous pourrions diviser ce chapitre entre **3 parties distinctes**, correspondant à autant de modes de fonctionnement.

Ces modules CCP sont fortement liés et dépendant des timers 1 et 2, aussi j'aurais pu placer ce chapitre directement après l'étude des timers. Cependant, ils sont également liés au convertisseur A/D, et, de plus, j'aurai besoin de ce dernier pour pouvoir vous proposer un exercice pratique en fin d'étude théorique. Ceci explique pourquoi j'en parle maintenant.

Cependant, sachez déjà que ces modules augmentent les capacités des timers, et donc pondèrent la conclusion les concernant quand à leurs utilisations classiques

Il faut savoir que **les 2 modules CCP1 et CCP2 sont strictement identiques**, excepté la possibilité, pour le module CCP2, de démarrer automatiquement la conversion A/D. J'en reparlerai.

20.2 Ressources utilisées et interactions

Au niveau ressources utilisées, nous pouvons simplement dire que **les modules CCPx utilisés en mode compare et en mode capture font appel au timer 1, alors que le mode PWM nécessite l'utilisation du timer 2.**

Vous comprenez déjà que vous allez vous heurter à **2 types de contraintes** :

- D'une part, **l'utilisation des timers dans les modes étudiés précédemment et d'un module CCPx va être soumise à des contraintes inhérentes aux registres utilisés.** Vous comprenez en effet qu'il va être impossible, par exemple, de charger TMR1L et TMR1H simultanément avec une valeur qui vous arrange pour l'utilisation en mode timer, et une autre pour l'utilisation du CCP1 en mode compare.
- D'autre part, **l'utilisation de 2 modules CCP simultanément va entraîner également des contraintes,** dans la mesure où ils devront utiliser **les mêmes ressources.** Tout sera donc question d'étude et de compromis.

Nous allons principalement nous intéresser aux interactions lors de l'utilisation de 2 modules simultanément. Les contraintes liées à l'utilisation classique des timers seront abordées dans chaque cas particulier.

Puisque chacun des modules dispose de 3 modes de fonctionnement, nous aurons 9 possibilités d'interaction. Mais, comme les 2 modules sont en fait identiques, **les interactions se résument à 6 cas possibles.** En effet, les contraintes liées par exemple à CCP1 en mode capture et CCP2 en mode compare sont strictement identiques aux contraintes pour CCP1 en mode compare et CCP2 en mode capture.

Pour notre tableau des contraintes, « CCPx » concerne CCP1 ou CCP2 au choix, alors que CCPy concerne forcément l'autre CCP.

Mode de CCPx	Mode de CCPy	Interaction
Capture	Capture	Les modules doivent utiliser la même base de temps du timer 1
Capture	Compare	Si le module compare est utilisé en mode trigger, le reset peut perturber les mesures du module capture.
Capture	PWM	Aucune interaction, les modules utilisent un timer différent.
Compare	Compare	En mode trigger, le premier reset survenu empêche l'autre comparateur d'atteindre sa valeur
Compare	PWM	Aucune interaction, les modules utilisent un timer différent.
PWM	PWM	La fréquence sera identique, ainsi que les mises à jour via l'interruption TMR2

En somme, rien que du logique, ne vous tracassez pas pour les termes que vous ne comprenez pas, nous allons parler de tout ça en détail.

20.3 Les registres CCP1CON et CCP2CON

Tout d'abord, ne confondez pas, ces registres ont la même fonction, simplement CCP1CON concerne le module CCP1, tandis que CCP2CON concerne le module CCP2.

Ce registre CCPxCON permet donc, en toute logique, de déterminer le mode de fonctionnement du module. Voici son contenu, « x » remplace « 1 » ou « 2 » suivant le module utilisé dans tout le reste du chapitre.

CCPxCON

b7 :	Inutilisé	: Lu comme « 0 »
b6 :	Inutilisé	: Lu comme « 0 »
b5 :	CCPxX	: module Capture Compare and Pwm x bit X
b4 :	CCPxY	: module Capture Compare and Pwm x bit Y
b3 :	CCPxM3	: module Capture Compare and Pwm x Mode select bit 3
b2 :	CCPxM2	: module Capture Compare and Pwm x Mode select bit 2
b1 :	CCPxM1	: module Capture Compare and Pwm x Mode select bit 1
b0 :	CCPxM0	: module Capture Compare and Pwm x Mode select bit 0

Tout d'abord, voyons les bits CCPxX et CCPxY. Ces bits sont en fait les 2 bits de poids faible qui complètent le nombre de 10 bits utilisé pour le mode de fonctionnement PWM. J'en parlerai donc au moment de l'étude de ce mode de fonctionnement. Dans les autres modes, ces bits sont donc inutilisés.

Les bits CCPxM3 à CCPxM0 servent à déterminer quel sera le mode de fonctionnement du module concerné. Les possibilités sont les suivantes :

CCPM	Fonctionnement
0000	Module CCPx à l'arrêt
0100	Mode capture validé sur chaque flanc descendant
0101	Mode capture validé sur chaque flanc montant
0110	Mode capture validé sur chaque multiple de 4 flancs montants
0111	Mode capture validé sur chaque multiple de 16 flancs montants
1000	Mode compare, place la sortie à 1 sur débordement (+ bit CCPxIF = 1)
1001	Mode compare, place la sortie à 0 sur débordement (+ bit CCPxIF = 1)
1010	Mode compare, positionne CCPxIF sans affecter la sortie
1011	Mode compare, positionne CCPxIF sans affecter la sortie, et génère le trigger
11xx	Mode PWM

Au niveau du mode compare générant le trigger, il faut distinguer l'action du module CCP1 de celle du module CCP2 :

- Pour CCP1, l'événement « trigger » remet TMR1 à 0 (reset)
- Pour CCP2, l'événement « trigger » remet TMR1 à 0 (reset) et lance automatiquement la conversion A/D (si le module A/D est en service).

Notez qu'un reset provoque l'arrêt des modules CCP, le contenu du prédiviseur est de plus remis à 0.

N'oubliez pas que pour pouvoir utiliser les modules CCP, il faut que le timer utilisé soit correctement configuré, et, bien entendu, mis en service.

20.4 Le mode « capture »

Nous allons commencer par étudier le plus simple des 3 modes, à savoir le mode capture.

La première chose à remarquer est que ce mode fait intervenir une pin comme événement déclencheur. Il s'agit donc d'une entrée.

Il est donc impératif de configurer la pin CCPx en entrée via le registre TRISC avant de pouvoir utiliser le module CCPx en mode « capture ». Comme c'est le cas par défaut après une mise sous tension, on a cependant peu de chance de l'oublier.

20.4.1 Principe de fonctionnement

Le mode capture est simple à comprendre. Il est en étroite liaison avec les pins RC1/CCP2 et RC2/CCP1 du PIC. Attention à l'inversion des chiffres, la logique ne coule pas de source.

En fait, le principe est le suivant :

- Au moment de l'apparition de l'événement déclencheur sur la pin concernée, la valeur (16 bits) du timer 1 contenue dans les registres TMR1H et TMR1L est copiée dans les registres CCPR1H et CCPR1L. (Ah, tiens, voici un moyen d'annuler le bug concernant la lecture au vol de TMR1).

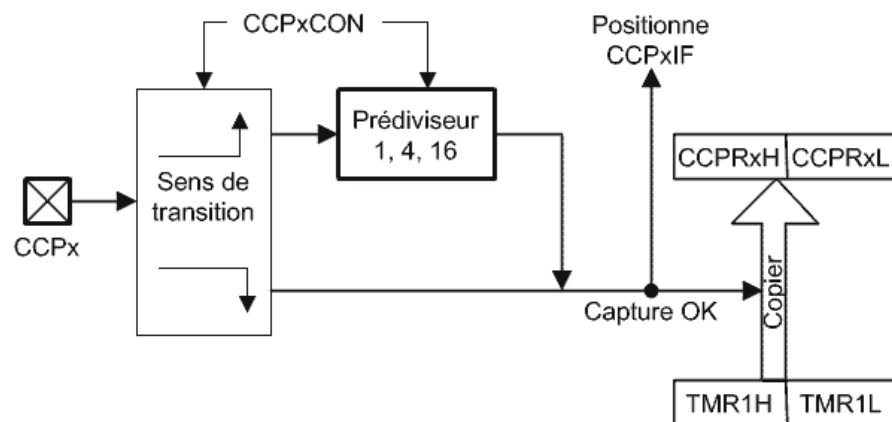
- Simultanément, le bit CCP1IF du registre PIR1 est validé, et une interruption intervient si elle est configurée.

L'événement déclencheur est une variation du signal sur la pin CCP1/RC2 pour le module CCP1, et sur la pin CCP2/RC1 pour le module CCP2. **L'événement qui provoque la capture dépend des bits CCPxM3 à CCPxM0.**

Vous avez plusieurs possibilités :

- Soit, la capture s'effectue chaque fois que la tension sur la pin CCPx passe de Vdd à Vss (CCPM = 0100)
- Soit la capture s'effectue chaque fois que la tension sur la pin CCPx passe de Vss à Vdd (CCPM = 0101)
- Ou alors, la capture s'effectue au bout de 4 transitions niveau bas/niveau haut de la pin CCPx, on comprend qu'il s'agit d'un prédiviseur de signal par 4 (CCPM = 0110)
- Ou enfin, la capture s'effectue de façon identique, mais après chaque multiple de 16 transitions niveau bas/niveau haut, il s'agit donc d'un prédiviseur de signal par 16 (CCPM = 0111).

Si vous avez bien compris ce qui précède, vous pouvez vous imaginer le schéma-bloc correspondant. Je vous le donne cependant explicitement, remplacez les termes « x » par « 1 » ou « 2 » en fonction du numéro de module utilisé



Vous constatez, à propos des contraintes, que vous avez bien un flag CCPxIF par module, une pin CCPx, un registre de configuration CCPxCON, et un registre 16 bits de sauvegarde de la valeur capturée. Par contre, vous n'avez qu'un seul timer, à savoir TMR1, utilisé pour les 2 modules.

Remarquez que le prédiviseur ne s'applique que pour la détection des signaux à flancs montants.

20.4.2 Champs d'application

Vous voyez probablement déjà plusieurs applications typiques de ce mode de fonctionnement. Sachant que **le timer 1 peut fonctionner en mode timer ou en mode compteur**, vous pouvez :

- Déterminer le temps séparant 2 ou plusieurs événements. Ceci vous permet par exemple de réaliser un chronomètre de haute précision. Si le temps est supérieur au temps de comptage sur 16 bits de TMR1, celui-ci peut générer une interruption, et donc incrémenter une variable qui comptera les multiples de 65536 événements.
- Compter un nombre d'événements comptés par le timer 1 survenus entre 2 ou plusieurs flancs présentés sur la pin CCPx.

Je vous donne un pseudo-code pour la réalisation d'un chronomètre. Le bouton-poussoir (sans rebond, bien sûr) est connecté sur CCPx

- On lance le timer1, on autorise interruptions TMR1 et CCPx
- Dans l'interruption TMR1, on incrémente une variable
- On presse sur le bouton (start)
- L'interruption CCPx sauve les registres CCPRxH et CCPRxL ainsi que la variable (variable1)
- On presse sur le bouton (stop)
- Lors de l'interruption CCPx on sauve la variable (variable2)
- Le temps exact écoulé entre les 2 événements en nombre de cycles sera : $((\text{variable2} - \text{variable1}) * 65536 + (\text{CCPRxH} - \text{CCPRxH sauve}) * 256 + (\text{CCPRxL} - \text{CCPRxL sauve})) * \text{prédiviseur}$.

L'avantage, c'est que la capture se fait au moment précis de l'événement, et donc le temps de réaction du programme n'intervient plus dans le calcul du temps. De même, plus aucun problème, ni de bug, pour la lecture des 2 octets de TMR1. Et dire que Microchip recommandait de changer de timer pour ce genre d'applications...

Attention, il s'agit d'un pseudo-code simplifié, il vous faudra, pour en faire un programme opérationnel, gérer les risques de débordements de la variable, pour ne citer qu'un exemple. Mais cela vous montre les avantages de cette méthode.

20.4.3 Remarques et limites d'utilisation

L'utilisation de ce mode de fonctionnement impose l'application de certaines règles et conseils de prudence. Il importe d'être attentif. Certaines de ces règles sont d'ailleurs d'application pour le mode « compare ». Voici ces règles :

-Le timer 1 doit impérativement être configuré en mode « timer » ou en mode « compteur synchrone ». **En mode compteur asynchrone, la capture ne fonctionne pas.**

Je vous renvoie au chapitre sur le timer 1 pour plus de renseignements.

- **Tout changement de configuration du mode « capture » peut provoquer un positionnement indésirable du bit CCPxIF.** Avant tout changement, vous devez donc interdire les interruptions CCPx, et forcer CCPxIF à 0 avant de réautoriser les interruptions.

Pour le CCP1 :

```
bsf    STATUS,RP0      ; passer en banque 1
bcf    PIE1,CCP1IE     ; interdire interruptions CCP1
bcf    STATUS,RP0      ; repasser en banque 0
movlw  nouveau_mode    ; nouveau mode pour CCP1
movwf  CCP1CON         ; dans registre de commande
bcf    PIR1,CCP1IF     ; effacer flag d'interruption
bsf    STATUS,RP0      ; passer en banque 1
bsf    PIE1,CCP1IE     ; réautoriser interruptions CCP1
bcf    STATUS,RP0      ; repasser en banque 0
```

Pour le CCP2 :

```
bsf    STATUS,RP0      ; passer en banque 1
bcf    PIE2,CCP2IE     ; interdire interruptions CCP2
bcf    STATUS,RP0      ; repasser en banque 0
movlw  nouveau_mode    ; nouveau mode pour CCP2
movwf  CCP2CON         ; dans registre de commande
bcf    PIR2,CCP2IF     ; effacer flag d'interruption
bsf    STATUS,RP0      ; passer en banque 1
bsf    PIE2,CCP2IE     ; réautoriser interruptions CCP2
bcf    STATUS,RP0      ; repasser en banque 0
```

Ceci, bien entendu, ne concerne que le cas où vous utilisez les interruptions des modules concernés.

Attention, CCP1IE se trouve dans PIE1, alors que CCP2IE se trouve dans PIE2. Même remarque pour CCP1IF qui se situe dans PIR1, alors que CCP2IF est dans PIR2.

- Tout arrêt du mode capture (changement de mode, ou arrêt du module CCPx) provoque l'effacement du contenu du prédiviseur (les événements déjà comptés sont perdus).
- La modification du prédiviseur en cours de fonctionnement peut positionner le bit CCPxIF de façon non souhaitée. Autrement dit, vous devez commencer par interdire les interruptions CCPx, comme expliqué ci-dessus.
- Le contenu du prédiviseur n'est pas effacé lors de cette modification. Il est donc conseillé d'effacer d'abord CCPxCON avant de choisir le nouveau prédiviseur. Ainsi, le contenu de celui-ci sera effectivement effacé.

```
clrf   CCPxCON         ; arrêt du module CCPx
movlw  nouvelle_valeur ; nouveau prédiviseur et CCPx en service
movwf  CCPxCON         ; modifier prédiviseur
```


- Dans le cas où la pin CCPx serait configurée en sortie, l'établissement d'un niveau sur cette pin par logiciel serait interprété comme un niveau entrant, et pourrait donc générer la prise en compte d'un événement de capture, exactement comme si la modification de niveau sur la pin était due à un événement extérieur. Ceci vous permet donc de créer des captures pilotées par soft.

20.4.4 Mode « sleep » et astuce d'utilisation

Si vous placez votre PIC en sommeil (sleep), votre timer 1 ne pourra plus compter, puisque le mode asynchrone n'est pas autorisé dans l'utilisation des modules CCP. Cependant le prédiviseur fonctionne, lui, de façon asynchrone, et est donc dans la possibilité de positionner le flag CCPxIF.

De ce fait, une interruption provoquée par CCPx pourra réveiller votre PIC, bien que la mise à jour des registres CCPRxH et CCPRxL ne se réalise pas dans ce cas.

Vous pouvez penser que ceci ne sert donc à rien, et vous avez en partie raison. Cependant, en réfléchissant un peu, vous constatez que cette astuce vous permet en réalité de disposer de 2 pins supplémentaires (CCP1 et CCP2) capables de vous générer des interruptions.

Donc, si vous ne vous servez pas d'un module CCP, mais que vous avez besoin d'une entrée d'interruption supplémentaire, rien ne vous interdit de configurer le module CCP de votre choix en mode « capture » pour disposer automatiquement de l'entrée d'interruption CCPx correspondante. Il suffit alors d'ignorer le contenu des registres CCPRxH et CCPRxL.

Bien évidemment, vous pouvez également utiliser simultanément les 2 entrées d'interruption CCP1 et CCP2.

Voici donc une astuce qui peut se révéler très pratique, d'autant que ces pins peuvent être configurées en tenant compte d'un prédiviseur ou du choix du sens de transition. Elles se comportent donc comme une « RB0/INT » améliorée.

20.5 Le mode « compare »

Ce mode de fonctionnement est basé sur la correspondance de la valeur du timer 1 (TMR1H/TMR1L) avec la valeur contenue dans CCPRxH/CCPRxL. L'égalité de ces valeurs entraînera les réactions souhaitées.

ATTENTION : Le résultat d'une égalité ne créera l'effet qu'au moment de l'exécution du cycle suivant. Il y a donc toujours un retard de 1 cycle d'instruction entre la vérification de l'égalité, et l'action qui en découle.

Ceci explique toutes les expressions « +1 » dans les formules de calcul des différents temps.

Souvenez-vous qu'il en était de même pour le timer 2, pour lequel nous avons positionnement du flag TMR2IF au moment où TMR2 passait de la valeur de PR2 à 0x00. Le reset de TMR2 intervenait donc bien le cycle suivant son égalité avec PR2.

Les temps calculés de génération d'interruption étaient donc : contenu de PR2 + 1. Il en sera de même pour le mode « compare » de nos modules CCPx. Tous ces « retards » sont dus au mode de fonctionnement synchrone des PICs.

Certaines configurations de ce mode font intervenir la pin CCPx en tant que sortie. Vous devrez donc dans ce cas configurer cette pin en sortie en effaçant le bit de TRISC correspondant.

20.5.1 Principe de fonctionnement

Comme je viens de l'expliquer, l'égalité entre les registres du timer 1 (TMR1H/TMR1L) et la valeur de consigne fixée par les registres CCPRxH/CCPRxL entraîne une ou plusieurs actions en fonction du mode choisi.

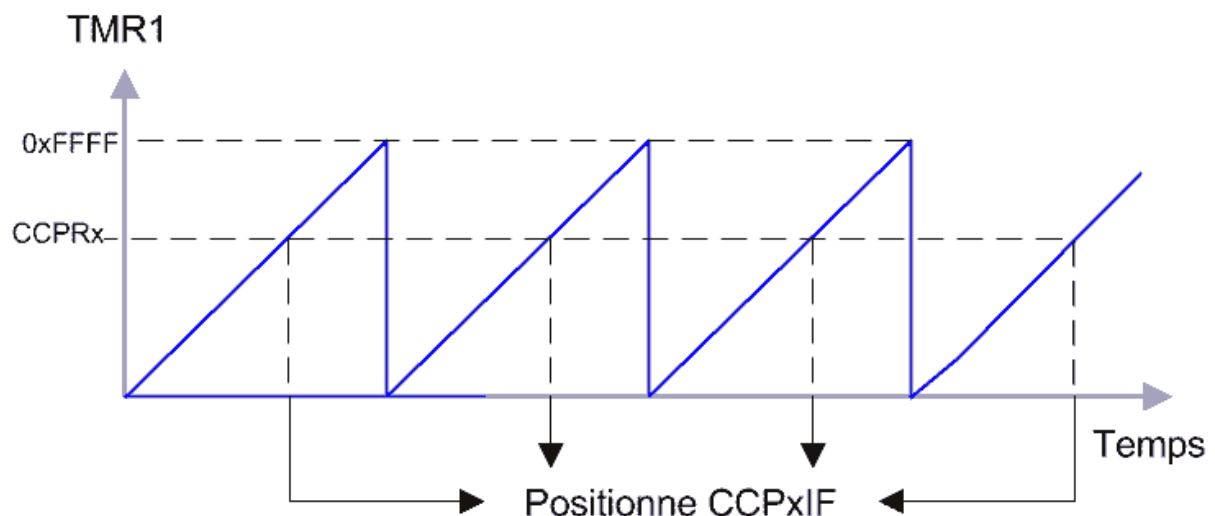
Si nous prenons le mode défini par les bits CCPxM3 à CCPxM0 configurés à « 1010 », nous aurons le fonctionnement suivant :

- Quand le contenu du mot de 16 bits formé par TMR1H/TMR1L atteint celui formé par CCPRxH/CCPRxL, le flag CCPxIF est positionné **dès le cycle suivant**. Une interruption a éventuellement lieu si elle a été préalablement configurée

Attention, le timer 1 ne déborde pas (ne repasse pas à 0x0000), il continue de compter normalement. La prochaine correspondance entraînera donc une nouvelle interruption 65536 cycles plus tard (si on n'a pas modifié les registres concernés entre-temps).

Nous ne sommes donc pas en présence d'un fonctionnement semblable à celui du timer 2, pour lequel le dépassement de la valeur entraînait automatiquement la remise à 0 de celui-ci.

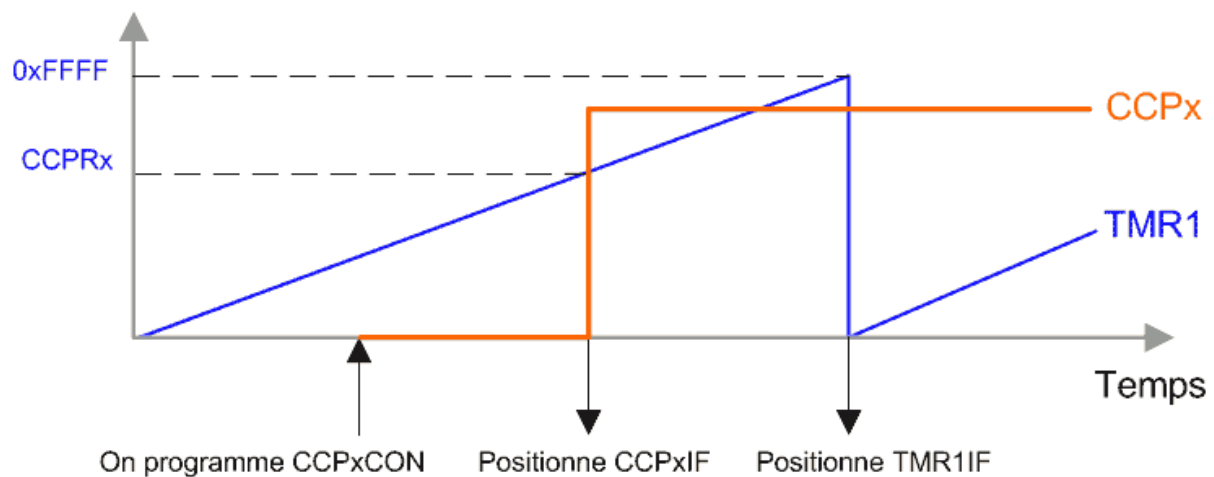
Voici un graphique qui présente l'évolution du contenu de TMR1 (TMR1H/TMR1L) en fonction du temps, et le positionnement de CCPxIF en fonction de CCPRx (CCPRxH/CCPRxL). Attention, vu l'échelle utilisée, le temps d'un cycle n'est pas représentable. N'oubliez pas que les actions s'effectuent toujours le cycle suivant la comparaison.



Autour de ce fonctionnement de base, encore appelé mode « software », nous avons 3 variantes.

La combinaison « 1000 » des bits CCPxM3 à CCPxM0, à condition que la pin CCPx correspondante soit placée en sortie via TRISC, induit le fonctionnement **THEORIQUE** suivant :

- Au moment de la configuration de CCPxCON, la sortie CCPx est forcée automatiquement à « 0 », indépendamment du contenu précédemment placé dans PORTC par le programme
- Le timer 1 compte. Au cycle suivant la correspondance des valeurs du TMR1 et de consigne, le flag CCPxIF est positionné, une interruption a éventuellement lieu.
- La pin CCPx passe à 1 automatiquement, et y reste jusqu'à sa modification par le programme, ou par une modification de CCPxCON



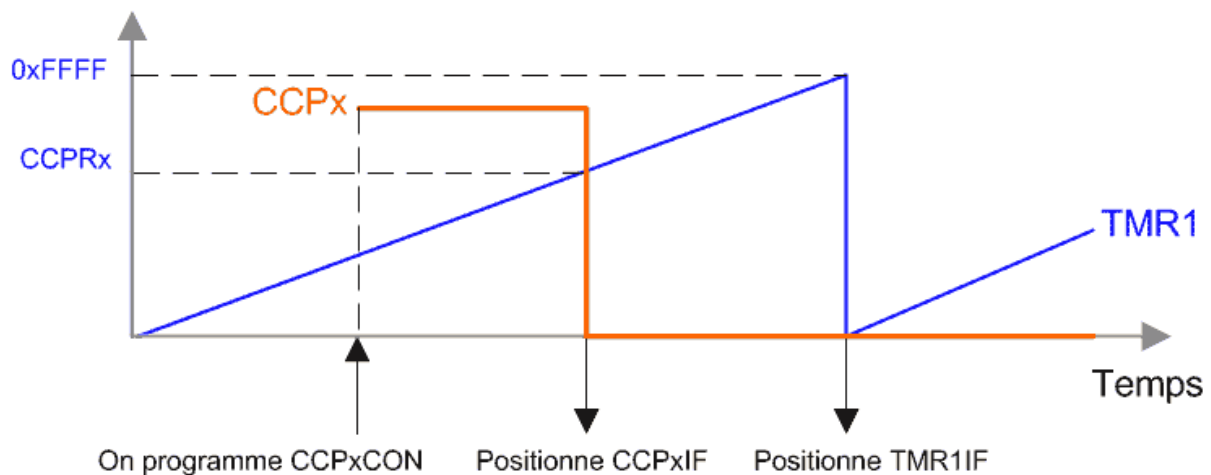
La courbe CCPx commence à l'instant de la programmation de CCPxCON. Avant cet instant, son niveau peut être quelconque (1 ou 0).

La durée du signal utile (ici, la durée de l'état bas) de notre pin CCPx sera :

$$T = (CCPRx + 1) * 4 * T_{osc} * \text{prédiviseur}$$

La combinaison « 1001 » des bits CCPxM3 à CCPxM0 induit un fonctionnement semblable, excepté que les niveaux sur CCPx sont inversés :

- Au moment de la configuration de CCPxCON, la sortie CCPx est forcée automatiquement à « 1 », indépendamment du contenu précédemment placé dans PORTC par le programme
- Le timer 1 compte. Au cycle suivant la correspondance des valeurs du TMR1 et de consigne, le flag CCPxIF est positionné, une interruption a éventuellement lieu.
- La pin CCPx passe à 0 automatiquement, et y reste jusqu'à sa modification par le programme, ou par une modification de CCPxCON



ATTENTION : le fonctionnement réellement observé est le suivant : la pin CCPx N'EST PAS positionnée automatiquement sur la valeur inverse de celle obtenue au moment de la comparaison. En fait, nous devons trouver une astuce pour forcer cette pin manuellement à son niveau de départ. Je vous encourage donc à lire la méthode que j'ai imaginée et utilisée dans le second exercice. J'ai interrogé Microchip au sujet de cette discordance entre datasheet « mid-range » et fonctionnement réel. La réponse que les responsables techniques m'ont donnée est que les datasheets ont été écrits avant la sortie du 16F876, et que ce dernier ne correspond pas, à ce niveau, aux caractéristiques annoncées. Comprenez : « Il s'agit d'un bug ».

J'en profite pour un aparté : Ce n'est pas parce que quelqu'un s'est représenté la référence dit ou écrit quelque chose, que ceci doit faire office de vérité toute puissante et non vérifiée. Tout le monde peut faire des erreurs (de bonne ou de mauvaise foi), à vous de toujours vous interroger sur la validité des messages reçus et prémâchés (je pense principalement aux informations souvent orientées reçues via les media).

Reste une dernière variante à notre option de base, lorsqu'on configure nos bits de contrôle CCPxM3 à CCPxM0 avec la valeur « 1011 ». Dans ce mode, la pin CCPx reste inchangée, mais nous générons un signal « trigger » qui permet de commander automatiquement un ou deux événements supplémentaires.

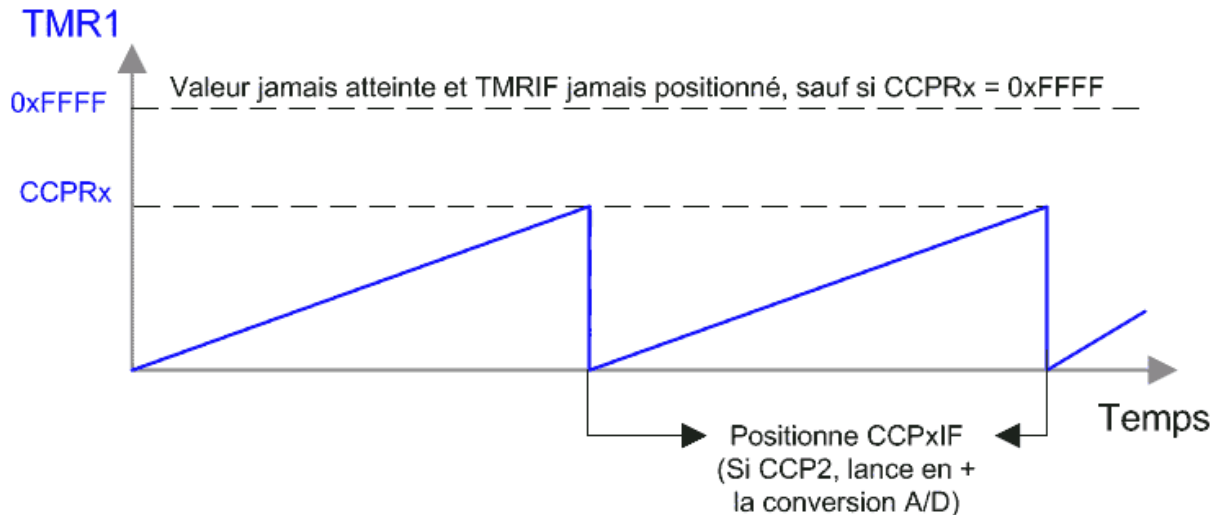
Le trigger en question provoque, sur les 2 modules CCP, le **reset automatique du timer 1**. Donc, ceci nous ramène exactement au **fonctionnement du timer 2, mais sur 16 bits** :

- Le cycle suivant l'égalité des registres TMR1H/TMR1L avec la valeur de consigne CCPRxH/CCPRxL, le timer 1 est remis à 0 (il déborde sur la valeur fixée par CCPRxH/CCPRxL)
- Au moment du débordement, le flag CCPxIF est positionné, et une interruption a lieu si elle a été configurée.

Notez déjà que le débordement provoqué par le module CCP n'induit pas le **positionnement du flag TMR1IF**, SAUF si la valeur placée dans CCPRxH/CCPRxL était

0xFFFF. Dans ce dernier cas, le positionnement des 2 flags CCPxIF et TMR1IF sera simultané.

Mais le trigger provoque une seconde action, **uniquement pour le module CCP2**. Pour ce module, en même temps que les événements précédemment cités, le bit « GO » du registre ADCON0 sera automatiquement positionné, lançant la conversion (si le convertisseur A/D était en service, bien entendu).



Donc, si vous avez compris, et que vous voulez utiliser votre timer 1 avec une valeur de comparaison (comme pour le timer 2), vous utiliserez le module CCP1 ou CCP2 en mode compare avec trigger, et vous serez prévenu (avec interruption éventuelle) de la fin de la durée par le positionnement de CCP1IF ou CCP2IF, et non par TMR1IF.

Le temps de cycle induit par ce mode de fonctionnement, sera donc, en toute logique :

$$T = (CCPRxHL + 1) * T_{cy} * \text{prédiviseur}$$

Ou encore :

$$T = (CCPRxHL + 1) * 4 * T_{osc} * \text{prédiviseur}$$

Avec CCPRxHL le nombre de 16 bits formé par la combinaison de CCPRxH et CCPRxL

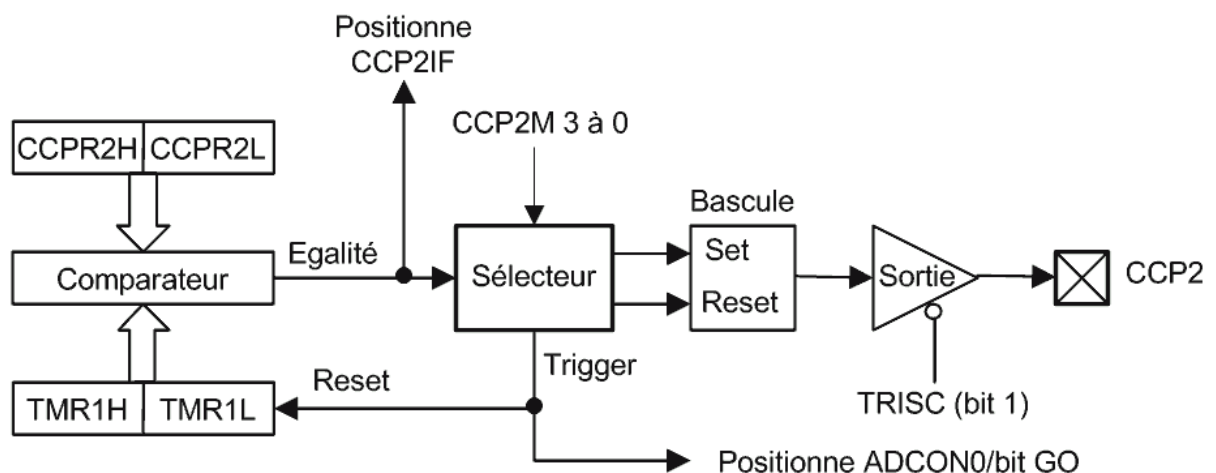
Pour résumer les différentes possibilités du mode « compare », on peut dire :

- Ce mode est basé sur la comparaison du contenu du timer 1 avec une valeur de consigne.
- **Le cycle suivant l'égalité** de ces registres, le flag CCPxIF est positionné
- De plus il est possible simultanément (mais non obligatoire), :
 - Soit de forcer la sortie CCPx à 0 ou à 1
 - Soit de remettre à « 0 » le timer 1, de façon à provoquer un phénomène de débordement pour le module CCP1

- Soit, pour le module CCP2, de resetter le timer 1 ET de lancer la conversion A/D si le convertisseur était en service. Nous verrons l'utilisation pratique de cette fonction dans nos exercices de fin de chapitre.

Vous voyez donc que vous disposez de toute une panoplie de nouvelles fonctions liées aux timers 1.

Voici le schéma-bloc du module CCP2 :



Le module CCP1 est identique, excepté qu'il ne dispose pas de la possibilité de positionner le bit GO du registre ADCON0

20.5.2 Champs d'application

De nouveau, toutes ces possibilités nous ouvrent la voie de nouvelles mises en œuvre de nos applications.

Le mode software pourra être utilisé pour signaler qu'un temps établi de façon précise (comptage sur 16 bits) a été atteint, ou qu'une quantité d'événements comptabilisée par le timer 1 a également été atteinte. Donc, en général, pour toute occurrence unique d'un comptage particulier.

La modification de la sortie CCPx permet de fournir un signal qui intervient un temps précis après le démarrage du timer, ou qui dure un temps précis, ou encore de fournir un signal de sortie après comptabilisation d'un nombre déterminé d'événements reçus par le timer 1. L'avantage de cette méthode est qu'elle n'introduit pas de retard de réaction dû au traitement logiciel de l'occurrence constatée. Nous verrons un exemple pratique de cette possibilité.

Le trigger permet bien évidemment de transformer le timer 1 en un timer à grande flexibilité. Ce mode permet de fait de disposer de l'avantage de la flexibilité du timer 2, en conservant un comptage sur 16 bits. Ce mode devrait pouvoir vous tirer de toutes les situations de mesure de temps et d'événements que vous rencontrerez.

Le trigger du module CCP2, appliqué au convertisseur analogique/numérique permet de faciliter grandement la méthode de numérisation.

Je développe un peu ce dernier point. La méthode d'échantillonnage utilisée sur les PICs implique les opérations suivantes :

- On démarre le convertisseur et on sélectionne le canal
- On attends le temps Tacq de charge du condensateur
- On lance la conversion.

Ces étapes peuvent être transformées, en utilisant le mode « trigger » du module « comparateur » :

- On démarre le convertisseur et on sélectionne le canal
- On programme le temps Tacq dans le module CCP2.

Une fois le temps Tacq terminé, le module CCP2 se chargera lui-même de démarrer la conversion. Vous n'avez donc plus à vous charger de détecter, par interruption ou par pooling, la fin de la durée Tacq.

20.5.3 Remarques et limites d'utilisation

De nouveau, j'attire votre attention sur certains points de ce mode de fonctionnement :

- Le module CCPx en mode « compare » ne peut fonctionner que si le timer 1 fonctionne en mode « timer » ou en mode « compteur synchrone ». **Le fonctionnement en mode « compteur asynchrone » ne permet pas le fonctionnement de ce module.**
- **L'utilisation de la pin CCPx dans ce mode n'est possible qu'après avoir configuré la pin concernée en sortie, en effaçant le bit du registre TRISC correspondant**
- **Il n'est pas possible de mélanger les modes de fonctionnement du mode « compare ».** Par exemple, il est impossible d'obtenir le positionnement de la pin CCPx et d'utiliser simultanément le trigger.

20.5.4 Le mode « sleep »

Le timer 1 configuré en « timer » ou en « compteur synchrone » ne peut compter si le PIC est placé en mode sommeil. Donc, il n'y a aucune chance dans ces conditions que la correspondance de valeurs entre TMR1H/TMR1L et CCPRxH/CCPRxL intervienne.

Comme la suite des événements dépend de cette occurrence, **le mode « compare » est donc inutilisable durant la mise en sommeil du PIC.**

Lorsque le PIC se réveille, suite à un autre événement, le fonctionnement se poursuit où il avait été arrêté.

Si une pin CCPx est utilisée comme sortie du module, le mode « sleep » maintient cette pin dans l'état actuel.

20.5.5 Fonctionnement non conforme

Je rappelle ici la remarque que j'ai précédemment faite lors de l'étude théorique du mode compare avec pilotage de la pin CCPx :

Attention, ce qui suit concerne exclusivement les modèles de 16F87x disponibles au moment de la réalisation de cet ouvrage. Il vous appartiendra, si vous voulez vous assurer d'une éventuelle correction, de vous renseigner sur le site de Microchip, ou de tenter vous-même l'expérience avec les PICs en votre possession.

Le datasheet des pics mid-range de Microchip est très clair : si on inscrit la valeur « B'00001001' » dans le registre CCP1CON, la sortie CCP1 (si elle est configurée en sortie) doit passer instantanément à « 1 ».

En fait, il n'en n'est rien. Si le module CCP force effectivement cette pin à « 0 » une fois la comparaison entre CCPR1HL et TMRHL réalisée, par contre l'écriture d'une valeur dans CCP1CON ne permet pas sa remise à 1 instantanée.

Il est possible que Microchip corrige ce problème dans les futures versions du 16F876. Dans le cas contraire, voyez l'exercice 2 de pilotage de servomoteur dans lequel je donne une méthode « maison » pour corriger ce phénomène.

Autre remarque : Microchip indique que l'utilisation d'un module CCP utilisé en mode « compare » en même temps qu'un module utilisé en mode « capture », ou de 2 modules CCP utilisés en mode « compare », est soumis à la contrainte suivante :

Chacun des modules compare précédemment impliqués devra être utilisé exclusivement en mode « compare avec trigger ». Non seulement j'ai trouvé ceci illogique (car le premier reset empêche le second d'être exécuté), mais, de plus, les essais que j'ai réalisés démontrent que ce n'est pas du tout le cas. Je n'ai donc pas retenu cette contrainte, preuve en est que le second exercice utilise 2 modules CCP en mode compare, dont un seul est utilisé avec trigger.

De nouveau, en cas de nouvelle version de 16F87x , à vous de vérifier si ce fonctionnement reste constant (pour ma part, je pense que oui)

Mes propres conclusions (qui n'engagent donc que moi) sont les suivantes :

- La première « erreur » est un bug présent dans les 16F87x, preuve en est le nombre de corrections sur le même sujet présentes sur le site Microchip pour toute une série d'autres PICs.
- La seconde erreur est une erreur dans les datasheets, le fonctionnement des PICs semble beaucoup plus logique que la contrainte imposée par les datasheets. En effet, l'utilisation, par exemple, de 2 modules CCP en mode « compare avec trigger » n'a pas le moindre sens (un reset du timer1 empêche l'autre module CCP d'atteindre sa valeur de comparaison).

20.6 Le mode « PWM »

Nous voici dans la partie la plus délicate des possibilités des modules CCP. Non que les principes soient compliqués, mais plutôt qu'il n'est pas simple de les expliquer de façon claire et concise quel que soit le niveau du lecteur.

Je vais donc tenter de faire de mon mieux, mais cela nécessitera un peu de théorie (comme d'habitude, allez-vous me dire). Et oui, les 16F87x sont des PICs de « pros » qui nécessitent plus de cette théorie indispensable pour exploiter correctement toutes les fonctions que le petit 16F84.

Notez que si vous êtes déjà un vrai « pro », ceci doit vous embêter, mais il faut bien penser à tout le monde, non ? Et puis, un peu de lecture, ça ne fait de tort à personne.

20.6.1 La théorie du « PWM »

PWM signifie « **P**ulse **W**idth **M**odulation », ce qu'on pourrait traduire par **modulation de largeur d'impulsion**.

En somme, il s'agit d'un **signal binaire de fréquence fixe** dont le **rapport cyclique** peut être **modulé** par logiciel.

Etant donné qu'un signal binaire n'a plus de secret pour vous, vous savez donc qu'il s'agit d'un signal qui peut prendre 2 états. Notre module « PWM » utilisera **une pin de notre PIC configurée en sortie**.

Il me faut aborder la notion de rapport cyclique, la modulation étant simplement l'expression du fait que celui-ci peut être modifié en permanence.

Le **rapport cyclique** d'un signal binaire à fréquence fixe peut être défini comme étant le **rapport entre le temps où il se trouve à l'état « 1 » par rapport au temps total d'un cycle**. Un cycle n'étant constitué, par définition, que d'un état « 1 » suivi d'un état « 0 », la somme des temps des 2 états étant constante.

Notez donc qu'il y a **2 paramètres** qui définissent un signal « PWM » :

- **La durée d'un cycle complet** (ou, par déduction, sa fréquence de répétition)
- **Le rapport cyclique**

Donc, si on pose :

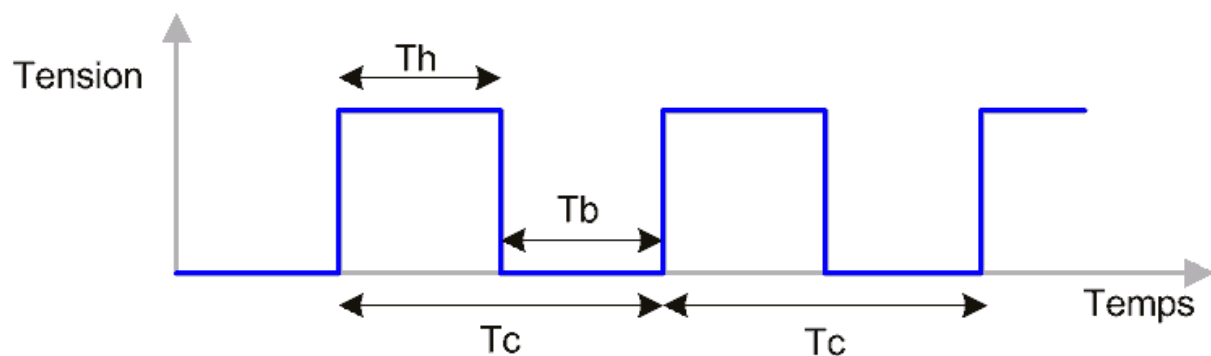
- T_c = Durée d'un cycle
- R_c le rapport cyclique
- T_h = durée de l'état haut
- T_b = durée de l'état bas

, on peut dire :

- $T_c = T_h + T_b$ (Durée d'un cycle en secondes = durée de l'état haut + durée de l'état bas)
- Fréquence du signal (en hertz) $= 1/T_c$
- $R_c = T_h / T_c$ (rapport cyclique en % = temps à l'état haut divisé par le temps de cycle)

Je vais donc commencer par illustrer, de façon claire, quelques exemples de signaux de ce type.

Exemple d'un signal PWM avec un rapport cyclique de 50% :

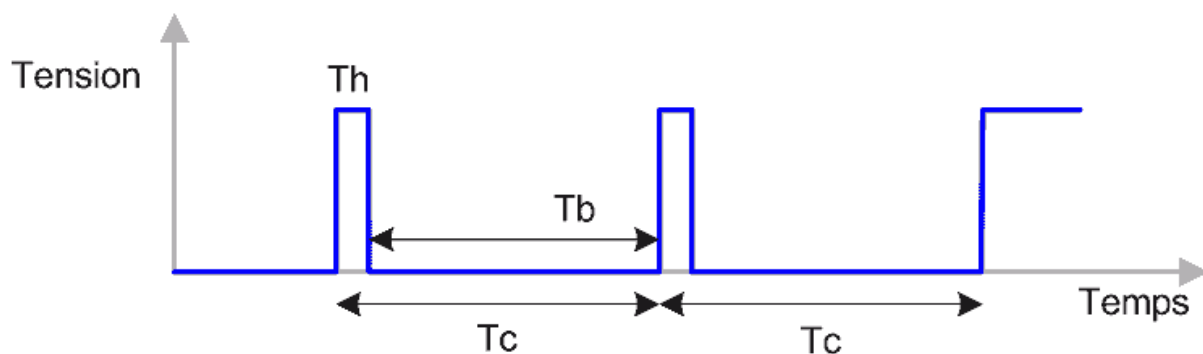


Vous voyez que ce signal est effectivement de fréquence fixe, puisque chaque temps T_c est identique.

Le temps T_h est identique au temps T_b , ce qui donne bien un rapport cyclique de 50%, puisque $R_c = T_h / T_c = T_h / (T_b + T_h) = T_h / (2 T_h) = \frac{1}{2} = 50\%$

Un tel signal s'appelle **signal carré**. C'est un cas particulier d'un signal PWM.

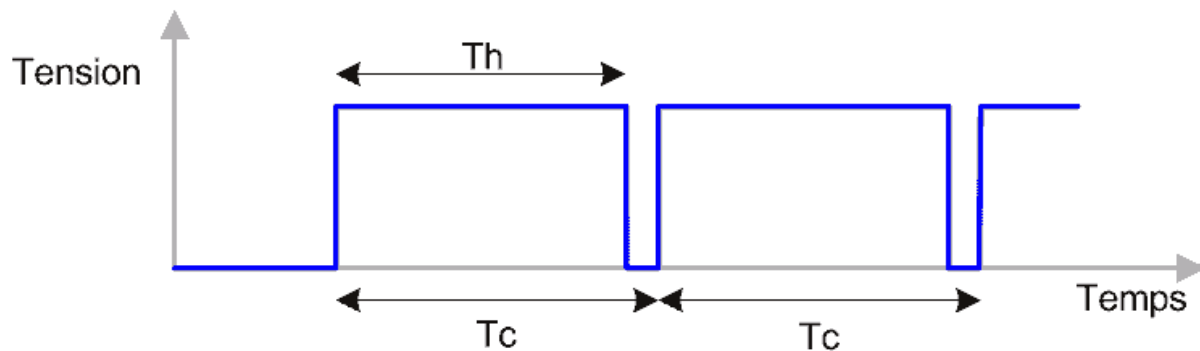
Exemple d'un signal PWM avec un rapport cyclique de 10% :



Vous pouvez constater que ce signal possède strictement le même temps de cycle T_c que le précédent. Sa fréquence est donc identique.

Cependant, son rapport cyclique a été modifié. T_h représente maintenant 10% du temps de cycle total, alors que T_b représente 90%

Voyons maintenant un rapport cyclique de 90% :



De nouveau, le temps T_c est inchangé, seul le rapport cyclique a été modifié. Le temps T_h dure maintenant 90% du temps T_c , T_b occupant fort logiquement les 10% restants.

Reste maintenant à voir les 2 cas particuliers restants, à savoir le rapport cyclique de 0%, et celui de 100%.

Fort logiquement, un signal avec un rapport cyclique de 0% est un signal dont le temps à l'état haut occupe 0% du temps total. C'est donc un signal qui est constamment à l'état bas.

De même, un signal avec un rapport cyclique de 100% est un signal qui est constamment à l'état haut.

Cependant, si vous dessinez des temps signaux, vous constatez qu'il vous est impossible d'indiquer le temps T_c , donc sa fréquence. Et, de fait, ce signal ne variant plus, n'est plus un signal périodique au sens électronique du terme (les mathématiciens vont me sortir des temps infinis, mais cela ne correspond pas à la réalité électronique).

Le rapport cyclique d'un signal PWM doit donc être supérieur à 0% et inférieur à 100%.

Voici une partie de la théorie abordée, vous avez maintenant compris que le module PWM de notre PIC permet de créer un signal périodique dont il est possible de faire varier (moduler) le rapport cyclique.

20.6.2 La théorie appliquée aux PICs

Nous avons vu que nous avons besoin de 2 choses pour créer notre signal. D'une part, le temps T_c , qui détermine la fréquence (fixe) de notre signal, et d'autre part le rapport cyclique (variable) de ce dernier.

Concernant le rapport cyclique, les PICs influencent plutôt un autre paramètre, c'est-à-dire le temps T_h .

Les 2 valeurs utilisées dans la programmation seront donc T_c et T_h . Si vous avez besoin d'une autre valeur (R_c), il vous suffit de la tirer des formules simples précédentes.

Nous allons maintenant étudier la façon de gérer le premier paramètre, à savoir le temps T_c .

Ce temps est défini tout simplement par le timer 2. Vous programmez éventuellement le prédiviseur, vous chargez la valeur adéquate dans PR2, et le temps mis par votre TMR2 pour déborder vous donne votre temps T_c .

Déjà une remarque très importante :

Le postdiviseur n'est pas utilisé dans le module PWM. Donc n'intervient pas dans le calcul de T_c .

Ceci implique également que vous pouvez utiliser votre timer 2 en tant que générateur pour le module PWM, et, grâce au postdiviseur, travailler avec un autre temps (multiple du premier) dans le reste de votre programme.

Pour faire simple, avec le même timer TMR2, avec T_{cy} = durée d'un cycle d'instruction, et en vous souvenant de la formule donnée au moment de l'étude du timer 2 :

- Temps du cycle utilisé pour le module PWM : $T_c = (T_{cy} * \text{prédiviseur}) (PR2 + 1)$
- Temps du timer2 utilisé « classiquement » dans le programme = $T_c = (T_{cy} * \text{prédiviseur} * \text{postdiviseur}) (PR2 + 1)$

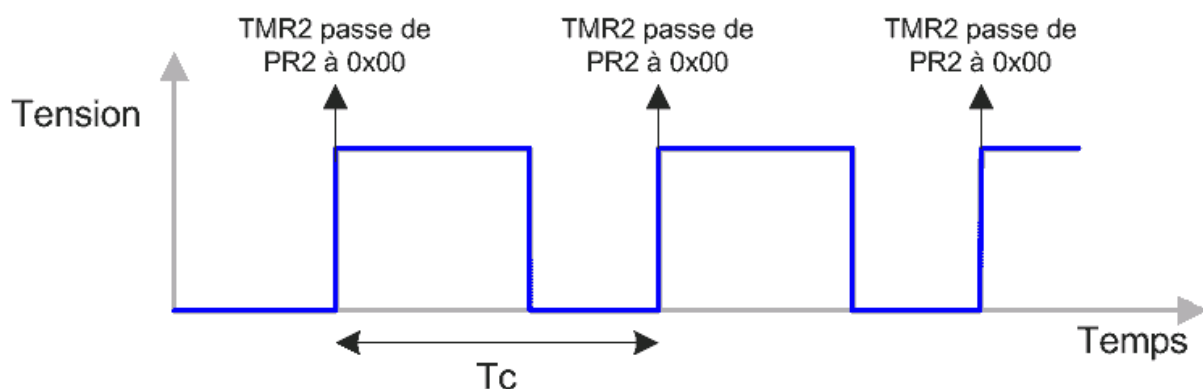
Inconvénient : il ne sera pas possible d'obtenir des temps longs avec notre module PWM. Celui-ci est donc prévu pour travailler avec des fréquences relativement importantes.

Nous avons déjà parlé du timer 2 dans son emploi classique, ce qui nous intéresse donc ici est la première formule utilisée dans le module PWM.

Le module PWM travaille avec le timer 2, et au niveau de T_{cy} de la façon suivante :

- Vous entrez votre valeur de débordement dans PR2
- A chaque fois que TMR2 déborde de PR2 à 0x00, la pin CCPx passe au niveau 1.

Donc, si nous reprenons un signal PWM quelconque, nous avons déjà :



Comme vous constatez, nous avons déterminé l'emplacement de montée du signal de « 0 » à « 1 ». Or, comme le temps T_c est le temps séparant 2 montées successives du signal (ou 2 descentes), nous avons défini T_c . Pour rappel :

$$T_c = (PR2 + 1) * T_{cy} * \text{prédiviseur}$$

Ou encore

$$T_c = (PR2 + 1) * 4 * T_{osc} * \text{prédiviseur}$$

En effet, un cycle d'instruction vaut 4 cycles d'oscillateur.

Reste donc à définir l'emplacement de redescente de notre signal pour obtenir notre signal complet.

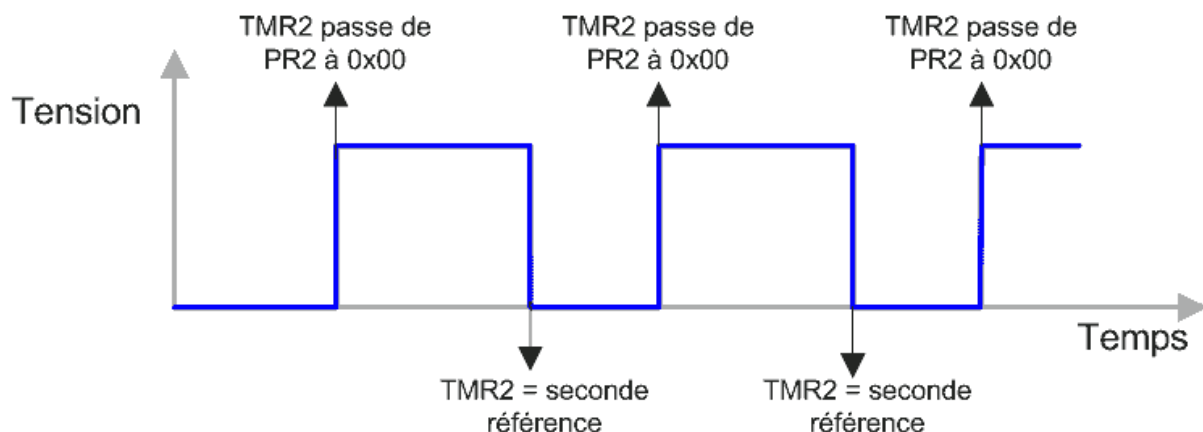
Ce qu'il nous faudrait, c'est un deuxième registre « PR2 », qui nous indiquerait, au moment où TMR2 atteindrait cette valeur, que le signal doit retomber à « 0 ». Et bien, c'est ce qui est implémenté, en gros, dans notre PIC.

Je vais commencer par le principe général, et je vais me rapprocher au fur et à mesure de l'exposé de la situation réelle. Le principe va donc être le suivant :

Le Timer 2 compte : on imagine que le signal CCP vaut actuellement « 0 » :

- TMR2 arrive à la valeur de PR2
- Au cycle suivant, TMR2 repasse à « 0 », CCPx passe à « 1 »
- TMR2 arrive à la seconde valeur de consigne, CCPx passe à « 0 »
- TMR2 arrive à la valeur de PR2
- Au cycle suivant, TMR2 = 0, CCPx vaut « 1 »
- TMR2 arrive à la seconde valeur de consigne, CCPx passe à « 0 »
- Et ainsi de suite...

Ceci vous donne la représentation suivante :



Vous pouvez immédiatement faire la constatation suivante :

Pour que le système fonctionne, la valeur inscrite en tant que seconde référence doit être inférieure à celle de (PR2+1), sans quoi TMR2 n'atteindrait jamais cette valeur. De plus, cette valeur, contrairement à PR2 ne doit pas provoquer le reset du timer, sans quoi il lui serait impossible d'atteindre la valeur de PR2.

Quelle sera donc notre marge de manœuvre, et de ce fait la précision obtenue ?

Et bien, elle dépendra de la valeur inscrite dans le registre PR2. Plus cette valeur est faible, moins nous avons le choix des valeurs à inscrire en tant que seconde référence, cette dernière devant rester inférieure à PR2 et supérieure à 0.

Or PR2 dépend du calcul de notre temps Tc. Nous ne pouvons donc pas y mettre ce que nous voulons pour notre application pratique, à moins de décider de choisir le quartz le mieux adapté à notre application.

Supposons donc que notre PR2 contienne la valeur décimale D'50'. Pour faire varier notre rapport cyclique de 0 à 100%, nous ne pouvons mettre que des valeurs de D'0' à D'50'. Donc, nous ne pouvons régler notre rapport cyclique qu'avec une précision de $1/50^{\text{ème}}$, soit 2%.

Dans la plupart des applications, cette précision risque d'être insuffisante. C'est pourquoi Microchip vient une fois de plus à notre secours.

Comme nous ne pouvons pas augmenter la valeur de référence, nous allons simplement y ajouter des « décimales », ceci affinera notre possibilité de réglage, et donc la précision finale obtenue dans les mêmes proportions.

Notez que le terme « décimal » est impropre dans ce cas puisque nous sommes en système binaire, et non décimal. Mais j'ai trouvé l'image plus « parlante ». Par la suite, je parlerai de nombres fractionnaires.

Il a été décidé, pour nos PICs, d'ajouter 2 « décimales binaires » à notre compteur TMR2. Ce compteur se présentera donc comme étant de la forme :

$b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 . b_{-1} b_{-2}$

Ceci formant un nombre de 10 bits effectifs, et donc multiplie la précision par 4. Bien entendu, pour que ça serve à quelque chose, notre seconde valeur de comparaison disposera également de ces 2 digits supplémentaires.

Maintenant, que représentent ces digits « après la virgule » ? C'est simple. Tout comme le premier chiffre après la virgule d'un nombre décimal représente des multiples de $1/10$ (10^{-1}), le second des multiples de $1/100$ (10^{-2}), le premier bit après la virgule représente des multiples de $1/2$ (2^{-1}), le second des multiples de $1/4$ (2^{-2})

Nos chiffres $b_{-1} b_{-2}$ représentent donc le nombre de $1/4$ du temps d'incrément de notre TMR2. Donc, en toute bonne logique :

- Avec un prédiviseur de 1, la précision est de $T_{cy} / 4$, soit T_{osc}
- Avec un prédiviseur de 4, la précision est de T_{cy} , soit $4 T_{osc}$
- Avec un prédiviseur de 16, la précision est de $4 * T_{cy}$, soit $16 T_{osc}$

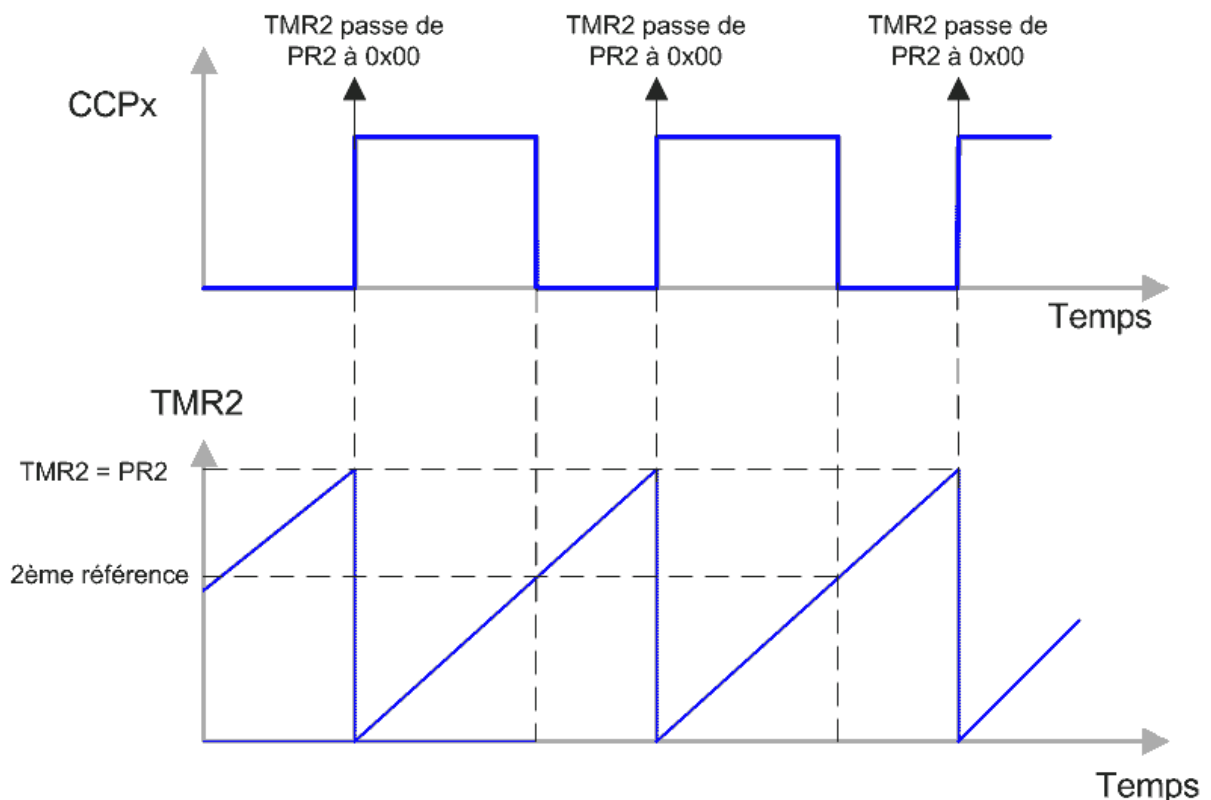
Comme notre PR2, et donc le temps T_c continue de fonctionner, lui, sur 8 bits, notre cycle se déroule donc de la façon suivante :

Le Timer 2 compte : on imagine que le signal CCP vaut actuellement « 0 »

- TMR2 (8 bits) arrive à la valeur de PR2 (8 bits)
- Au cycle suivant, TMR2 repasse à « 0 », CCPx passe à « 1 »
- TMR2 + 2 « décimales » atteint la seconde valeur de consigne (8 bits + 2 « décimales »), CCPx passe à « 0 », le timer 2 continue de compter.
- TMR2 (8 bits) arrive à la valeur de PR2 (8 bits)
- Au cycle suivant, TMR2 = 0, CCPx vaut « 1 », et ainsi de suite

Je sais que ça peut paraître ardu, mais si vous avez compris ce qui précède, alors vous n'aurez aucun problème pour la suite.

Voici d'ailleurs représenté sous forme d'un graphique, la chronologie des événements en fonction du contenu du registre TMR2 du timer 2 :



Faites attention aux datasheets de Microchip, ils ne parlent pas de nombres fractionnaires, ils parlent de nombres de type b9 b8 b0. N'oubliez pas qu'en fait, exprimés de la sorte, ces valeurs sur 10 bits concernant le module PWM représentent des quarts de valeurs entières.

La précision obtenue pour le réglage du rapport cyclique dépend de la valeur inscrite dans PR2, complétée par les 2 digits fractionnaires ajoutés.

Un exemple :

Avec un PIC cadencé à 20MHz, et une fréquence de votre signal PWM de 78,125Khz :

Le temps T_c vaut : $1/78,125 \text{ Khz} = 12,8 \mu\text{s}$

$$T_c = (PR2 + 1) * \text{Prédiviseur} * 4 * T_{osc}$$

$$T_{osc} = 1/20\text{MHz} = 50 \text{ ns}$$

$$PR2 = (T_c / (\text{prédiviseur} * 4 * T_{osc}) - 1$$

$$PR2 = (12,8\mu\text{s} / 200\text{ns}) - 1 = 64 - 1 = 63 \text{ (avec prédiviseur = 1)}$$

Donc, on placera 63 dans notre registre PR2

Il sera alors possible d'ajuster notre rapport cyclique sur des valeurs comprises entre B'00000000,00' et B '00111111,11 », soit 256 valeurs différentes, donc une précision sur 8 bits, ou encore une marge d'erreur de 0,4%.

En somme, la précision vaut $1 / ((PR2+1) * 4)$, et est donc en toute logique multipliée par 4 par rapport à une comparaison sans utiliser de fractions (sur 8 bits).

Il reste à préciser un point. Les signaux PWM sont en général utilisés, comme leur nom l'indique, pour moduler le rapport cyclique d'un signal, et donc pour faire varier continuellement celui-ci en fonction des résultats à obtenir.

Comme il faut intervenir sur des valeurs de consigne de 10 bits, l'écriture ne pourra pas se faire en une seule fois, et donc pourra provoquer une valeur temporaire indésirable. Ce phénomène est appelé « glitch ».

Mais, de nouveau, Microchip nous fournit une solution, en utilisant un registre intermédiaire qui servira de valeur de comparaison, et qui sera chargé au moment du débordement de TMR2.

La procédure exacte est donc la suivante :

- Le Timer 2 compte : on imagine que le signal CCP vaut actuellement « 0 »
- TMR2 arrive à la valeur de PR2
- Au cycle suivant, TMR2 repasse à « 0 », CCPx passe à « 1 »
- En même temps, la valeur programmée comme consigne par l'utilisateur est copiée dans le registre final de comparaison.
- TMR2 arrive à la valeur de la copie de la seconde valeur de consigne, CCPx passe à « 0 »
- TMR2 arrive à la valeur de PR2
- Au cycle suivant, TMR2 = 0, CCPx vaut « 1 », et ainsi de suite

Donc, vous en déduirez que toute modification de la valeur de comparaison, et donc de la durée de T_h ne sera effective qu'à partir du cycle suivant celui actuellement en cours.

On conclut cette partie fortement théorique en résumant les formules qui seront utiles. Tout d'abord le calcul de PR2, qui fixe le temps total de cycle T_c :

$$PR2 = (TC / (\text{prédiviseur} * 4 * T_{osc}) - 1$$

Ensuite, la formule qui lie les 10 bits de notre second comparateur, sachant que ces 10 bits (COMPARE) expriment un multiple de quarts de cycles d'instructions, donc un multiple de temps d'oscillateur, avec le temps du signal à l'état haut (Th).

$$Th = COMPARE * \text{prédiviseur} * T_{osc}$$

Donc,

$$COMPARE = Th / (\text{prédiviseur} * T_{osc})$$

On peut également dire, en fonction du rapport cyclique (Rc), que

$$Th = T_c * Rc$$

Il faut encore rappeler que pour comparer TMR2 avec COMPARE codé sur 10 bits, il faut que TMR2 soit également codé sur 10 bits. Il faut donc compléter TMR2 avec 2 bits qui représentent les quarts de cycle d'instruction.

En fait, tout se passe très simplement. Si vous choisissez un prédiviseur de 1, TMR2 sera complété, en interne, par le numéro (de 0 à 3) du temps T_{osc} en cours.

Si, vous utilisez un prédiviseur de 4, ce nombre sera complété par le nombre d'événements déjà comptabilisés par le prédiviseur (de 0 à 3).

Si, par contre, vous utilisez un prédiviseur de 16, ce nombre sera complété par le nombre de multiples de 4 événements déjà comptabilisés par le prédiviseur.

De cette façon, TMR2 sera bien complété avec 2 bits qui représentent des quarts de valeur. Inutile donc de vous en préoccuper, c'est l'électronique interne qui s'en charge, mais, au moins, comme ça, vous le savez.

20.6.3 Les registres utilisés

Nous voici débarrassé de la théorie pure, passons maintenant à un peu de concret.

Nous avons besoin de plusieurs registres pour programmer toutes ces valeurs. En fait, nous les avons déjà tous rencontrés, ne reste donc qu'à expliquer leur rôle dans ce mode particulier.

Nous avons besoin d'une valeur de débordement pour notre timer 2, cette valeur se trouve, comme je l'ai déjà dit dans le registre PR2. C'est donc une valeur sur 8 bits.

La valeur de la seconde comparaison (celle qui fait passer la sortie de 1 à 0) est une valeur de 8 bits complétée de 2 bits fractionnaires.

Le nombre entier sera inscrit dans le registre CCPRxL. Les 2 bits fractionnaires qui complètent ce nombre sont les bits DCxB1 et DCxB0 du registre CCPxCON.

Comme nous l'avons vu, ce nombre de 10 bits sera copié en interne par le PIC vers un autre registre, qui sera le registre **CCPRxH** complété de 2 bits internes non accessibles. Notez qu'en mode « PWM », il vous est impossible d'écrire dans ce registre. Vous n'avez donc pas, en fait, à vous en préoccuper, le PIC s'en charge.

Pour lancer le mode « PWM », nous devons donc procéder aux initialisations suivantes :

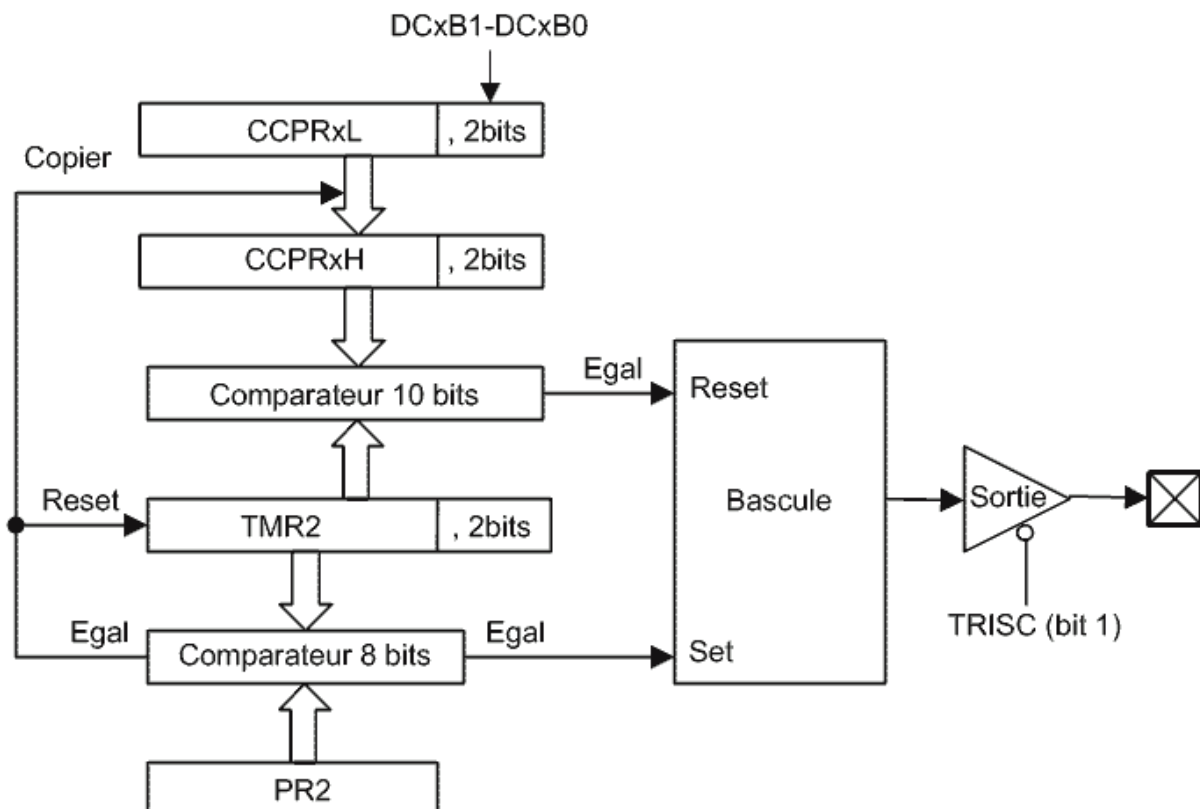
- 1) On initialise PR2 en fonction de la durée totale du cycle (Tc):

$$PR2 = (TC / (\text{prédiviseur} * 4 * T_{osc}) - 1$$
- 2) On calcule la valeur de comparaison DCB en valeur fractionnaire suivant la formule :

$$DCB = Th / (\text{prédiviseur} * T_{osc})$$

On place les bits 9 à 2 dans CCPRxL (valeur entière), les bits 1 et 0 (fraction) étant positionnés dans DCxB1 et DCxB0 du registre CCPxCON
- 3) On place la pin CCPx en sortie en configurant TRISC
- 4) On lance le timer 2 en programmant son prédiviseur
- 5) On configure CCPxCON pour travailler en mode « PWM ».

Vous voyez qu'après avoir ingurgité toute la théorie, la méthode vous semble simple, et c'est le cas. Je vous donne le schéma-bloc simplifié du module « PWM » :



Je commente un peu. Vous voyez en dessous le fonctionnement normal de notre timer 2, qui déborde sur PR2 grâce au comparateur 8 bits. Ceci entraîne en même temps la recopie des 10 bits de CCPRxL/DCxB1/DCxB0 vers CCPRxH/2 bits internes.

Au même moment, la sortie est forcée à « 1 » via la bascule et l'étage de sortie, validé par le registre TRISC. Pour information, le fonctionnement de la bascule est simple : une impulsion sur l'entrée « set », force la sortie à « 1 », tandis qu'une impulsion sur l'entrée « reset » force cette même sortie à « 0 ». Entre les 2 impulsions, la sortie ne change pas d'état.

Quand la valeur TMR2, complétée par 2 « décimales » devient identique la valeur qui a été copiée dans CCPRxH, la sortie est forcée à « 0 ». Vous voyez, rien de magique, cela devrait commencer à devenir clair.

20.6.4 Champs d'application

En fait, vous utiliserez ce module chaque fois que vous aurez besoin d'un signal de fréquence fixe, mais de rapport cyclique variable.

Citons par exemple l'exemple typique du pilotage d'un servomoteur utilisé en modélisme (j'y reviendrai pour l'exercice pratique), ou la variation de vitesse d'un moteur à courant continu (modélisme), et bien d'autres applications.

20.6.5 Remarques et limites d'utilisation

Pour que tout fonctionne comme prévu, il faut veiller à respecter certaines contraintes, auxquelles j'ai déjà fait allusion. Un rappel ne fait cependant pas de tort :

La valeur de référence encodée dans CCPRxL ne peut être supérieure à la valeur contenue dans PR incrémentée de 1. Dans le cas contraire, le signal ne pourrait jamais atteindre cette consigne, et la pin CCPx resterait bloquée au niveau haut (rapport cyclique > 100%)

Il est possible d'utiliser le timer 2 à la fois comme timer classique et comme générateur pour le module PWM. Dans ce cas, prédiviseur et valeur de PR2 seront forcément identiques. Par contre, comme le postdiviseur n'est pas utilisé dans le module PWM, il reste possible d'obtenir des temps de « timer » multiples du temps utilisé dans le module PWM.

Le registre CCPRxH n'est pas accessible en écriture lors du fonctionnement en mode PWM.

La prise en compte du changement de valeur du rapport cyclique ne se fera qu'après la fin du cycle en cours.

20.6.6 Le mode « sleep »

La mise en sommeil du PIC provoque l'arrêt du timer 2. Le module « PWM » ne pourra donc pas continuer de fonctionner dans ces conditions.

La pin CCPx maintiendra la tension qu'elle délivrait au moment du passage en sommeil du PIC.

Au réveil de celui-ci par un événement déclencheur, le cycle se poursuivra à l'endroit où il avait été arrêté.

20.7 Exercice pratique : commande d'un servomoteur par le PWM

Enfin, une application concrète. J'ai choisi de vous faire réaliser le pilotage d'un servomoteur de modélisme, dont la position sera tributaire de la position d'un potentiomètre de réglage.

De cette façon, sous allons mettre en œuvre dans un premier temps :

- Le fonctionnement PWM du module CCP
- L'utilisation du mode compare avec trigger du module CCP pour lancer la conversion A/D
- La méthode efficace d'utilisation du convertisseur A/D

Voyons tout d'abord les contraintes. Un module en mode « PWM » et un autre en mode « compare » ne pose aucun problème de contrainte. Mais le seul module capable de démarrer le convertisseur A/D est le CCP2. Ceci nous impose donc d'utiliser CCP1 pour le mode « PWM », et donc, de fait, de piloter ce servomoteur via la pin CCP1/RC2

Une fois de plus, programmation et électronique devront être étudiés ensemble.

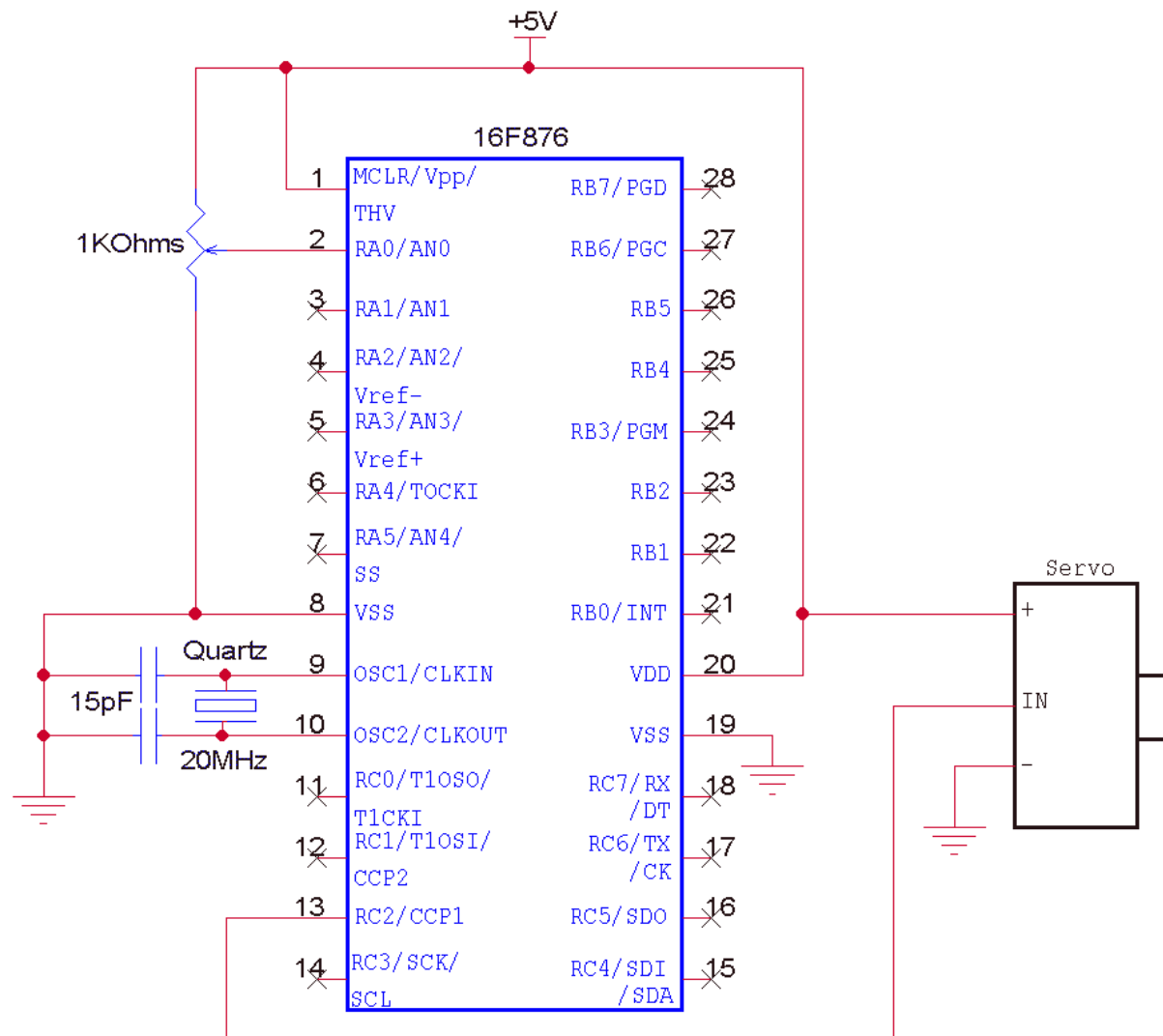
Le potentiomètre nous envoie une valeur analogique. Nous n'avons pas besoin de tension de référence externe. Le tableau de paramétrage de PCFG nous donne, pour une entrée analogique sans tension de référence :

PCFG 3 à 0	AN4 RA5	AN3 RA3	AN2 RA2	AN1 RA1	AN0 RA0	Vref-	Vref +	A/D/R
1110	D	D	D	D	A	Vss	Vdd	1/4/0

Ceci nous impose l'utilisation de AN0/RA0 comme pin d'entrée analogique.

Vous voyez de nouveau que notre logiciel nous impose 2 contraintes matérielles. Plus vous utilisez les fonctions dédiées des PICs, plus vous devez vous préoccuper de votre logiciel avant de pouvoir dessiner votre schéma.

Le schéma, donc, sera du type :



Remarquez que votre servo dispose de 3 pins, dont 2 destinées à l'alimentation, et une recevant un signal de type PWM destiné à indiquer la position de consigne souhaitée.

Ce servo est sensé fonctionner sous 5V, et avec un courant de commande « IN » inférieur à 20mA. Dans le cas contraire, adaptez votre schéma en conséquence.

Remarquez pour ceux qui ne disposent pas d'un servo et qui désirent réaliser ce montage à moindre coût, qu'ils disposent d'autres méthodes pour vérifier le fonctionnement correct du montage :

- Ceux qui ont un voltmètre à aiguille peuvent placer ce dernier sur la pin RC2. La tension moyenne indiquée sera fonction du rapport cyclique fourni.
- Ceux qui disposent d'un oscilloscope peuvent placer celui-ci sur la pin RC2, ils pourront voir directement la forme des signaux obtenus.

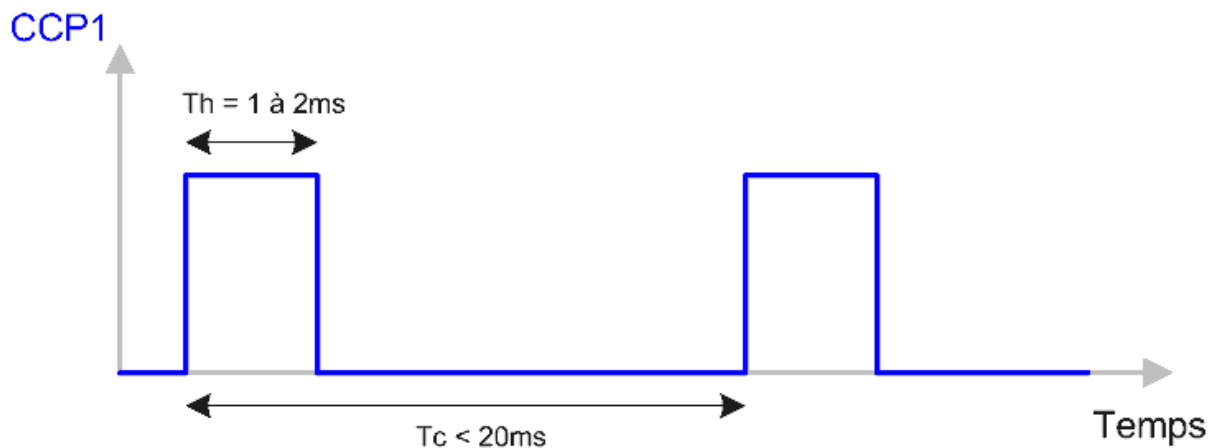
Reste maintenant à établir quels sont les signaux nécessaires à notre servomoteur.

La commande d'un servomoteur nécessite la fourniture d'impulsions sur sa broche de commande. Ces impulsions devront être répétées à un intervalle de temps inférieur à 20ms. La largeur d'impulsion fournit l'angle de rotation du servomoteur par rapport à sa position centrale (0°) :

Une impulsion de 1,5 ms génère un angle de 0°

Une impulsion de 1ms génère un angle de -90°

Une impulsion de 2ms génère un angle de +90°



Nous voyons donc qu'en faisant varier la largeur d'impulsion entre 1 et 2 ms, nous pouvons obtenir n'importe quel angle de notre servomoteur sur une plage de 180°.

Faire varier une largeur d'impulsion, c'est exactement dans les possibilités de notre module PWM.

Souvenez-vous qu'il nous faudra fixer le temps de cycle, la largeur du niveau haut étant définie par le servomoteur :

Ce temps de cycle devra être inférieur à 20ms (imposé par la norme utilisée par les servomoteurs) et supérieur à 2ms (temps à l'état haut maximum).

Le temps entre 2 incrémentations de notre Timer 2 vaut : prédiviseur * Tcy. Ou prédiviseur * 4 * Tosc

Avec un prédiviseur de 1, nous pouvons obtenir une durée maximale de :

$$T_c = 1 * 4 * T_{osc} * 256 (PR2_{maxi} + 1) = 51,2 \mu s, \text{ ce qui est trop court.}$$

Avec un prédiviseur de 4, la durée maximale est de :

$$T_c = 4 * 4 * 50ns * 256 = 204 \mu s, \text{ ce qui est encore trop court}$$

Avec un prédiviseur de 16, la durée maximale est de :

$$T_c = 16 * 4 * 50ns * 256 = 819 \mu s, \text{ ce qui reste trop court.}$$

Et bien zut, ça ne marche pas. C'est dû au fait que notre postdiviseur est inactif pour le timer 2. Pourtant on voudrait bien pouvoir utiliser notre module PWM pour cette application. Reprenons donc notre formule :

$$T_c = \text{prédiviseur} * 4 * T_{osc} * (PR2+1).$$

Comme nous avons déjà un prédiviseur et un PR2 à leur valeur maximale, comment encore allonger TC maximum ? Tout simplement en augmentant T_{osc}.

T_{osc}, le temps de l'oscillateur, dépend du quartz utilisé pour cadencer notre PIC. Pour rappel : $T_{osc} = 1/F_{quartz}$.

Donc, en remplaçant, pour faire simple, notre quartz de 20Mhz, par celui de 4MHz que nous avons utilisé pour notre PIC 16F84, et que vous devez donc, en bonne logique, posséder, nous aurons :

$$T_{osc} = 1 / 4\text{Mhz} = 0,25\mu\text{s}.$$

Dans ces conditions, notre T_c maximum devient :

$$T_{c \text{ maxi}} = 16 * 4 * 0,25\mu\text{s} * 256 = 4096 \mu\text{s} = 4,096 \text{ ms}$$

Cette fois c'est bon. Notez que je vous informe déjà qu'avec ce type de signal, et les durées très longues mises en jeu, la méthode consistant à utiliser le « PWM » est loin d'être la meilleure. C'est pourquoi je vous proposerai une méthode bien plus efficace. Mais le but de cet exercice est de comprendre et de mettre en œuvre les CCP en mode « PWM » et en mode « compare avec trigger ».

Remplaçons donc dans notre montage le quartz de 20 Mhz par celui de 4Mhz.

Dans ce cas, nous mettrons la valeur 0xFF dans PR2, pour obtenir un temps TC égal à 2,176ms.

Notre valeur Th devra varier de 1 à 2 ms, ce qui correspondra à des valeurs de :

$$Th \text{ minimum} = 1000\mu\text{s}, \text{ donc } CCPR1H \text{ minimum} = 1000 / (16*4*0,25) = 62,5$$

$$Th \text{ maximum} = 2000\mu\text{s}, \text{ donc } CCPR1H \text{ maximum} = 2000 / (16*4*0,25) = 125$$

Donc, nous devons transformer notre valeur de potentiomètre, qui varie de 0 à 1023, en une valeur qui varie de 0 à 62,5 (donc, 250 valeurs différentes). Ceci nous permettra de placer dans CCPR1L la valeur résultant de :

$$CCPR1L = 62,5 + \text{valeur variable de 0 à 62,5 par pas de } 1/4$$

Notre potentiomètre permet, grâce à ses 10 bits, de générer 1024 valeurs différentes. En conservant la plus proche puissance de 2 disponible de ce dont nous avons besoin (250 valeurs), nous voyons que nous pouvons retenir 256 valeurs différentes.

Donc, nous conserverons 8 bits (256 valeurs) pour la valeur de notre potentiomètre, alors que nous avons besoin de 250 valeurs. Nous répartirons approximativement le surplus de part et d'autre des limites, ce qui nous donnera :

$$CCPR1L = 61,75 + (\text{valeur potentiomètre sur 8 bits} / 4)$$

Soit une valeur minimale de 61,75 et une valeur maximale de $61,75 + (255/4) = 125,5$

Notre temps T_h pourra donc varier de :

$$T_{h \min} = 61,75 * 16 * 4 * 0,25\mu s = 0,998 \text{ ms}$$

à

$$T_{h \max} = 125,5 * 16 * 4 * 0,25\mu s = 2,008 \text{ ms}$$

Nous avons donc bien généré notre signal qui varie de 1 à 2 ms.

Il faut maintenant se poser la question de la résolution (précision) obtenue. C'est très simple :

Dans l'intervalle utile, nous avons 250 valeurs intermédiaires, donc notre précision sera de

180 degrés de rotation / 250 = 0,72 degrés. Nous pourrions donc positionner notre servomoteur avec une précision de 0,72 degrés.

Ceci correspond à une variation minimale du temps de 1ms pour 180°, soit $(1\text{ms} / 180) * 0,72 = 4\mu s$.

Cette même précision, exprimée en %, nous donne : $1/250 = 0,4 \%$

Donc, pour résumer tout ceci, on peut dire :

- Nous mettrons 0xFF dans notre PR2, ce qui nous donne un temps T_c de 4,096ms
- Nous chargerons la valeur de notre potentiomètre, de 0 à 1023
- Nous conservons les 8 bits de poids fort (soit 6 bits entiers + 2 « décimales »)
- Nous ajoutons (en quarts de valeurs) $(61,75 * 4)$, soit 247
- Le résultat tient alors sur 10 bits (en réalité le bit 9 vaudra toujours 0 dans ce cas)
- Nous mettrons les 8 bits de poids fort comme partie entière dans CCPR1L
- Nous mettrons les 2 bits de poids faible (quarts) dans CCP1X et CCP1Y

Pour exemple, si notre potentiomètre vaut 0, on aura :

- Valeur sur 10 bits = 00
- On conserve les 8 bits de poids fort, soit B'00000000'
- On ajoute 247, on obtient donc D'247', soit B'0011110111' sur 10 bits
- On place B'00111101' dans CCPR1L
- On place B'11' dans CCP1X/CCP1Y

De la sorte, notre valeur minimale est de :

Partie entière = $B'00111101' = 61$

Partie fractionnaire (quarts) = $B'11'$ quarts = $\frac{3}{4} = 0,75$

Valeur complète = 61,75

Si notre potentiomètre vaut 1023, on aura :

- Valeur sur 10 bits = $D'1023' = B'1111111111'$
- On conserve les 8 bits de poids fort, soit $B'11111111'$, soit $D'255'$
- On ajoute $D'247'$, on obtient donc $D'502'$, soit $B'0111110110'$ sur 10 bits
- On place $B'01111101'$ dans CCP1L
- On place $B'10'$ dans CCP1X/CCP1Y

De la sorte, notre valeur maximale est de :

Partie entière = $B'01111101' = 125$

Partie fractionnaire = $B'10' = \frac{2}{4} = 0,5$

Valeur complète = 125,5

Je sais que j'ai insisté assez lourdement, et que j'ai effectué de nombreuses répétitions, mais j'ai préféré passer pour un radoteur plutôt que de vous laisser dans l'hésitation concernant certaines procédures. Que ceux qui ont compris « du premier coup » veuillent bien m'excuser.

Nous avons parfaitement réglé le problème de notre module CCP1, configuré en « PWM », occupons-nous maintenant de notre module CCP2, configuré en « compare avec trigger ».

Après le temps Tacq, la conversion A/D démarrera automatiquement (du fait du trigger), et générera une interruption après un temps de 12 Tad.

Grâce au tableau que je vous ai fourni dans le chapitre sur le convertisseur A/D, vous constatez que, pour un PIC à 4Mhz, vous devez choisir un prédiviseur du convertisseur = 8, ce qui vous donne un temps Tad = 2µs. Le temps d'échantillonnage prendra donc $12 * 2\mu s = 24 \mu s$.

Le temps Tacq est de minimum 20µs, si on ne veut pas « se casser la tête » à calculer.

Donc, une conversion complète, avec son acquisition, durera donc $20\mu s + 24\mu s = 44 \mu s$.

Nous avons besoin de la valeur du potentiomètre au maximum une fois par cycle « PWM ». Inutile en effet de remettre à jour plusieurs fois CCP1L durant le même cycle, il n'est de toute façon transféré dans CCP1H qu'à la fin du cycle en cours.

A cela il faut ajouter 2 TAD avant l'acquisition suivante, soit $44\mu s + 4\mu s = 48 \mu s$ séparant au minimum 2 acquisitions.

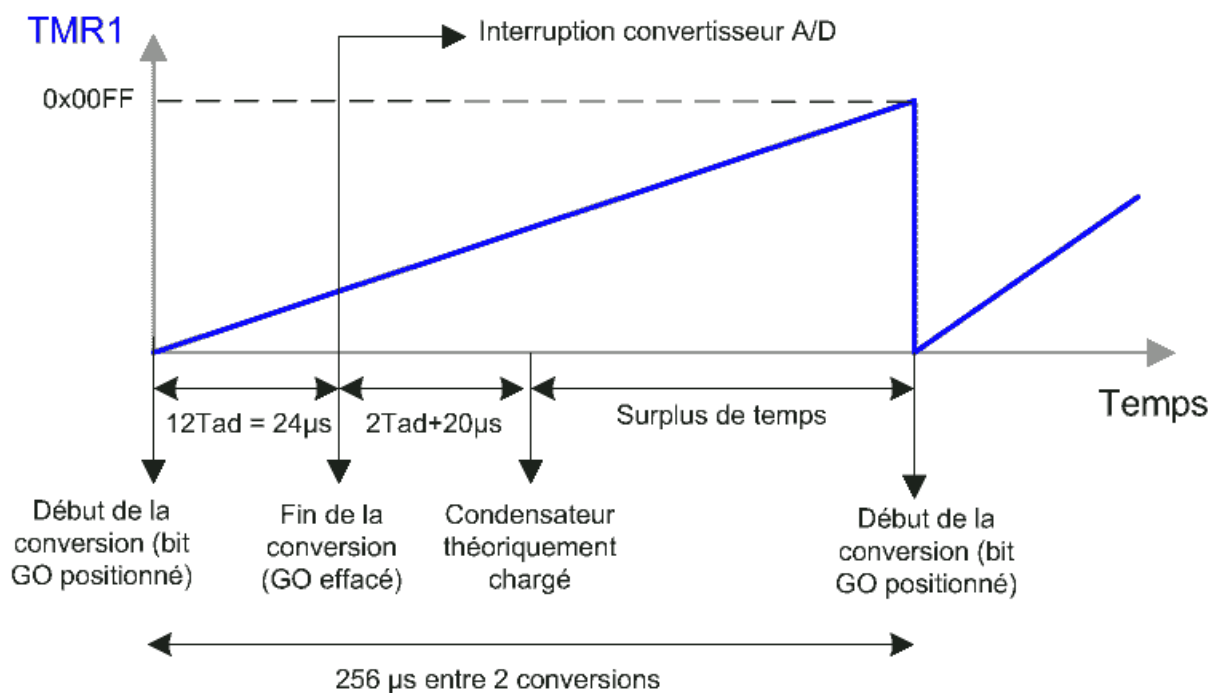
Ce cycle « PWM » dure 4096µs, ce qui nous laisse le temps d'effectuer : $4096 / 48 = 85$ mesures dans l'intervalle.

Nous allons profiter de cette opportunité, tout en nous montrant plus modeste, en effectuant 16 mesures dans cet intervalle. De cette façon, nous pourrions limiter l'influence d'un parasite. Ceci nous donne un temps séparant 2 échantillonnages de $4096 \mu s / 16 = 256 \mu s$.

Comme notre temps d'échantillonnage est fixe (12 TAD), nous allons allonger le temps séparant le début de l'acquisition du début de la conversion. Ceci revient en fait à allonger le temps d'acquisition.

Ce nouveau temps sera donc de : $256 \mu s - 24 \mu s = 232 \mu s$. Le temps séparant 2 démarrages d'échantillonnage successifs sera lui de $256 \mu s$, puisque durant le temps de conversion, le timer1 continue de compter.

Je vous donne la chronologie des événements avec ce petit graphique :



Vous voyez qu'une fois lancé, vous ne vous occupez plus de rien, vous serez prévenu par une interruption chaque fois qu'une nouvelle valeur du potentiomètre sera disponible.

Nous allons donc configurer notre timer 1 avec un temps de débordement de $256 \mu s$.

Nous allons utiliser le temps de charge de notre condensateur (temps d'acquisition) T_{acq} de $220 \mu s$. Nous placerons donc dans CCPR1H/CCPR1L la valeur 16 bits tirée de la formule :

$$\text{Temps de cycle du timer 1} = (CCPR2HL + 1) * 4 * T_{osc} * \text{prédiviseur} = 256 \mu s$$

Soit, avec un prédiviseur de 1 :

$$CCPR2HL = (256 / (4 * T_{osc}) - 1 = 255 \quad (\text{ne pas oublier que notre quartz est maintenant à } 4\text{Mhz}).$$

Donc, $D'255' = B'0000000011111111'$ sur 16 bits. Nous placerons donc :

CCPR2H = B'00000000' = 0x00
CCPR2L = B'11111111' = 0xFF

Pour ceux qui n'ont pas suivi l'astuce, la méthode normale aurait été de dire :

- Je configure CCP1 pour une durée de Tacq de $2T_{ad} + 20 \mu s$
- La conversion est terminée, j'attends $256 \mu s - 48 \mu s$
- Je reconfigure CCP1 pour une durée de Tacq de $2T_{ad} + 20 \mu s$
- La conversion est terminée, j'attends $256 \mu s - 48 \mu s$
- Etc.

Ceci nécessitait 2 mesures de temps distinctes. J'ai donc simplifié en :

- Je configure CCP1 avec $256 \mu s$. On aura donc une conversion toutes les $256 \mu s$
- La conversion est terminée en $12T_{ad}$, ce qui nous laisse $256 \mu s - 12T_{ad} = 232 \mu s$ comme Tacq. Ce temps est supérieur aux $(20 \mu s + 2T_{ad})$ minimum requis, donc aucun problème.

Le temps total est donc identique ($256 \mu s$), la fréquence d'échantillonnage également, qui est suffisante pour obtenir 16 mesures du potentiomètre entre 2 modifications du « PWM ». Mais on n'a plus besoin de la boucle de temporisation supplémentaire.

Une fois tout configuré, notre module CCP1 fonctionnera donc de la façon suivante, dans la routine d'interruption du convertisseur A/D :

- On sauve la valeur lue

Ben oui, c'est tout. En effet, à intervalle de $256 \mu s$, une nouvelle conversion sera automatiquement lancée, nous n'avons plus à nous occuper que de sauver les valeurs. Nous sommes prévenu de la fin d'une conversion par une interruption ADIF.

Nous obtenons donc, entre 2 mises à jour de « PWM », 16 valeurs dont il nous suffira de tirer la moyenne. Cette moyenne devra donc être écrite dans nos paramètres du module « PWM » comme expliqué plus haut. Quand allons-nous écrire cette valeur ?

En fait, nous avons 2 possibilités ;

- Soit nous mettons « PWM » à jour toutes les 16 mesures du potentiomètre (dans ce cas, cette mise à jour se fera dans la routine d'interruption A/D
- Soit nous mettons « PWM » à jour une fois par cycle T_c , et donc à chaque interruption de débordement du timer 2.

Nous n'avons que l'embarra du choix. Je choisis la seconde solution, qui est la bonne. En effet, comme nous modifierons les valeurs au début du cycle, nous sommes certains de ne pas obtenir de fonctionnement aléatoire puisque la copie de notre valeur de consigne sur 10 bits s'effectuera au début de l'interruption suivante. Nous sommes donc certains que cette copie n'interviendra pas entre le moment où nous modifions nos 2 bits fractionnaires et le moment où nous modifions nos 8 autres bits.

Notre routine d'interruption du timer 2 va donc :

- Calculer la moyenne des 16 mesures du potentiomètre sauvées
- Entrer cette moyenne comme paramètre du module « PWM ».

J'en profite pour faire une petite remarque. J'en vois d'ici qui se disent : « On est dans un exercice pratique, et Bigonoff fait toujours de la théorie et des mathématiques, il n'y a pas moyen de s'en passer ? ».

La réponse est malheureusement non. En effet, je n'ai pas donné toutes ces formules uniquement pour faire joli, et pour ne les utiliser que dans la théorie. Tout ce qui concerne les mesures de temps, conversion etc. nécessitent de préalablement calculer toutes les valeurs de consignes dont vous aurez besoin.

Il n'y a malheureusement pas de miracle. En électronique « traditionnelle, il fallait calculer des constantes de temps avec des condensateurs, des résistances, et autres composants. En électronique programmable, on calcule également, mais simplement en employant des formules différentes. Les contraintes restent identiques.

Mais vous allez voir que toute cette théorie, qui semble ne plus finir, se traduit finalement par un programme ultra-simple.

Faites un copier/coller de votre fichier « m16f876.asm » et renommez la copie en « servo1.asm ». Créez un nouveau projet qui utilise ce fichier.

Commençons, comme toujours, par éditer l'en-tête :

```
;*****
; Exercice sur l'utilisation des modules CCP.
; Pilotage d'un servomoteur en utilisant le module CCP1 en mode PWM
; et le module CCP2 en mode compare avec trigger
;
;*****
;
; NOM: Servo1
; Date: 06/06/2002
; Version: 1.0
; Circuit: Platine d'expérimentation
; Auteur: Bigonoff
;
;*****
;
; Fichier requis: P16F876.inc
;
;*****
;
; Le PIC DOIT être cadencé avec un quartz de 4Mhz
; L'entrée de commande du servo est connectée sur CCP1/RC2
; Le potentiomètre de réglage de position du servo est connecté sur AN0
;
;*****

LIST p=16F876 ; Définition de processeur
#include <p16F876.inc> ; fichier include
```

```

__CONFIG __CP_OFF & __DEBUG_OFF & __WRT_ENABLE_OFF & __CPD_OFF & __LVP_OFF &
__BODEN_ON & __PWRTE_ON & __WDT_ON & __HS_OSC

; __CP_OFF          Pas de protection
; __DEBUG_OFF       RB6 et RB7 en utilisation normale
; __WRT_ENABLE_OFF  Le programme ne peut pas écrire dans la flash
; __CPD_OFF         Mémoire EEprom déprotégée
; __LVP_OFF         RB3 en utilisation normale
; __BODEN_ON        Reset tension en service
; __PWRTE_OFF       Démarrage rapide
; __PWRTE_ON        Démarrage temporisé
; __WDT_ON          Watchdog en service
; __HS_OSC          Oscillateur haute vitesse (4Mhz<F<20Mhz)

```

Remarquez que qui dit servomoteurs, dit en général fonctionnement sur piles ou accumulateurs. J'ai donc actionné le bit de configuration BODEN_ON qui provoque le reset du PIC en cas de tension d'alimentation insuffisante. A vous donc, dans une utilisation pratique, de gérer ces situations.

Nous trouvons ensuite les assignations système :

```

;*****
;                                ASSIGNATIONS SYSTEME                                *
;*****

; REGISTRE OPTION_REG (configuration)
; -----
OPTIONVAL EQU B'10000000' ; RBPU b7 : 1= Résistance rappel +5V hors service

; REGISTRE INTCON (contrôle interruptions standard)
; -----
INTCONVAL EQU B'01000000' ; PEIE b6 : masque autorisation générale périphériques

; REGISTRE PIE1 (contrôle interruptions périphériques)
; -----
PIE1VAL EQU B'01000010' ; ADIE b6 : masque interrupt convertisseur A/D
; TMR2IE b1 : masque interrupt TMR2 = PR2

; REGISTRE ADCON1 (ANALOGIQUE/DIGITAL)
; -----
ADCON1VAL EQU B'00001110' ; 1 entrée analogique, justification à gauche

```

Pas grand-chose de particulier à dire ici. Comme nous gardons 8 bits pour le convertisseur, nous alignerons à gauche, en n'utilisant qu'une seule entrée analogique. Nos 2 sources d'interruptions AD et TRM2 sont validées.

Ensuite, nous définissons les constantes qui vont être utilisées dans notre programme :

```

;*****
;                                ASSIGNATIONS PROGRAMME                                *
;*****

OFFSET EQU D'247' ; valeur mini de CCPRx en quarts de valeurs
PR2VAL EQU 0xFF ; valeur d'initialisation de PR2
CCPR2VAL EQU 0x00FF ; temps Tacq pour module CCP2

```

Suivies par nos macros :

```

;*****
;
;                               MACRO                               *
;*****

; Changement de banques
; -----

BANK0 macro                ; passer en banque0
    bcf    STATUS,RP0
    bcf    STATUS,RP1
endm

BANK1 macro                ; passer en banque1
    bsf    STATUS,RP0
    bcf    STATUS,RP1
endm

```

Nos variables en banque 1 se limitent à la zone de stockage des 16 valeurs analogiques lues, à un compteur qui indique l'emplacement, parmi les 16 disponibles à utiliser pour la prochaine lecture, et une variable sur 16 bits destinée à recevoir les calculs concernant les consignes PWM

```

;*****
;                               VARIABLES BANQUE 0                     *
;*****

; Zone de 80 bytes
; -----

CBLOCK 0x20                ; Début de la zone (0x20 à 0x6F)
zoneval    : 0x10          ; 16 emplacements de sauvegarde
numval     : 1             ; numéro de l'emplacement de sauvegarde
consigne   : 2             ; calcul de la consigne
ENDC       ; Fin de la zone

```

En zone commune, nous avons nos habituelles variables de sauvegarde pour les routines d'interruption. Notez que notre programme principal ne fait rien. Mais j'ai conservé quelques sauvegardes pour que vous puissiez améliorer cet exemple pour en faire un programme opérationnel.

```

;*****
;                               VARIABLES ZONE COMMUNE                 *
;*****

; Zone de 16 bytes
; -----

CBLOCK 0x70                ; Début de la zone (0x70 à 0x7F)
w_temp     : 1             ; Sauvegarde registre W
status_temp : 1           ; sauvegarde registre STATUS
ENDC

```

La routine d'interruption principale contient les tests pour les 2 interruptions utilisées. Notez que, cette fois, pour varier les plaisirs, j'ai enlevé les instructions « goto restoreg », ce qui induit que si 2 interruptions simultanées se produisent, elles seront traitées en même temps. Ceci pour vous montrer qu'il y a plusieurs façons de procéder.

```

;*****
;
;                               DEMARRAGE SUR RESET                               *
;*****

    org 0x000      ; Adresse de départ après reset
    goto init      ; Initialiser

; //////////////////////////////////////
;                               I N T E R R U P T I O N S                               *
; //////////////////////////////////////

;*****
;
;                               ROUTINE INTERRUPTION                               *
;*****

;sauvegarder registres
;-----
org 0x004          ; adresse d'interruption
movwf w_temp       ; sauver registre W
swapf STATUS,w     ; swap status avec résultat dans w
movwf status_temp  ; sauver status swappé
BANK0              ; passer en banque0

; Interruption convertisseur A/D
; -----
btfss PIR1,ADIF    ; tester si interrupt en cours
goto intsw1        ; non sauter
call intad         ; oui, traiter interrupt
bcf PIR1,ADIF      ; effacer flag interrupt

; Interruption TMR2
; -----
intsw1
btfss PIR1,TMR2IF  ; tester si interrupt en cours
goto restorereg    ; non, fin d'interrupt
call inttmr2       ; oui, traiter interrupt
bcf PIR1,TMR2IF    ; effacer flag interrupt

;restaurer registres
;-----
restorereg
swapf status_temp,w ; swap ancien status, résultat dans w
movwf STATUS        ; restaurer status
swapf w_temp,f      ; Inversion L et H de l'ancien W
swapf w_temp,w      ; Réinversion de L et H dans W
retfie              ; return from interrupt

```

La routine d'interruption du convertisseur A/D est toute simple, puisqu'elle se contente de sauvegarder la valeur lue dans l'emplacement désigné, et d'incrémenter ce pointeur d'emplacement.

```

;*****
;
;                               INTERRUPTION CONVERTISSEUR A/D                               *
;*****
;-----
; Sauvegarde la valeur du potentiomètre sur 8 bits dans un des 16 emplacements
; prévus. numval contient le numéro de l'emplacement
; Relance une nouvelle conversion, avec Tacq = 232µs, ce qui permet au moins
; 16 mesures durant le temps de cycle du module PWM (4096 µs)
; -----

```

```

intad
    movf    numval,w        ; numéro de l'emplacement de sauvegarde
    andlw   0x0F            ; garder valeurs de 0 à 15
    addlw   zoneval         ; ajouter emplacement début zone
    movwf   FSR             ; dans pointeur
    movf    ADRESH,w        ; charger 8 bits forts du convertisseur A/D
    movwf   INDF            ; sauvegarder valeur
    incf    numval,f        ; incrémenter numéro de valeur
    return                    ; fin d'interruption

```

La routine d'interruption du timer 2 permet le calcul de la moyenne des 16 valeurs précédemment sauvegardées, et la déduction de la nouvelle valeur de réglage du PWM.

```

;*****
;                               INTERRUPTION TIMER 2                               *
;*****
;-----
; Calcule la moyenne des 16 valeurs du potentiomètre, ajoute l'offset, et
; règle le module PWM en conséquence
; 1) on ajoute les 16 valeurs, on obtient donc 16 * valeur moyenne, donc
;   résultat sur 12 bits
; 2) On divise résultat par 16, donc on obtient la valeur moyenne sur 8 bits
; 3) On ajoute l'offset exprimé en 1/4 de valeurs, le résultat tient sur 9 bits
; 4) On sauve les 2 bits faibles dans CCP1X et CCP1Y
; 5) on divise les 9 bits par 4 pour obtenir la valeur entière et on sauve les
;   8 bits 0xxxxxxx obtenus dans CCP1L
;-----
inttmr2
    ; faire la somme des 16 valeurs
    ; -----
    movlw   zoneval         ; adresse de début de la zone
    movwf   FSR             ; dans pointeur
    clrf    consigne        ; efface poids fort du résultat
    clrf    consigne+1      ; idem poids faible
inttmr2l
    movf    INDF,w          ; charger valeur pointée
    addwf   consigne+1,f    ; ajouter à poids faible résultat
    btfsc   STATUS,C        ; tester si débordement
    incf    consigne,f      ; oui, incrémenter poids fort
    incf    FSR,f           ; incrémenter pointeur
    btfss   FSR,4           ; tester si terminé (FSR = 0x30)
    goto    inttmr2l        ; non, valeur suivante

    ; division du résultat par 16
    ; -----
    rrf     consigne,f       ; poids fort / 2, inutile d'effacer carry
                                ; car b7/b4 seront inutilisés
    rrf     consigne+1,f     ; poids faible divisé par 2 avec b7=carry
    rrf     consigne,f       ; poids fort / 2, inutile d'effacer carry
    rrf     consigne+1,f     ; poids faible divisé par 2 avec b7=carry
    rrf     consigne,f       ; poids fort / 2, inutile d'effacer carry
    rrf     consigne+1,f     ; poids faible divisé par 2 avec b7=carry
    rrf     consigne,f       ; poids fort / 2, inutile d'effacer carry
    rrf     consigne+1,f     ; poids faible divisé par 2 avec b7=carry

    ; ajouter la valeur minimale (offset)
    ; -----
    movlw   OFFSET          ; charger offset (ici, il tient sur 8 bits)
    addwf   consigne+1,f    ; ajouter à poids faible
    btfsc   STATUS,C        ; tester si débordement
    incf    consigne,f      ; oui, incrémenter poids fort

```



```

; sauver les 2 bits fractionnaires
; -----
bcf    CCP1CON,CCP1X    ; effacer bit 1
bcf    CCP1CON,CCP1Y    ; effacer bit 0
btfsc  consigne+1,1     ; tester si futur bit "-1" = 1
bsf    CCP1CON,CCP1X    ; oui, mettre bit 1 à 1
btfsc  consigne+1,0     ; tester si futur bit "-2" = 1
bsf    CCP1CON,CCP1Y    ; oui, mettre bit 0 à 1

; placer valeur entière sur 8 bits
; -----
rrf    consigne,f       ; récupérer futur b6 dans carry
rrf    consigne+1,f     ; diviser par 2 avec entrée de b6 dans b7
rrf    consigne,f       ; récupérer futur b7 dans carry
rrf    consigne+1,w     ; diviser par 2, b6 et b7 en place
movwf  CCP1L1          ; placer nouvelle valeur de consigne
return                               ; fin d'interruption

```

Rien de bien compliqué, donc, il s'agit tout simplement de l'application des formules précédemment expliquées.

Vient le tour de notre routine d'initialisation. C'est ici que tout se passe, en fait, puisque tout sera automatique par la suite.

J'ai volontairement séparé nettement les différentes étapes, afin que vous puissiez voir d'un coup d'œil les différentes fonctions initialisées.

```

; //////////////////////////////////////
;                               P R O G R A M M E
; //////////////////////////////////////
;*****
;                               INITIALISATIONS
;*****
init
; initialisation PORTS (banque 0 et 1)
; -----
BANK1          ; sélectionner banque1
Bcf    TRISC,2  ; CCP1/RC2 en sortie

; Registre d'options (banque 1)
; -----
movlw  OPTIONVAL    ; charger masque
movwf  OPTION_REG   ; initialiser registre option

; registres interruptions (banque 1)
; -----
movlw  INTCONVAL    ; charger valeur registre interruption
movwf  INTCON       ; initialiser interruptions
movlw  PIE1VAL      ; Initialiser registre
movwf  PIE1         ; interruptions périphériques 1

; configurer le module CCP1
; -----
bcf    STATUS,RP0   ; passer banque 0
movlw  B'00001100'  ; pour mode PWM
movwf  CCP1CON      ; dans registre de commande CCP
movlw  PR2VAL       ; valeur de débordement

```

```

bsf    STATUS,RP0      ; passer en banque 1
movwf  PR2              ; dans registre de comparaison
bcf    STATUS,RP0      ; repasser en banque 0
movlw  B'00000110'     ; timer 2 on, prédiviseur = 16
movwf  T2CON            ; dans registre de contrôle

    ; configurer le convertisseur A/D
    ; -----
movlw  ADCON1VAL        ; 1 entrée analogique
bsf    STATUS,RP0      ; passer banque 1
movwf  ADCON1           ; écriture dans contrôle1 A/D
bcf    STATUS,RP0      ; repasser banque 0
movlw  B'01000001'     ; convertisseur ON, prédiviseur 8
movwf  ADCON0           ; dans registre de contrôle0

    ; configurer le module CCP2
    ; -----
movlw  B'00001011'     ; pour mode compare avec trigger
movwf  CCP2CON          ; dans registre commande CCP2
movlw  high CCPR2VAL    ; charger poids fort valeur de comparaison
movwf  CCPR2H           ; dans registre poids fort
movlw  low CCPR2VAL     ; charger poids faible valeur de comparaison
movwf  CCPR2L           ; dans registre poids faible
bsf    T1CON,TMR1ON     ; lancer le timer 1

    ; autoriser interruptions (banque 0)
    ; -----
clrf   PIR1             ; effacer flags 1
clrf   PIR2             ; effacer flags 2
bsf    INTCON,GIE       ; valider interruptions

```

Remarquez que nous n'avons pas d'initialisation de variables. Ceci peut se comprendre. En effet :

Concernant le pointeur d'emplacement, on garde dans la routine d'interruption, les 4 bits de poids faible. Donc, inutile d'initialiser, car, commencer à sauvegarder à un emplacement ou à un autre n'a aucune importance, puisqu'on aura sauvegardé de toute façon dans les 16 emplacements avant leur utilisation.

La zone des valeurs sauvegardées sera remplie par la routine d'interruption AD avant qu'on n'utilise ces valeurs dans la routine d'interruption TMR2. Donc, inutile non plus d'initialiser.

Le résultat « consigne » est initialisé avant chaque utilisation, et donc, une fois de plus, inutile de le configurer.

Ne reste que notre programme principal, qui, le pauvre, n'a rien d'autre à faire qu'à se tourner les pouces.

Ceci pour vous dire que la génération de notre signal PWM vous laisse énormément de temps pour gérer autre chose. Imaginez que vous utilisez une lecture de potentiomètre et la génération d'un signal PWM, et que tout ceci se réalise de façon pratiquement transparente pour votre programme principal.

Lancez l'assemblage, placez le fichier « servo1.hex » dans votre PIC, n'oubliez pas de changer le quartz, et lancez l'application. Votre potentiomètre vous permet maintenant de régler un beau signal PWM avec un état haut d'une durée de 1 à 2 ms.

Votre servomoteur, pour peu qu'il soit compatible avec la norme et qu'il fonctionne sous 5V, sera positionné en suivant les évolutions de votre potentiomètre. Attention cependant à la consommation de la pin d'entrée de votre servomoteur (cela ne devrait cependant pas poser de problème).

20.8 Exercice 2 : une méthode plus adaptée

Nous avons vu que des temps très longs comme ceux requis par la commande d'un servomoteur n'étaient pas très appropriés à l'échelle de temps du module « PWM ». Ce dernier est en effet prévu pour tourner bien plus vite. Il permet de plus de disposer de temps « haut » et « bas » de durée très précise.

Si nous regardons les chronogrammes nécessaires pour le pilotage de notre servomoteur, nous voyons que nous avons besoin de 2 temps différents :

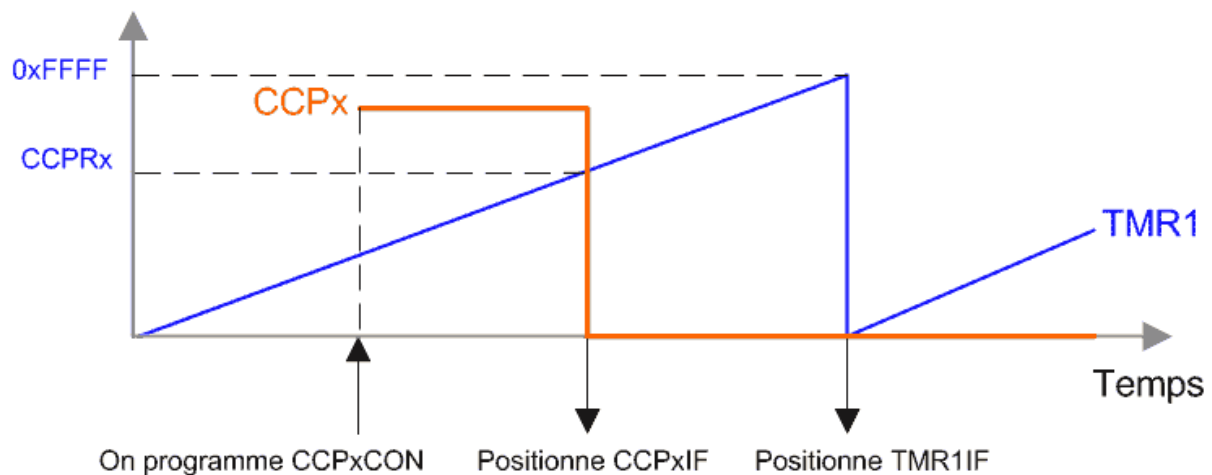
- Le temps de maintien à l'état haut du signal, entre 1 et 2 ms. Ce temps est critique et demande une grande précision, car c'est lui qui détermine la position de notre servomoteur
- Le temps total de cycle, qui lui n'intervient en aucune façon sur cette position, il réclame seulement d'être inférieur à 20ms. Ce temps n'est donc absolument pas critique.

Donc, on peut dire que pour piloter notre servo, on doit :

- Placer un signal à l'état haut durant une période très précise
- Attendre un temps non critique avant de recommencer.

Ceci nous amène tout naturellement à utiliser pour cette application le module CCP1 en mode « compare avec gestion de la pin CCP1 »

Souvenez-vous en effet, que la combinaison « 1001 » des bits CCPxM3 à CCPxM0 induit le fonctionnement suivant :



La largeur de l'état haut dépend très précisément du timer 1 et de CCPR. Aucun retard ou perturbation logiciel n'intervient. Il nous suffit donc de reprogrammer CCP1CON à intervalle de moins de 20ms pour générer notre signal de façon parfaite.

Par contre, nous ne devons pas oublier que notre timer 1 est également utilisé dans notre module CCP2, utilisé, lui, en mode « trigger ». Nous avons donc des contraintes d'utilisation du fait de l'utilisation des 2 modules CCP.

Il nous suffit de réfléchir pour constater que, dans ce cas, le reset de notre timer 1 ne pourra intervenir avant que la valeur CCPR1HL ne soit atteinte. Dans le cas contraire, notre ligne CCP1 ne pourrait jamais repasser à « 0 ».

Ceci implique que notre temps séparant 2 échantillonnages ne pourra être inférieur à, dans le pire des cas, 2ms. Sachant que 20ms séparent au maximum 2 impulsions, cela nous permet de réaliser 10 mesures de notre potentiomètre. Le chronogramme vous est donné plus bas.

Nous aurions pu également décider d'effectuer les conversions A/D au rythme déterminé par un autre timer (tmr0 ou tmr2). Ceci nous délivrait de cette contrainte, sous condition de lancer nous-mêmes la conversion A/D (pas d'utilisation du module CCP2 en mode trigger). La méthode de conversion étant alors celle utilisée dans notre exercice « lum2 ».

Bien entendu, il s'agit ici d'un exercice d'application des modules CCP. Je vais donc choisir la première solution.

Pour résumer, nous choisirons donc une valeur de comparaison CCPR1HL variable, qui nous donnera, en fonction de la valeur de notre potentiomètre, un temps variable entre 1 et 2 ms.

Nous choisirons de fait une valeur de CCPR2HL fixe, qui nous fixera le temps séparant 2 conversions, en étant conscient que cette valeur CCPR2HL devra être impérativement plus grande que la valeur CCPR1HL.

Ceci nous donnera donc un temps au minimum de 2 ms. Nous allons choisir 2,4ms, et 8 mesures de conversion A/D intermédiaires ce qui nous donnera un temps approximatif séparant 2 impulsions de $2,4 * 8 = 19,2\text{ms}$.

Nous utiliserons pour cet exercice notre quartz habituel de 20Mhz.

Notre timer1, utilisé avec un prédiviseur de 1, nous permet des mesures de temps de 0,2µs (1 cycle) à 13,1072 ms (65536 cycles). Nous aurons besoin de temps compris entre 1 à 2,4 ms, ce qui est parfaitement dans les possibilités de notre timer 1. Vous voyez que les durées mises en œuvre et le type de signal sont plus adaptées à cette façon de procéder.

La durée de l'impulsion répond à la formule suivante (J'appelle CCPR1HL la valeur 16 bits formée par CCPR1H/CCPR1L, de même TMR1HL la valeur formée par la concaténation de TMR1H et de TMR1L):

$$T = (CCPR1HL + 1) * T_{cy} * \text{prédiviseur}$$

Autrement dit :

$$CCPR1HL = (T / (T_{cy} * \text{prédiviseur})) - 1$$

Une durée de 1ms se traduira donc par une valeur de :

$$CCPR1HL \text{ minimum} = (1\text{ms} / 0,2\mu\text{s}) - 1 = 4999$$

Une durée de 2 ms réclamera une valeur de :

$$CCPR1HL \text{ maximum} = (2\text{ms} / 0,2\mu\text{s}) - 1 = 9999$$

La durée séparant 2 conversions A/D répond à la même formule, mais relative à CCPR2HL :

$$CCPR2HL = (2,4\text{ms} / 0,2\mu\text{s}) - 1 = 11999$$

Nous allons donc exécuter les procédures suivantes (après initialisation) :

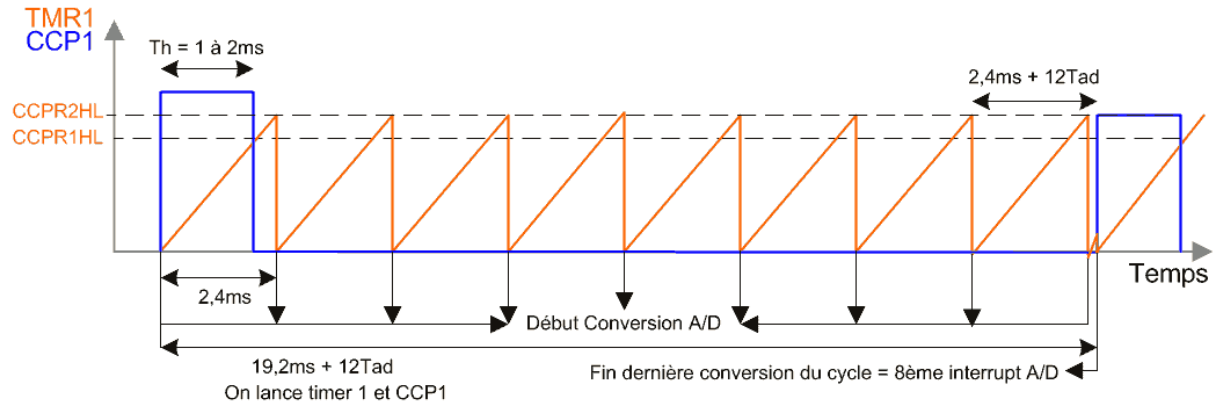
- On lance le timer 1 et on lance le module CCP1 (début de l'impulsion)
- Après le temps dépendant de CCPR1HL, on obtient la fin de l'impulsion
- On aura ensuite 8 interruptions A/D durant lesquelles on mesure le potentiomètre
- Lors de la 8^{ème} interruption, on mesure la moyenne, on calcule une nouvelle valeur de consigne CCPR1HL, on arrête et on remet à 0 le timer 1.
- On recommence les opérations

Notez que l'interruption intervient à la fin de la conversion A/D, alors que le reset du timer 1 (trigger) annonce la fin du temps d'acquisition Tacq et donc le début de la conversion.

Comme cette conversion dure 12 Tad, soit 19,2µs avec notre quartz de 20Mhz, le début de l'impulsion suivante se fera donc 12 Tad après le début de la 8^{ème} conversion. Donc un temps total séparant 2 impulsions de $(8 * 2,4\text{ms}) + 19,2\mu\text{s} = 19,2192 \text{ ms}$.

Si on veut être précis, on doit encore ajouter le temps entre le début de l'interruption, et le redémarrage par logiciel du timer 1. Le chronogramme suivant n'en tient pas compte. L'important dans cette application est que nous restions sous le temps total de 20ms.

Nous aurons donc le chronogramme simplifié suivant :



Juste un petit mot d'explications. sur notre chronogramme, chaque flèche pointée vers le bas indique un des débuts de conversion A/D comprise entre chaque impulsion. Le dernier début de conversion s'effectue après 19,2ms.

$12 * T_{ad}$ plus tard ($19,2\mu\text{s}$), la fin de cette conversion a lieu. A ce moment, on resette tmr1 , et on relance une nouvelle impulsion (qui intervient donc un encore un peu plus tard, le temps d'exécuter les quelques instructions nécessaires).

Reste un petit détail à mettre au point. En effet, on doit simultanément mettre CCP1 et TMR1 en service pour lancer l'impulsion, ce qui est impossible. On va donc devoir, soit :

- Mettre TMR1 en service, puis CCP1, ce qui implique que lorsque CCP1 passe à « 1 », TMR1 a déjà compté un cycle
- Soit mettre CCP1, puis TMR1 en service, ce qui implique que quand TMR1 commencera à compter, il y aura déjà un cycle que CCP1 sera à « 1 ».

En fait, cela n'a aucune importance pour cette application, mais il est bon que vous preniez garde à ce phénomène, pour le cas où votre application nécessiterait un créneau fixe d'une grande précision.

De toute façon, l'anomalie de fonctionnement expliquée dans la partie théorique va nous imposer une façon pratique de faire légèrement différente.

Voyons maintenant ce qui se passe au niveau du potentiomètre. Nous avons une valeur comprise entre 0 et 1023.

Or, nous avons besoin d'une valeur comprise, pour CCPR1HL entre 4999 et 9999. Ceci nous donne une plage de variation de 5000.

Nous allons donc décider de multiplier la valeur de notre potentiomètre par 5, ce qui nous donnera des valeurs comprises entre 0 et 5115. Nous répartirons le « surplus » de part et d'autre des valeurs limites, ce qui nous donne :

$$\begin{aligned}\text{Valeur minimale de CCPR1HL} &= 4999 - ((5115-5000)/2) = 4942 \\ \text{Valeur maximale de CCPR1HL} &= 4942 + 5515 = 10057.\end{aligned}$$

Ceci nous donnera un temps compris entre :

$$T = (\text{CCPR1HL}+1) * T_{cy} * \text{prédiviseur}$$

$$T_{\text{minimum}} = 4943 * 0,2\mu\text{s} = 988,6\mu\text{s} = 0,9886 \text{ ms}$$

$$T_{\text{maximum}} = 10058 * 0,2 = 2011,6\mu\text{s} = 2,0116 \text{ ms}$$

Nous avons donc bien réalisé nos objectifs. La valeur à placer dans CCPR1HL sera donc :

$$\text{CCPR1HL} = (\text{potentiomètre} * 5) + 4942$$

Nous constatons que nous conservons l'intégralité des 10 bits de la conversion A/D. Ceci nous donne dans ce cas 1000 valeurs utiles intermédiaires, soit une précision de :

$$\text{Précision relative} = 1/1000 = 0,1\%$$

Ceci nous permet de régler notre angle avec une précision de :

$$\text{Précision angulaire} = 0,1\% * 180 \text{ degrés} = 0,18 \text{ degré}.$$

Voyez la précision obtenue. Et encore, il faut savoir que, si vous avez suivi, cette précision est celle de notre convertisseur A/D, donc de notre potentiomètre. Vous pourriez très bien régler le servomoteur d'après une autre référence (consigne externe,...), ce qui vous permettrait d'avoir une précision de 1 cycle d'instruction (1Tcy), soit 0,2μs. La précision serait alors de :

$$1/5000 = 0,02\%$$

Vous pourriez donc positionner votre servomoteur avec une précision théorique de 0,036 degré. Ceci vous montre les possibilités de votre PIC.

Il nous reste à déterminer comment effectuer la multiplication par 5. L'idéal est de se servir de puissances de 2, afin d'éviter de se servir de programmes de multiplication (longs).

Rien de plus simple : Multiplier un nombre par 5 revient à le multiplier par 4, puis à ajouter le résultat au nombre initial. Les multiplications par 4 sont constituées de décalages, quant aux additions, elles sont gérées par le PIC.

Si on se souvient que nous avons effectué 8 mesures intermédiaires de notre potentiomètre, la somme de ces 8 valeurs nous donne 8 fois la valeur moyenne de notre potentiomètre.

Donc, pour obtenir 4 fois la valeur de notre potentiomètre, nous diviserons cette somme par 2. Reste à ajouter la valeur unitaire, valeur obtenue en divisant de nouveau la somme par 4 (division totale par 8) Ceci nous donne l'algorithme suivant (avec pot = valeur du potentiomètre):

- On additionne les 8 valeurs de mesure (on obtient $8 * \text{pot}$)
- On décale le résultat vers la droite (division par 2), et on sauve le résultat ($4 * \text{pot}$)
- On décale encore de 2 rangs vers la droite (division par 8). Résultat = pot
- On ajoute la valeur obtenue à la valeur précédemment sauvegardée ($4 * \text{pot} + \text{pot} = 5 * \text{pot}$).

Il est temps, maintenant, de passer à la réalisation de notre programme. Faites un copier/coller de votre fichier « m16F876.asm » et renommez cette copie « servo2.asm ».

Commençons par éditer l'en-tête et la configuration :

```

;*****
; Exercice sur l'utilisation des modules CCP. *
; Pilotage d'un servomoteur en utilisant le module CCP1 en mode *
; "compare avec sortie sur CCP1" et le module CCP2 en mode *
; "compare avec trigger" *
; *
;*****
; *
; NOM: servo2 *
; Date: 10/06/2002 *
; Version: 1.0 *
; Circuit: Platine d'expérimentation *
; Auteur: Bigonoff *
; *
;*****
; *
; Fichier requis: P16F876.inc *
; *
;*****
; *
; Le PIC est cadencé à 20 Mhz *
; L'entrée de commande du servo est connectée sur CCP1/RC2 *
; Le potentiomètre de réglage de position du servo est connecté sur AN0 *
; *
;*****
LIST p=16F876 ; Définition de processeur
#include <p16F876.inc> ; fichier include

__CONFIG __CP_OFF & __DEBUG_OFF & __WRT_ENABLE_OFF & __CPD_OFF & __LVP_OFF &
__BODEN_ON & __PWRTE_ON & __WDT_ON & __HS_OSC

;__CP_OFF Pas de protection
;__DEBUG_OFF RB6 et RB7 en utilisation normale
;__WRT_ENABLE_OFF Le programme ne peut pas écrire dans la flash
;__CPD_OFF Mémoire EEprom déprotégée
;__LVP_OFF RB3 en utilisation normale
;__BODEN_ON Reset tension en service
;__PWRTE_ON Démarrage temporisé
;__WDT_ON Watchdog en service

```


Maintenant, les assignments système :

Rien de bien particulier. On n'utilise qu'une seule source d'interruption. Concernant le convertisseur A/D, on utilise une seule entrée analogique. Comme on conserve le résultat sur 10 bits, on alignera cette fois le résultat à droite, ce qui facilite les additions.

Pour varier, j'ai utilisé le signe « = » au lieu de la directive « EQU », ce qui est similaire. Nous utilisons 2 constantes dans notre programme, la valeur minimale à ajouter à notre moyenne de la valeur du potentiomètre et qui correspond à une durée d'impulsion de 998ms, et la valeur fixe de notre second module CCP, qui précise la durée séparant 2 numérisations successives.

297

```

        bsf     STATUS,RP0
        bcf     STATUS,RP1
    endm

```

Concernant les variables en banque 0, nous avons besoin de 8 emplacements pour sauver les 8 lectures du potentiomètre, une pour sauver le résultat, et une autre pour indiquer l'emplacement de sauvegarde. En fait, comme vous commencez à devenir des « pros », on peut commencer tout doucement à utiliser de temps en temps des astuces.

Les 8 valeurs vont devoir être additionnées, pour pouvoir calculer la moyenne. On peut donc, au lieu de sauvegarder la dernière valeur lue, l'additionner directement à une des 7 autres. Cela nous évite une sauvegarde, et un emplacement de sauvegarde. Donc, ceux-ci passent de 8 à 7.

Et puisque nous avons déjà commencé à additionner dans un des emplacements, autant conserver celui-ci pour le résultat, ce qui nous économise la variable de résultat.

N'oubliez pas que les emplacements des valeurs du potentiomètre doivent pouvoir contenir 10 bits, donc utilisent 2 octets. Le résultat, qui sera au maximum de 8 fois la valeur maximale d'un potentiomètre tiendra donc sur $10 + 3 = 13$ bits.

Ceci nous épargne donc 2 octets de RAM pour le résultat, et 2 autres pour l'emplacement de sauvegarde économisé. Dans notre application, cela importe peu, mais il n'en sera pas toujours ainsi. J'ai des applications qui utilisent l'intégralité des emplacements RAM. Il est d'ailleurs possible en sus d'utiliser des registres non utilisés matériellement dans l'application concernée (ADRESL, EEDATA, etc.) pour y sauver des variables.

Voici donc notre zone RAM en banque 0 :

```

; *****
;                               VARIABLES BANQUE 0                               *
; *****

; Zone de 80 bytes
; -----

CBLOCK 0x20      ; Début de la zone (0x20 à 0x6F)
zoneval : D'14'  ; 7 emplacements de 2 octets pour pot
numval  : 1      ; numéro de l'emplacement de sauvegarde
ENDC           ; Fin de la zone

```

Dans notre zone RAM commune, j'ai laissé toutes les sauvegardes, pour le cas où vous désireriez compléter ce programme pour réaliser une application pratique :

```

; *****
;                               VARIABLES ZONE COMMUNE                               *
; *****

; Zone de 16 bytes
; -----

CBLOCK 0x70      ; Début de la zone (0x70 à 0x7F)
w_temp  : 1      ; Sauvegarde registre W
status_temp : 1  ; sauvegarde registre STATUS
FSR_temp : 1      ; sauvegarde FSR (si indirect en interrupt)

```

```
PCLATH_temp : 1      ; sauvegarde PCLATH (si prog>2K)
ENDC
```

Voyons maintenant notre routine d'interruption. Comme nous n'avons qu'une seule source d'interruption, inutile de tester de laquelle il s'agit. Commençons donc par sauvegarder nos registres :

```
;*****
;                               DEMARRAGE SUR RESET                               *
;*****

org    0x000      ; Adresse de départ après reset
goto   init       ; Initialiser

; //////////////////////////////////////
;                               I N T E R R U P T I O N S                               *
; //////////////////////////////////////

;*****
;                               ROUTINE INTERRUPTION                               *
;*****
;-----
; Sauvegarde la valeur du potentiomètre sur 10 bits dans un des 8 emplacements
; prévus. numval contient le numéro de l'emplacement (! 2 octets)
; Si emplacement = 0, en plus on lance une nouvelle impulsion
; -----
;                               ;sauvegarder registres
;-----
org    0x004      ; adresse d'interruption
movwf  w_temp     ; sauver registre W
swapf  STATUS,w   ; swap status avec résultat dans w
movwf  status_temp ; sauver status swappé
movf   FSR , w    ; charger FSR
movwf  FSR_temp   ; sauvegarder FSR
movf   PCLATH , w ; charger PCLATH
movwf  PCLATH_temp ; le sauver
clrf   PCLATH     ; on est en page 0
BANK0      ; passer en banque0
```

Ensuite, on détermine si on est en train de réaliser la 8^{ème} mesure de la valeur du potentiomètre, auquel cas, on devra en plus calculer la moyenne et lancer une nouvelle impulsion.

```
      ; Interruption convertisseur A/D
      ; -----
      ; tester si 8ème interruption
      ; -----
incf   numval,f    ; incrémenter pointeur emplacement
btfsc  numval,3    ; tester si numval = 8
goto   intpulse    ; oui, démarrer impulsion
```

Si ce n'est pas le cas, on sauve la valeur lue dans l'emplacement prévu (attention, 2 octets) :

```
      ; pointer sur emplacement de sauvegarde
      ; -----
```

```

movlw zoneval-2      ; premier emplacement de sauvegarde -1
movwf FSR             ; zone emplacement dans pointeur
bcf STATUS,C         ; effacer carry
rlf numval,w          ; charger numéro interrupt * 2
addwf FSR,f           ; FSR = emplacement concerné (de 0 à 6)

                ; sauver valeur potentiometre
                ; -----
movf ADRESH,w         ; charger 8 bits forts du convertisseur A/D
movwf INDF            ; sauver le poids fort
incf FSR,f            ; pointer sur emplacement poids faible
bsf STATUS,RP0        ; passer en banque 1
movf ADRESL,w         ; charger poids faible convertisseur A/D
bcf STATUS,RP0        ; repasser banque 0
movwf INDF            ; sauver le poids faible
goto restorereg       ; fin du traitement

```

Si, par contre, nous sommes en train de lire la 8^{ème} valeur du potentiomètre, nous commençons par stopper et resetter notre timer 1.

```

                ; arrêt et reset du timer1
                ; -----
intpulse
    bcf T1CON,TMR1ON  ; arrêt timer 1
    clrf TMR1L        ; effacer timer 1 poids faible
    clrf TMR1H        ; idem poids fort

```

Ensuite, on ajoute la valeur lue à la valeur sauvée dans le premier emplacement (qui devient le futur résultat). Attention, nous travaillons toujours avec des grandeurs codées sur 2 octets. N'oubliez pas qu'il faudra donc gérer le débordement de l'octet faible vers l'octet fort (CARRY) :

```

                ; ajouter valeur potentiometre au resultat
                ; le resultat sera zoneval et zoneval+1
                ; -----
movf ADRESH,w         ; charger 8 bits forts du convertisseur A/D
addwf zoneval,f        ; ajouter au poids fort emplacement 0 (lecture 1)
bsf STATUS,RP0        ; passer banque1
movf ADRESL,w         ; charger poids faible
bcf STATUS,RP0        ; repasser banque 0
addwf zoneval+1,f      ; ajouter au poids faible emplacement 0
btfsc STATUS,C         ; tester si débordement
incf zoneval,f         ; oui, incrémenter poids fort

```

Nous avons donc dans le résultat, la somme de la lecture 1 et de la lecture 8. Il nous reste donc à additionner les 6 valeurs intermédiaires.

Nous commencerons par le dernier octet, car ceci nous permet de réaliser en premier l'addition des poids faibles, ce qui est plus simple au niveau algorithme (si vous n'êtes pas convaincus, écrivez le code correspondant au sens inverse).

On profite d'en avoir terminé avec le compteur d'emplacements pour l'utiliser comme compteur de boucles. Non seulement ça nous économise une variable, mais, de plus, à la sortie de cette boucle, numval se retrouve automatiquement à « 0 ». Ceci pour vous montrer les réflexes qui seront les vôtres lorsque vous programmerez depuis un certain temps les microcontrôleurs aux ressources limitées.

```

                ; ajouter les 6 valeurs restantes

```

```

; -----
movlw zoneval+D'13' ; pointer sur dernier poids faible
movwf FSR           ; dans pointeur
movlw 0x06          ; 6 additions à effectuer
movwf numval        ; numval = compteur de boucles
intpl
movf INDF,w         ; charger poids faible
addwf zoneval+1,f   ; ajouter à poids faible résultat
btfsc STATUS,C      ; tester si débordement
incf zoneval,f      ; oui, incrémenter poids fort
decf FSR,f          ; pointer sur poids fort
movf INDF,w         ; charger poids fort
addwf zoneval,f     ; ajouter à poids fort résultat
decf FSR,f          ; pointer sur poids faible précédent
decfsz numval,f     ; décrémenter compteur de boucles
goto intpl          ; pas 0, addition suivante

```

Maintenant nous avons besoin de calculer notre valeur moyenne. Comme nous disposons pour le moment de 8 fois cette valeur, il nous faut donc diviser par 8. Cependant, comme nous aurons besoin de 4 fois la valeur pour la multiplication par 5, nous profiterons de cette division pour sauvegarder la première division par 2 (8 fois la valeur divisé par 2 égal 4 fois la valeur).

```

; calculer 4* valeur moyenne (zoneval)
; et valeur moyenne (zoneval+2)
; -----
bcf STATUS,C        ; effacer carry
rrf zoneval,f        ; diviser poids fort par 2
rrf zoneval+1,f      ; idem poids faible, avec carry
bcf STATUS,C        ; effacer carry
rrf zoneval,w        ; charger poids fort / 4
movwf zoneval+2      ; sauver
rrf zoneval+1,w      ; charger poids faible avec carry
movwf zoneval+3      ; sauver
bcf STATUS,C        ; effacer carry
rrf zoneval+2,f      ; calculer poids fort / 8
rrf zoneval+3,w      ; charger poids faible / 8

```

Suite à ceci, la multiplication finale de la valeur moyenne par 5 se résume à additionner la valeur moyenne avec la valeur moyenne multipliée par 4.

```

; calculer 5 fois valeur moyenne
; -----
addwf zoneval+1,f    ; ajouter poids faibles
btfsc STATUS,C      ; tester si carry
incf zoneval,f      ; oui, incrémenter poids fort
movf zoneval+2,w     ; charger poids fort
addwf zoneval,w      ; ajouter à poids fort

```

Reste à ajouter notre offset (également sur 2 octets) pour obtenir notre plage de consigne désirée, le résultat sera stocké dans zoneval et zoneval+1

```

; calculer nouveau temps de pulse
; -----
addlw HIGH OFFSET    ; ajouter poids fort offset
movwf zoneval         ; dans poids fort résultat
movf zoneval+1,w     ; charger poids faible résultat

```

```

addlw LOW OFFSET      ; ajouter à poids faible offset
btfsc STATUS,C        ; tester si débordement
incf zoneval,f         ; oui, incrémenter poids fort résultat
movwf zoneval+1       ; sauver poids fort résultat

```

Reste à lancer l'impulsion pour une durée liée à la valeur précédemment calculée :

```

; lancer l'impulsion durant CCP1HL
; -----
movf zoneval,w         ; charger poids fort résultat
movwf CCP1H            ; dans poids fort consigne
movf zoneval+1,w       ; charger poids faible résultat
movwf CCP1L            ; dans poids faible consigne
movlw B'00001001'     ; comparateur, CCP1 = 0 sur égalité
bsf T1CON,TMR1ON      ; lancer timer 1
movwf CCP1CON          ; dans registre de contrôle

```

En fait, si vous faites ceci, votre pin CCP1 doit passer à « 1 » au moment de l'écriture dans CCP1CON.

Sur les PICs qui sont en ma possession (16F876SP-20), cela ne fonctionne pas, comme je l'ai expliqué dans la partie théorique. Si cela fonctionne chez vous, avec les PICs en votre possession au moment de la lecture de cet ouvrage, utilisez cette méthode, c'est la plus simple et la plus logique.

D'ailleurs, si vous passez ceci dans MPLAB 5.5, ceci ne fonctionne pas non plus. Microchip a donc intégré cette anomalie dans son émulateur.

En fait, si on la force à « 1 », la pin CCP1 passe bien à « 0 » à la fin du temps écoulé (CCP1HL = TMR1HL), mais, par contre la pin CCP1 ne passe jamais à « 1 » au moment de l'écriture du registre CCP1CON.

Qu'à cela ne tienne, **il nous suffira donc de forcer notre pin à « 1 » avant de lancer le comparateur**. On pourrait penser à :

```

; lancer l'impulsion durant CCP1HL
; -----
movf zoneval,w         ; charger poids fort résultat
movwf CCP1H            ; dans poids fort consigne
movf zoneval+1,w       ; charger poids faible résultat
movwf CCP1L            ; dans poids faible consigne
movlw B'00001001'     ; comparateur, CCP1 = 0 sur égalité
bsf T1CON,TMR1ON      ; lancer timer 1
bsf PORTC,2          ; forcer pin CCP1 à 1
movwf CCP1CON          ; dans registre de contrôle

```

Cela fonctionne effectivement dans MPLAB en mode simulateur (passage à « 1 » de RC2 (CCP1), puis passage à « 0 » le cycle suivant l'identité : CCP1HL = TMR1HL). Le problème, c'est que la pin reste désespérément à « 0 » sur notre PIC réelle placée sur son circuit. Pourquoi ?

En fait, c'est très simple et très logique. **Une fois le mode « compare avec gestion CCP » lancé, la pin RC2 n'est plus commandée par le PORTC, mais par le comparateur du module CCP.**

Donc, MPLAB a beau vous indiquer que PORTC,2 vaut « 1 », la pin RC2 reste en réalité à « 0 ». Ce n'est pas le PORTC qui commande RC2 (encore les limites de la simulation).

Il nous faut donc trouver une autre astuce. J'ai trouvé la suivante (après avoir écrit, comme indiqué, à Microchip, leur technicien m'a répondu « votre méthode pour forcer la pin CCP semble la mieux indiquée »). Si vous trouvez une meilleure astuce, n'hésitez pas à vous en servir (et à m'en faire part, bien entendu).

C'est le module CCP1 qui doit placer la pin CCP1 à « 1 ». Il suffit donc de le mettre en mode « B'00001000' et de forcer manuellement l'égalité entre TMR1HL et CCPR1HL.

Ainsi, la pin CCP1 passe immédiatement à « 1 », puisque ce n'est que l'initialisation de CCPxCON qui pose problème, et non son fonctionnement sur l'égalité.

Il suffit ensuite de rebasculer sur notre mode normal, qui remettra la pin CCP1 à 0 au bout du temps défini.

Notez que pour créer un pulse à « 0 », nous aurions du agir de façon strictement inverse (d'abord B'00001001', puis B'00001000' dans CCPxCON).

Pour créer une égalité immédiate, il suffit de mettre dans CCPR1HL la même valeur que dans TMR1HL. Comme nous avons remis notre timer à 0, nous ferons de même avec la valeur de consigne.

```
                ; forcer CCP1 à "1"
                ; -----
clrfsf CCPR1L    ; effacer comparateur consigne poids faible
clrfsf CCPR1H    ; idem poids fort (égalité forcée avec TMR1HL)
movlw B'00001000' ; comparateur, passe CCP1 à 1 sur égalité
movwf CCPR1CON    ; lancer comparateur sur égalité forcée (pin = 1 sur
                  ; cycle suivant
bsf     T1CON,TMR1ON ; lancer timer 1 (donc ici, CCP1 passe à 1)
```

Vous voyez que c'est très simple : on efface CCPR1LH, puis on force le mode compare avec passage de CCP1 à 1 sur égalité. Comme l'égalité est présente (le timer est stoppé), la pin CCP1 passe à « 1 » au cycle suivant, soit juste au moment où on lance notre timer (qui commence à compter le temps de mise à 1 de CCP1). Nous n'avons donc aucun décalage entre valeur de consigne et durée de l'impulsion.

Il nous faut ensuite lancer le module dans son mode « normal », c'est-à-dire en mode compare avec remise à « 0 » de CCP1, le cycle suivant l'égalité entre TMR1HL et CCPR1HL.

J'espère que c'est clair, et que vous n'êtes pas en train de « fumer ».

Si c'est le cas, je vous accorde une pause. Prenez une collation, présentez vos excuses à la personne (épouse, enfant) qui a eu la malencontreuse idée de vous distraire durant la lecture de ce chapitre (et à qui vous n'avez pas manqué de faire un commentaire peu courtois) et revenez relire le tout dans quelques minutes.

```

; lancer l'impulsion durant CCPR1HL
; -----
movf  zoneval,w      ; charger poids fort résultat
movwf CCPR1H         ; dans poids fort consigne
movf  zoneval+1,w    ; charger poids faible résultat
movwf CCPR1L         ; dans poids faible consigne
movlw B'00001001'    ; comparateur, CCP1 = 0 sur égalité
movwf CCP1CON        ; dans registre de contrôle

```

Notez que le timer 1 a déjà été redémarré, inutile donc de le faire une seconde fois.

Nous n'avons plus qu'à restaurer les registres sauvegardés :

```

; restaurer registres
; -----
restorereg
bcf   PIR1,ADIF      ; effacer flag interrupt
movf  PCLATH_temp,w  ; recharger ancien PCLATH
movwf PCLATH         ; le restaurer
movf  FSR_temp,w     ; charger FSR sauvé
movwf FSR            ; restaurer FSR
swapf status_temp,w ; swap ancien status, résultat dans w
movwf STATUS         ; restaurer status
swapf w_temp,f       ; Inversion L et H de l'ancien W
; sans modifier Z
swapf w_temp,w       ; Réinversion de L et H dans W
; W restauré sans modifier status
retfie               ; return from interrupt

```

Bon, le plus dur est fait. Passons maintenant à notre routine d'initialisation. RC2 en sortie, registres d'interruption et d'option ne posant pas de problème :

```

; //////////////////////////////////////
;                               P R O G R A M M E
; //////////////////////////////////////

;*****
;                               INITIALISATIONS
;*****
init

; initialisation PORTS (banque 0 et 1)
; -----
BANK1      ; sélectionner banque1
bcf  TRISC,2      ; CCP1/RC2 en sortie

; Registre d'options (banque 1)
; -----
movlw OPTIONVAL      ; charger masque
movwf OPTION_REG     ; initialiser registre option

; registres interruptions (banque 1)
; -----
movlw INTCONVAL      ; charger valeur registre interruption
movwf INTCON         ; initialiser interruptions
movlw PIE1VAL        ; Initialiser registre
movwf PIE1           ; interruptions périphériques 1

```


Ensuite, on configure notre convertisseur A/D, avec un prédiviseur de 32, puisque nous avons remis notre quartz de 20MHz :

```

; configurer le convertisseur A/D
; -----
movlw  ADCON1VAL      ; 1 entrée analogique
bsf     STATUS,RP0    ; passer banque 1
movwf   ADCON1        ; écriture dans contrôle1 A/D
bcf     STATUS,RP0    ; repasser banque 0
movlw   B'10000001'   ; convertisseur ON, prédiviseur 32
movwf   ADCON0        ; dans registre de contrôle0

```

Puis notre module CCP2, qui va lancer nos conversions A/D. Le module CCP1 n'a pas à être lancé, puisqu'il le sera dans la routine d'interruption A/D.

```

; configurer le module CCP2
; -----
movlw   B'00001011'   ; pour mode compare avec trigger
movwf   CCP2CON        ; dans registre commande CCP2
movlw   high CCPR2VAL  ; charger poids fort valeur de comparaison
movwf   CCPR2H         ; dans registre poids fort
movlw   low CCPR2VAL   ; charger poids faible valeur de comparaison
movwf   CCPR2L         ; dans registre poids faible
bsf     T1CON,TMR1ON   ; lancer le timer 1, synchrone, prédiv = 1

```

Ne reste plus qu'à initialiser notre compteur à « 0 », et à lancer les interruptions. On aurait pu se passer d'effacer PIR1, mais au diable l'avarice.

```

; initialiser variable
; -----
clrf    numval         ; on commence par la première interruption

; autoriser interruptions (banque 0)
; -----
clrf    PIR1           ; effacer flags 1
bsf     INTCON,GIE     ; valider interruptions
goto    start          ; programme principal

```

Reste notre programme principal, qui ne fait rien, une fois de plus. Libre à vous d'utiliser cette ossature pour créer un vrai programme de pilotage de modèle réduit.

« Et en plus de passer son temps derrière son ordi, Madame, il joue avec des petites voitures (ou des petits avions) ». Vraiment, vous cumulez, vos oreilles doivent souvent siffler, ou alors vous avez une épouse compréhensive (comme la mienne... quoi que, parfois...).

Mais bon, moi je ne fais pas de modélisme (j'ai essayé, mais mon avion était plus souvent à l'atelier qu'en l'air... Hé oui, pas doué, il paraît).

Fin du break, on retourne aux choses sérieuses :

```

;*****
;
;          PROGRAMME PRINCIPAL
;*****
start
    clrwdt             ; effacer watch dog

```

```
goto start          ; boucler  
END                ; directive fin de programme
```

Lancez l'assemblage, placez « servo2.hex » dans votre pic, alimentez, et vous voici en présence d'un signal de pilotage de servomoteur parfaitement standard.

Notez que si vous avez besoin de 2 servomoteurs, rien ne vous interdit d'utiliser CCP2, quitte à lancer alors manuellement la conversion A/D. Vous avez besoin de plus de servomoteurs ? Pas de problème, vous activez les entrées de commande des servomoteurs les uns après les autres en vous servant d'autres pins du PIC. Ces sorties attaquent des portes logiques qui laissent passer les pins CCP1 et CCP2 successivement sur les entrées concernées des servomoteurs.

Ne reste qu'à diminuer dans les mêmes proportions le temps de cycle total. Petit exemple théorique :

- On configure CCP1 et CCP2 chacun pour piloter une sortie.
- On affecte un temps T_c identique aux 2 modules de 10ms
- La pin RB0 sert à sélectionner servo1/servo2 ou servo3/servo4
- Après 10ms, on génère les 2 pulses destinés aux servos 1 et 2
- Après 10 ms, on change l'état de RB0
- On génère les 2 pulses destinés aux servos 2 et 3
- On modifie RB0, et on recommence

Vous voyez que chaque servo aura bien reçu son impulsion dans l'intervalle de 20ms. Avec cette méthode simple, vous pouvez piloter $(20\text{ms}/2\text{ms}) * 2 = 20$ servomoteurs simultanés, tout en n'utilisant que très peu de temps CPU, et en conservant une précision imbattable. Le timer 2 peut servir à déterminer votre temps de cycle total.

20.9 Conclusion

Ceci termine le chapitre consacré aux modules CCP. Vous voyez que ces modules ajoutent en fait des fonctions très puissantes aux timers 1 et 2.

Lorsque vous avez des signaux calibrés à générer, des mesures d'événements à réaliser, des temps précis à mesurer, votre premier réflexe doit consister à regarder si un des modes de fonctionnement des modules CCP ne serait pas adapté à votre problème.

J'ai donné un exemple d'utilisation, mais il y en a bien d'autres. Par exemple, au niveau de certaines transmissions série qui nécessitent des bits de largeur variable, bits qui sont particulièrement simples à gérer avec la procédure précédente.

Notes : ...

Notes : ...

21. Le module MSSP en mode SPI

21.1 Introduction sur le module MSSP

Le module **MSSP**, pour **Master Synchronous Serial Port**, permet l'échange de données du PIC avec le mode extérieur, en utilisant des **transmissions série synchrones**.

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, et je vais vous y aider, 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.

Notez que nous avons déjà rencontré des communications série dans le chapitre consacré à la norme ISO 7816 dans la première partie du cours (16F84). Il nous fallait alors gérer chaque bit reçu. Nous verrons qu'avec ces modules, nous n'avons plus à nous occuper de la transmission des bits en eux-mêmes, mais nous les recevons et émettrons directement sous forme d'octet, toute la procédure de sérialisation étant alors automatique.

Outre le gain en taille 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 dans 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 méthodes. Soit en mode SPI, soit en mode I²C**. J'ai décidé de séparer ces fonctions en 2 chapitres distincts, car, bien qu'elles utilisent le même module, et donc les mêmes registres, elles nécessitent des explications assez différentes.

Mais commençons par le commencement...

21.2 Les liaisons série de type synchrone

Je viens de vous parler de **liaisons série synchrones**, encore convient-il de vous donner un minimum d'explications sur ce que cela signifie :

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).

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).

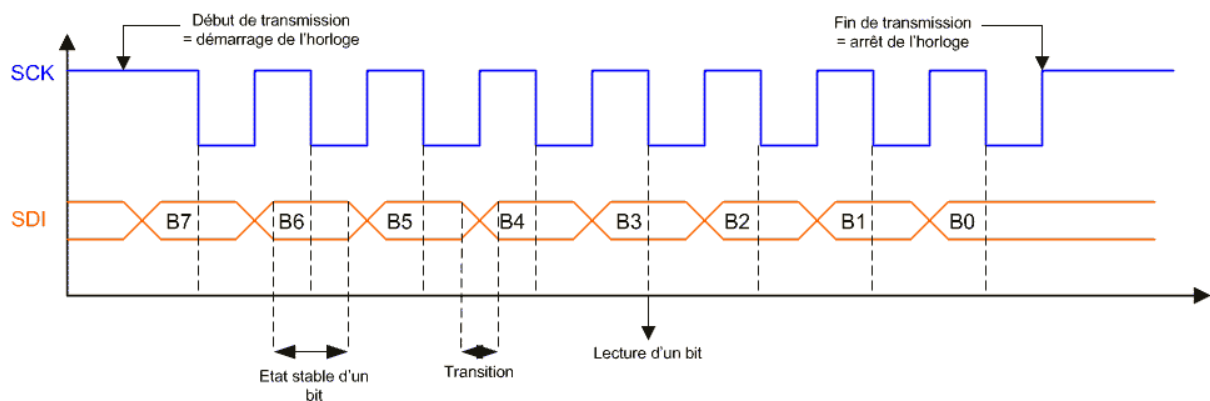
Cette méthode présente donc l'avantage de permettre la lecture des bits toujours au moment le plus favorable, et évite les problèmes dus aux imprécisions de la vitesse. Il ne nécessite pas non plus dans le cas du SPI de bit de départ (start-bit), ni de fin (stop-bit).

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 :

- Soit vous commencez par le bit 7, et vous poursuivez jusqu'au bit 0. C'est la méthode utilisée par le module MSSP.
- 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. C'est un exemple, les synchronisations et les niveaux varient d'un circuit à l'autre, ainsi que nous allons le voir plus loin. Dans cet exemple, la lecture s'effectue sur le front descendant du signal d'horloge :

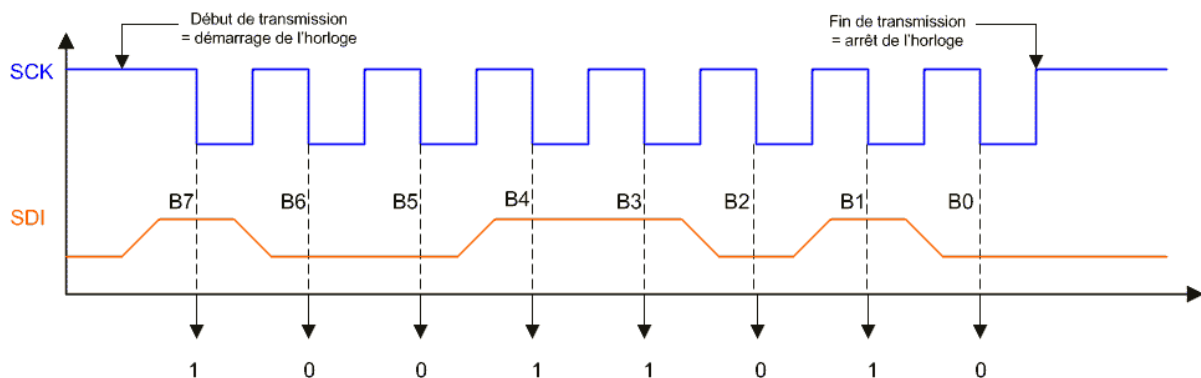


Vous constatez que :

- 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 lente, 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). Les habitués des chronogrammes de ce type voudront bien m'excuser pour cette précision.

21.3 Le mode SPI

SPI signifie **S**erial **P**eripheral **I**nterface. 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.

Le mode SPI nécessite sur notre PIC :

- Une entrée de données série (**SDI** pour **S**erial **D**ata **I**nterface)
- Une sortie de données série (**SDO** pour **S**erial **D**ata **O**utput)
- Une entrée/sortie horloge (**SCK** pour **S**erial **C**lock)
- Une entrée optionnelle de sélection du PIC (**SS** pour **S**lave **S**elect)

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

- SDI est multiplexé avec RC4
- SDO est multiplexé avec RC5
- SCK est multiplexé avec RC3
- SS est multiplexé avec RA5.

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 « multi-slave »).

Vous constaterez donc, que, si on excepte le fil de masse, **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 full-duplex.

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 dédié (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 de vos circuits**. Ceci nécessitera donc un fil supplémentaire.

Tout ceci nous donne une pléthore de modes de fonctionnement, que nous allons étudier en séquence.

21.4 Les registres utilisés

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

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

Ceci implique que la description générale de ces registres nécessite de nombreuses conditions (telle fonction si tel mode, telle autre pour tel autre mode etc.). Ceci risque de rendre les explications confuses. De plus, l'utilisateur se sert en général du module, au sein d'un programme, pour une seule fonction. Peut donc lui importe d'avoir une vue absolument générale de toutes les fonctions.

J'ai donc choisi de décrire ces registres et les bits qui les composent pour chaque mode d'utilisation particulier. Le lecteur pourra se reporter au datasheet de Microchip s'il souhaite une vue d'ensemble.

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. Tout fonctionne donc un peu comme l'instruction « rlf », excepté que le bit sortant n'est pas envoyé vers le carry, mais directement 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

A 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.

Vous pouvez parfaitement illustrer ce fonctionnement en créant un petit programme à passer au simulateur de MPLAB. Vous allez faire passer le contenu de l'émetteur dans le récepteur. On crée 2 variables, une s'appelle « emis », et l'autre « recu ». Vous imaginerez le carry comme étant la ligne de transmission.

```

movlw 0x08      ; Pour 8 octets
movwf cmpt      ; dans compteur de boucles
boucle
  rlf  emis,f    ; on décale l'émetteur, le bit à envoyer est dans le carry
  rlf  recu,f    ; on fait entrer le carry par la droite dans le récepteur
  decfsz cmpt,f  ; 8 bits traités ?
  goto boucle    ; non, suivant
  nop           ; oui, les 8 bits de l'émetteur sont dans le récepteur

```

Remarquez bien que le transfert s'effectue en 2 étapes non simultanées (les 2 lignes rlf successives, chacune exécutée sur un cycle d'horloge différent). Si vous avez compris ceci, alors vous avez compris ce qu'est une liaison série synchrone.

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 :

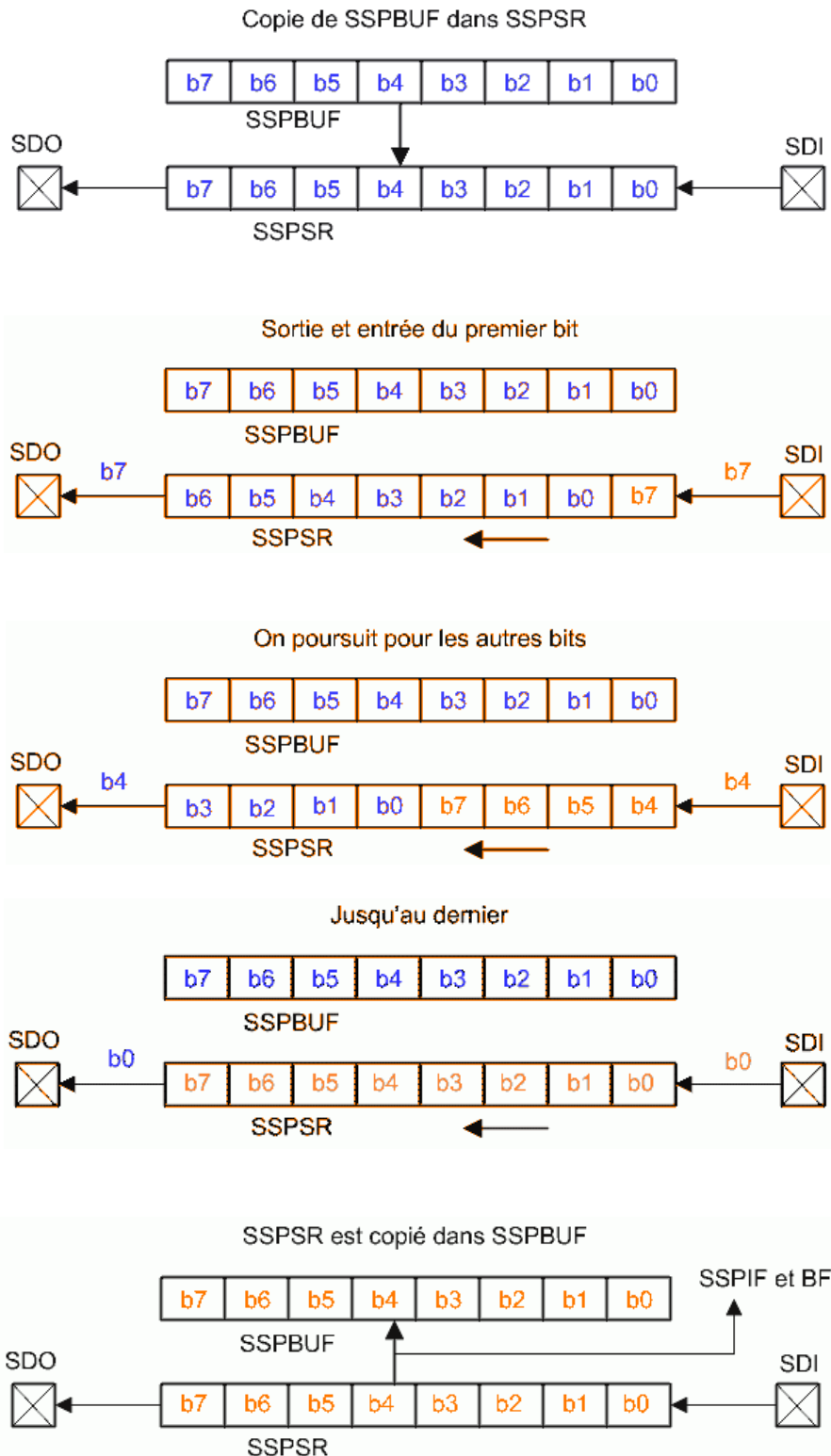
- On transfère la donnée à émettre dans SSPBUF
- Le PIC copie cette donnée dans SSPSR
- On opère 8 décalages vers la gauche, chaque bit sortant est envoyé vers SDO, chaque bit entrant provient de SDI
- Le PIC copie la donnée vers SSPBUF, donc remplace la donnée à émettre par la donnée reçue.
- A ce moment, 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 factice, 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.

A l'inverse, 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 ($b_0 = 0$), la ligne SDO restera figée à l'état bas jusqu'au prochain envoi d'un octet. Par contre, si l'octet est impair ($b_0 = 1$), cette ligne restera figée à l'état haut dans les mêmes conditions.

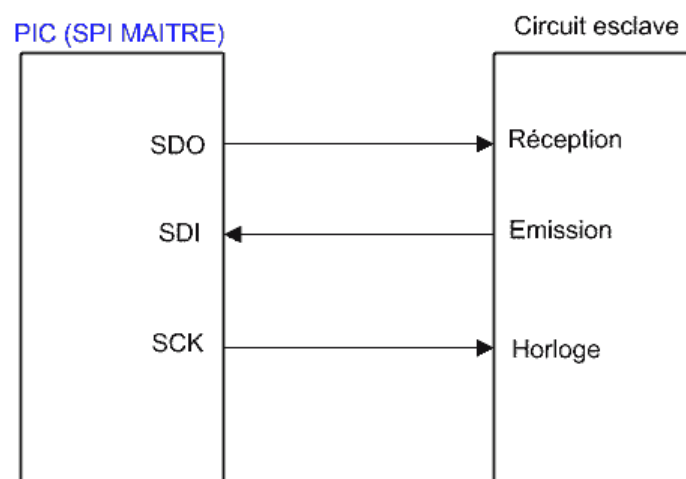
Ceci n'a aucune importance concernant la transmission, mais les amateurs d'oscilloscopes pourront se poser des questions s'ils examinent les signaux présents sur cette pin.

21.5 Le mode SPI Master

Je poursuis mes descriptions par l'utilisation du module MSSP en mode SPI, utilisé dans sa fonction de « Master » ou « maître ». Vous commencez à vous rendre compte du nombre de combinaisons, qui va rendre les exercices systématiques impossibles (ou alors vous risquez d'avoir votre cours après la cessation de fabrication des composants). Rassurez-vous, vous en aurez cependant suffisamment pour pouvoir passer de la théorie à la pratique.

21.5.1 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.

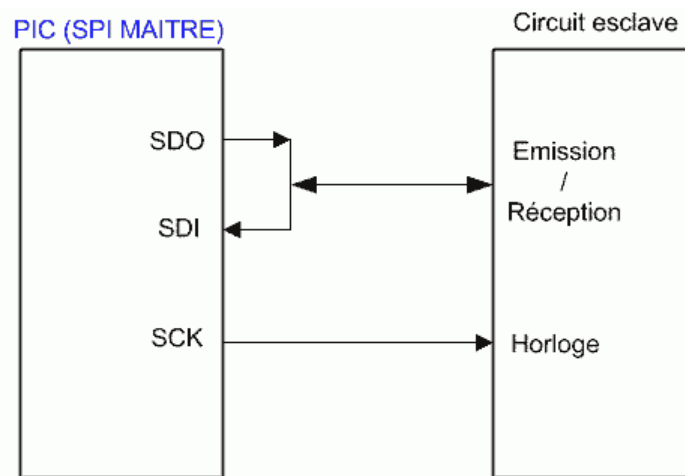
Le circuit esclave peut être aussi bien un autre PIC qu'un circuit ou un ensemble de circuits quelconque.

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 PICs, 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 :



Concernant la méthode de programmation du module, ces 2 modes n'induisent aucune différence. Il suffit simplement ici d'ajouter la gestion de la validation ou non de la pin SDO, suivant que l'on se trouve en émission ou en réception. Il faut évidemment que l'esclave fasse de même de son côté, d'une façon ou d'une autre.

En pratique, cela s'effectuera de la façon suivante :

Routine de réception

```
bsf    STATUS,RP0    ; passage en banque 1
bsf    TRISC,5        ; mettre SDO en entrée (haute-impédance)
bcf    STATUS,RP0    ; repasser banque 0 (éventuellement)
suite    ; traitement normal de la réception
```

Fin de la routine de réception

Routine d'émission

```
bsf    STATUS,RP0    ; passage en banque 1
bcf    TRISC,5        ; on met SDO en sortie
bcf    STATUS,RP0    ; repasser banque 0 (éventuellement)
suite    ; traitement normal de l'émission
```

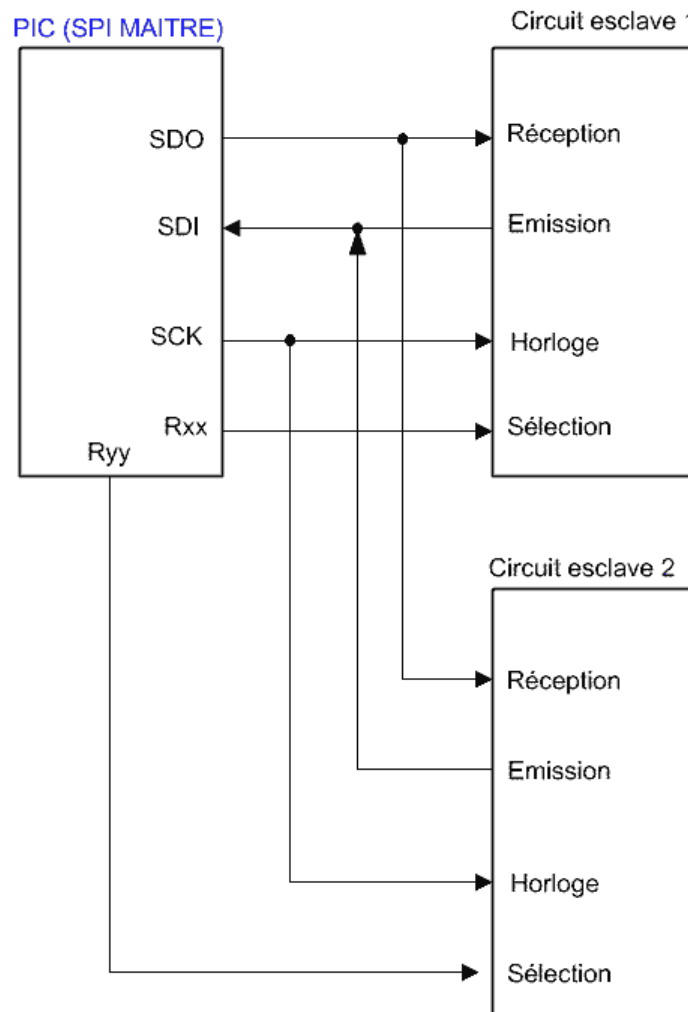
Fin de la routine d'émission

Ce mode particulier ne diffère donc des traitements « normaux » de réception et d'émission que nous allons voir que par la gestion du mode entrée/sortie de RC5 (SDO). Je n'en parlerai donc plus.

Bien entendu, il va de soi que l'émission et la réception d'un octet utile ne pourra dans ce cas être simultanée (mode half-duplex).

De nouveau, de façon interne au PIC, la réception et l'émission seront toujours simultanées. Ce mode implique donc que lorsqu'on émet dans cette configuration, on reçoit un octet qui ne provient pas en réalité de l'esclave.

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 (par exemple bsf PORTx,y) est supplémentaire par rapport aux transmissions classiques. J'utiliserai cette possibilité dans l'exercice proposé en fin de chapitre.

21.5.2 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, situé en banque 1, 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.

21.5.3 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
0	0	0	0	SPI master, période d'horloge : $T_{cy} = 4 * T_{osc}$
0	0	0	1	SPI master, période d'horloge : $T_{cy} * 4 = T_{osc} * 16$
0	0	1	0	SPI master, période d'horloge : $T_{cy} * 16 = T_{osc} * 64$
0	0	1	1	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 :

- 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.
- 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.

21.5.4 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 :

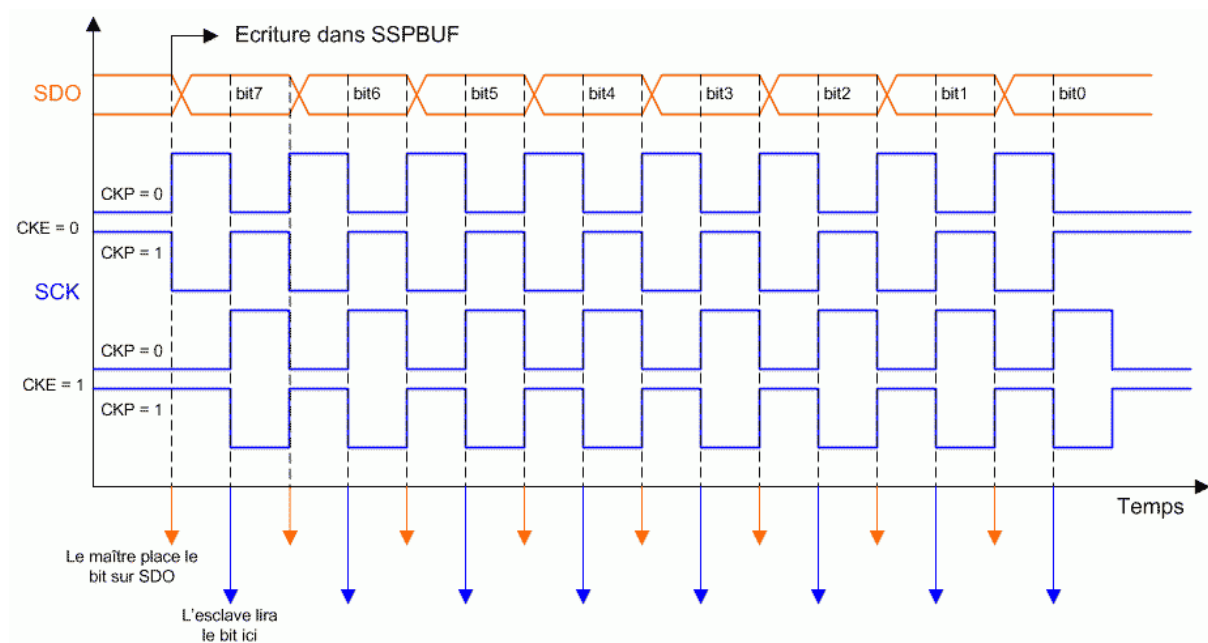
CKP CKE horloge

0	0	SCK à 0 au repos, le placement de la donnée induit un flanc montant de SCK
0	1	SCK à 0 au repos, le placement de la donnée induit un flanc descendant de SCK
1	0	SCK à 1 au repos, le placement de la donnée induit un flanc descendant de SCK
1	1	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 :



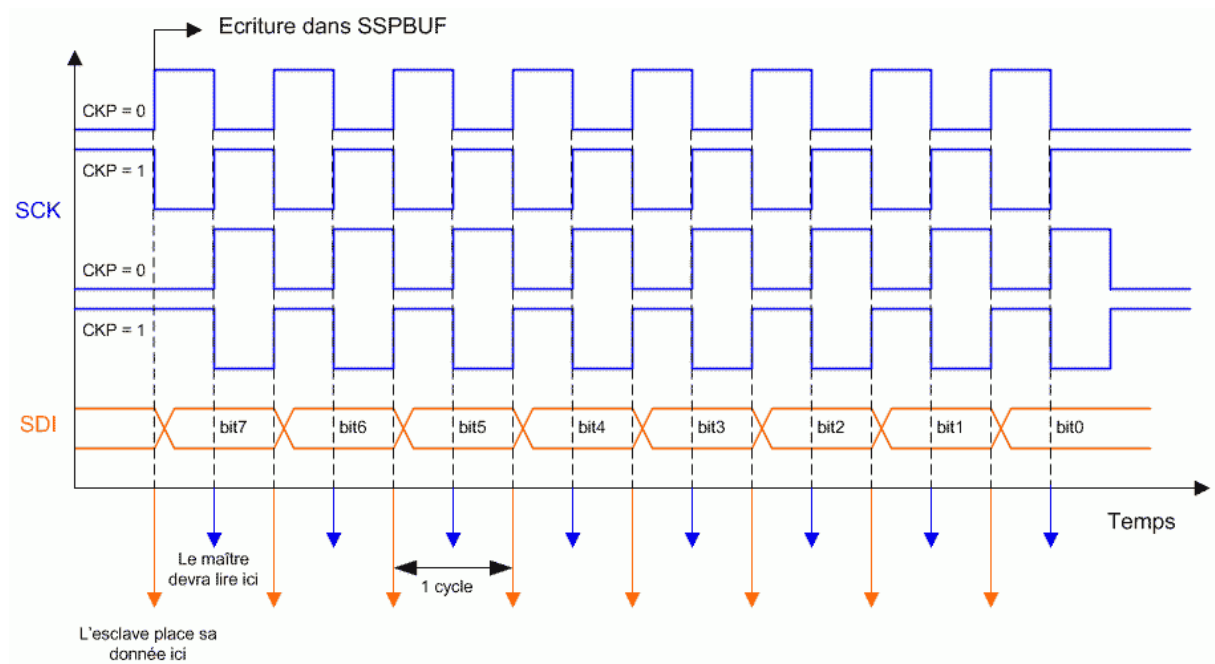
Vous constatez que l'émission des bits sur SDO est synchronisée avec l'horloge SCK, qui peut prendre 4 évolutions différentes. Vous choisirez le mode suivant le fonctionnement de l'esclave connecté. Sur le datasheet de ce dernier, le constructeur vous indiquera quelle forme le signal d'horloge doit prendre 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 redescendant de SCK, qui est le seul signal présent durant l'état stable du bit.

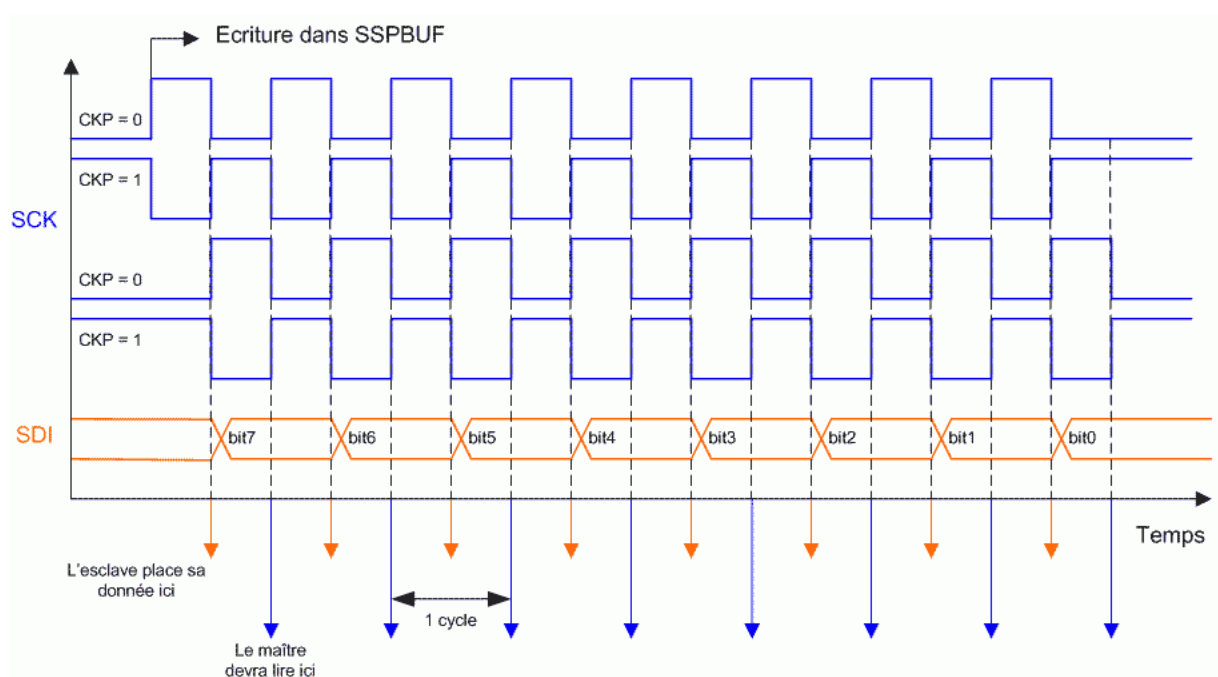
Voyons maintenant la réception d'un octet placé par l'esclave sur la ligne SDI. Nous avons 2 modes possibles, dépendants de SMP. De nouveau, ce choix découle directement de la chronologie de votre composant esclave. Le moment où son fonctionnement provoque le placement de la donnée induit le moment où 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 nouveau 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.

21.5.5 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 Fcy (fréquence de cycle), soit 5 MHz

0001 donnera une vitesse de Fcy / 4, soit 1,25 MHz

0010 donnera une vitesse de Fcy / 16, soit 312,5 KHz

0011 donnera une vitesse dépendant de votre timer 2, suivant la formule suivante :

$$T = (PR2 + 1) * Tcy * \text{prédiviseur} * 2$$

Ou encore

$$F = Fcy / ((PR2 + 1) * \text{prédiviseur} * 2)$$

Je rappelle que le postdiviseur n'est pas utilisé dans ce cas.

La vitesse maximale permise pour la liaison série synchrone est donc de $F_{osc}/4$, 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 $F_{cy} / (2 * \text{prédiviseur} * (PR2+1))$, soit $5\text{Mhz} / (2 * 16 * 256) = 610,3$ 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.

21.5.6 Initialisation du mode SPI Master

Si vous avez compris tout ce qui précède, alors vous n'avez même pas besoin de cette redondance d'information. Pour le cas où un petit résumé ne serait pas superflu, voici 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.

Rien donc de bien compliqué, une fois assimilé ce qui précède. 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.

Pour réaliser un exercice pratique, il nous faut bien entendu un maître et un esclave. Comme je ne sais pas quels circuits vous avez sous la main, je vous proposerai un exercice mettant en œuvre 2 PICs, un configuré en maître, et l'autre en esclave. Il nous faudra dès lors étudier ce second mode avant de pouvoir réaliser notre exercice pratique.

21.5.7 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.

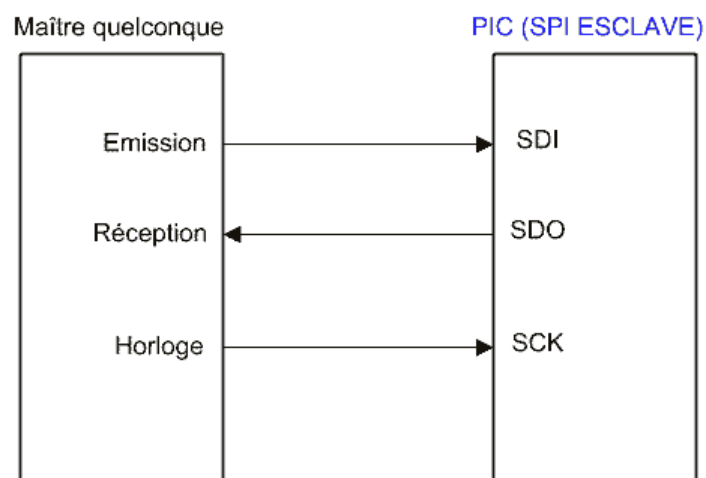
La transmission sera donc suspendue, et reprendra d'où elle se trouvait arrêtée, une fois le PIC réveillé par un autre événement externe (excepté un reset, bien sûr).

21.6 Le mode SPI SLAVE

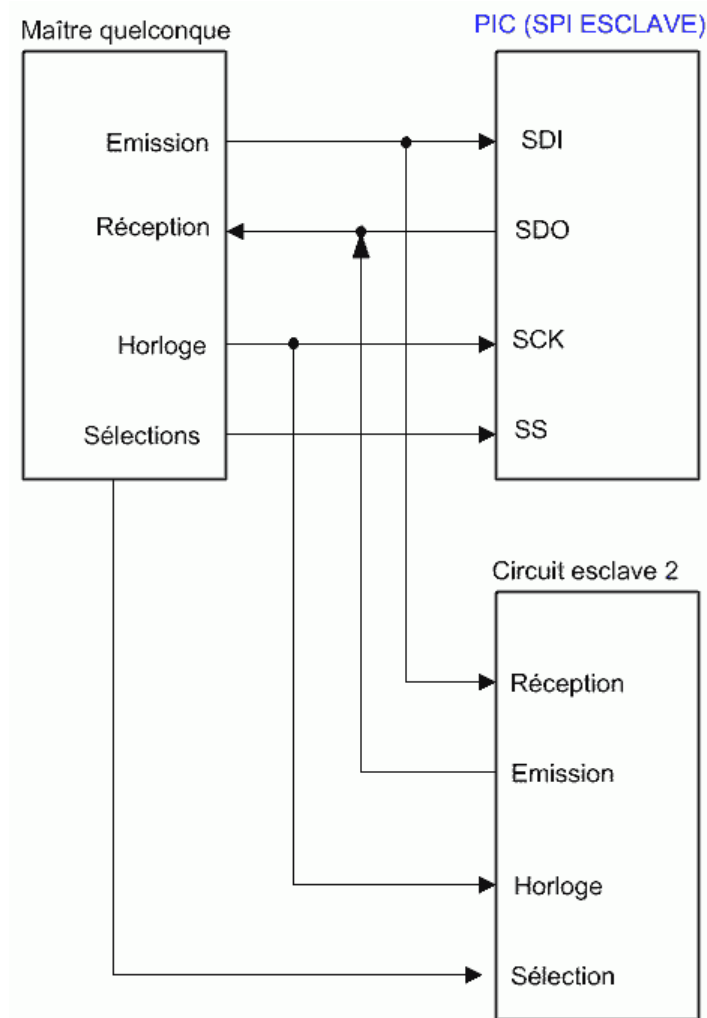
Comme vous l'aurez déjà compris depuis longtemps, ce mode (slave ou esclave) présente la particularité de **subir l'horloge de synchronisation** au lieu de l'imposer. Ceci va induire des contraintes différentes, contraintes paramétrées de nouveau par les mêmes registres que pour le mode « master ».

21.6.1 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.

21.6.2 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 données 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.

21.6.3 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.

21.6.4 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 ».

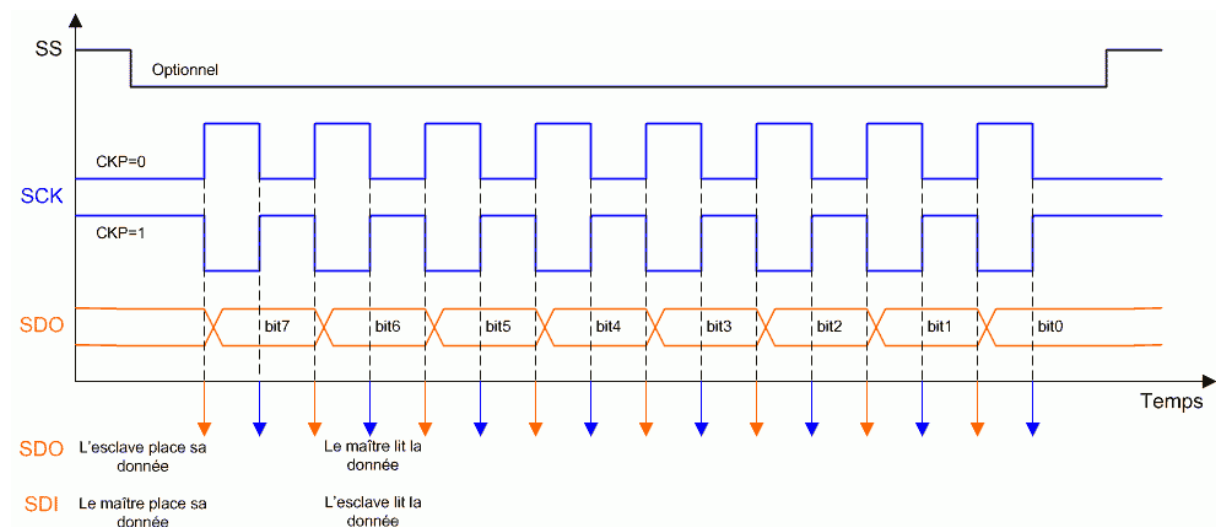
Notez, de plus, que dans ce cas vous devrez aussi définir RA5 comme entrée numérique en configurant correctement le registre ADCON1 (comme expliqué dans le chapitre sur le convertisseur A/D).

21.6.5 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.

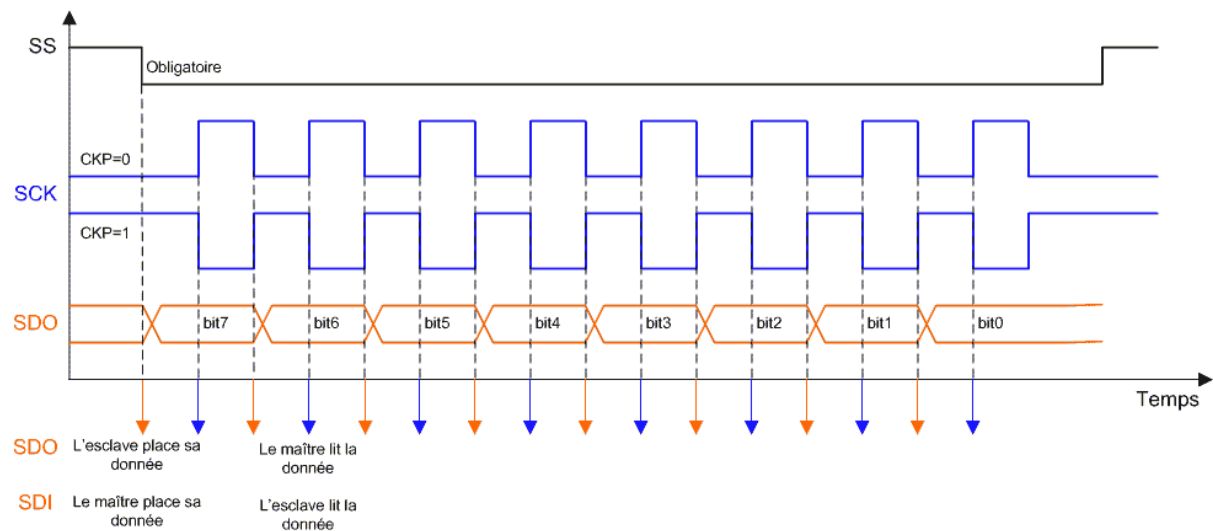
Cette fois, ce n'est plus le placement d'une valeur dans SSPBUF qui déclenche le transfert. **C'est en effet le PIC maître qui décide du début de cette transmission.** Si SCK vaut 0, le PIC esclave détecte le début de la transmission au moment de la détection du premier flanc actif de SCK.



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.

21.6.6 Vitesses de transmission

Je ne vais pas entrer ici dans trop de détails. Si vous décidez d'interfacer 2 PICs 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.

21.6.7 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.

21.6.8 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.

21.6.9 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*.

21.7 Exercice pratique : communication synchrone entre 2 pics

Nous voici arrivé au moment tant attendu de mettre toute cette théorie en application. La première chose à comprendre, c'est que pour établir une communication, il faut 2 interlocuteurs (le mono-dialogue est très peu enrichissant).

Donc, pour permettre à tous de réaliser cet exercice, je vais devoir choisir 2 composants que vous connaissez. Je choisis donc sans hésiter 2 PICs 16F876. Oui, je sais, il vous faudra en acheter un second, mais il vous servira sans aucun doute un jour ou l'autre, puisque vous lisez ce cours.

Voici donc ce que je vous propose. Nous allons programmer un PIC maître qui va compter des positions depuis 0 jusque 255. Ce PIC va envoyer ce numéro à intervalles de 1 seconde à un PIC esclave, qui lui renverra alors un octet contenu dans sa mémoire eeprom à la position demandée. Le maître recevra cet octet et s'en servira pour l'envoyer sur son PORTB, et provoquer ainsi l'allumage des LEDs correspondantes qui y seront connectées.

Ainsi, nous aurons réalisé une réception et une émission par un SPI maître, ainsi qu'une émission et une réception par un SPI esclave. Nous aurons vu de la sorte la plupart des mécanismes dans un seul exercice.

Un bouton-poussoir placé sur la pin RB0 de notre PIC esclave, permettra d'inverser l'octet à envoyer au PIC maître.

Voyons le schéma de notre montage :

- L'esclave lit l'octet « n » en provenance du maître, interprète, et place le résultat dans SSPBUF pour qu'il soit envoyé lors du prochain transfert.
- On recommence le tout avec l'octet « n+1 » pour le maître, et réponse à l'octet « n » pour l'esclave

Remarquez qu'il est très important de comprendre que l'octet reçu par le maître n'est pas la réponse à l'octet qu'il vient d'envoyer, mais la réponse à sa question précédente. En effet, émission et réception étant simultanée, il est impossible pour l'esclave de préparer sa réponse avant d'avoir reçu la question dans son intégralité. Nous verrons comment résoudre ce problème par la suite.

Bon, passons à la pratique. Nous allons établir une communication entre 2 PICs, il importe donc que ces 2 PICs parlent le même langage. Nous devons donc définir les paramètres de notre communication. J'ai choisi arbitrairement :

- Liaison à 312,5 Kbauds, donc $F_{osc}/64$ avec un quartz à 20MHz
- Parité de l'horloge : 0V au repos
- Le maître place son bit sur la transition : état de repos vers état actif.
- L'esclave renvoie son propre bit en même temps que le maître (début du cycle).
- Le maître lit donc ce bit un demi cycle plus tard, en milieu de cycle.

Qui dit 2 PICs dit 2 logiciels. Commençons par créer celui de notre PIC maître. Effectuez un copier/coller de votre fichier maquette, et renommez cette copie « SPIMast1.asm ». Créez un nouveau projet du même nom, et éditez l'en-tête.

```
;*****
; Programme de communication série synchrone entre 2 PICs 16F876.
; Logiciel pour le master.
;
; Le maître envoie le numéro de l'affichage désiré
; L'esclave renvoie l'octet correspondant de sa mémoire eeprom
; Le maître récupère l'octet et l'envoie sur son PORTB (8 LEDs)
;
; Liaison full-duplex. L'esclave renvoie donc l'octet correspondant à la
; demande précédente.
;
;*****
;
; NOM:      SPIMast1
; Date:     21/06/2002
; Version:  1.0
; Circuit:  platine d'expérimentation
; Auteur:   Bigonoff
;
;*****
;
; Fichier requis: P16F876.inc
;
;*****
;
; Notes: Les 8 LEDs sont connectées sur le PORTB
;        Les 2 PICs sont interconnectés via SDO,SDI, et SCK
;        La fréquence des quartz est de 20 MHz
;        La vitesse de communication est de  $F_{osc}/64$ , soit 312,5 KHz
;
;*****
```



```

LIST      p=16F876          ; Définition de processeur
#include <p16F876.inc>      ; fichier include

__CONFIG __CP_OFF & __DEBUG_OFF & __WRT_ENABLE_OFF & __CPD_OFF & __LVP_OFF &
__BODEN_OFF & __PWRTE_ON & __WDT_OFF & __HS_OSC

```

J'ai choisi d'envoyer un numéro toutes les secondes. J'ai choisi d'utiliser le timer 2. L'étude déjà réalisée dans le chapitre qui lui est consacré, nous donne :

- Prédiviseur : 16
- Postdiviseur : 10
- Valeur de comparaison : 249
- Nombre de passages dans la routine : 125

Le temps total sera en effet donné par la formule :

$T = (PR2+1) * \text{prédiviseur} * \text{postdiviseur} * T_{cy} * \text{nombre de passages}$

$T = (249+1) * 16 * 10 * 0,2\mu s * 125 = 1.000.000\mu s = 1s.$

La ligne de sélection de l'esclave (SS) étant connectée à la pin RC6 de notre maître, voici nos définitions et assignations :

```

;*****
;                                  DEFINITIONS ET ASSIGNATIONS                               *
;*****

#define SELECT PORTC,6           ; sélection de l'esclave
PR2VAL EQU '249'                ; Valeur de comparaison timer 2
CMPTVAL EQU '125'               ; 125 passages dans la routine d'interruption

```

Nos variables seront peu nombreuses, nous avons besoin d'un compteur de passages dans la routine d'interruption du timer2, de l'octet que l'on va incrémenter et envoyer à l'esclave, et d'un flag qui va nous servir à savoir quand les 125 passages dans la routine d'interruption auront été exécutés.

```

;*****
;                                  VARIABLES BANQUE 0                                   *
;*****

; Zone de 80 bytes
; -----

CBLOCK 0x20          ; Début de la zone (0x20 à 0x6F)
num : 1              ; numéro à envoyer à l'esclave
cmpt : 1             ; compteur de passages d'interruption
flags : 1            ; 8 flags d'usage général
                     ; b0 : une seconde s'est écoulée
ENDC                ; Fin de la zone

```

Les variables de sauvegarde ne concernent que les registres « STATUS » et « W ».

```

;*****
;
;                               VARIABLES ZONE COMMUNE
;*****
; Zone de 16 bytes
; -----

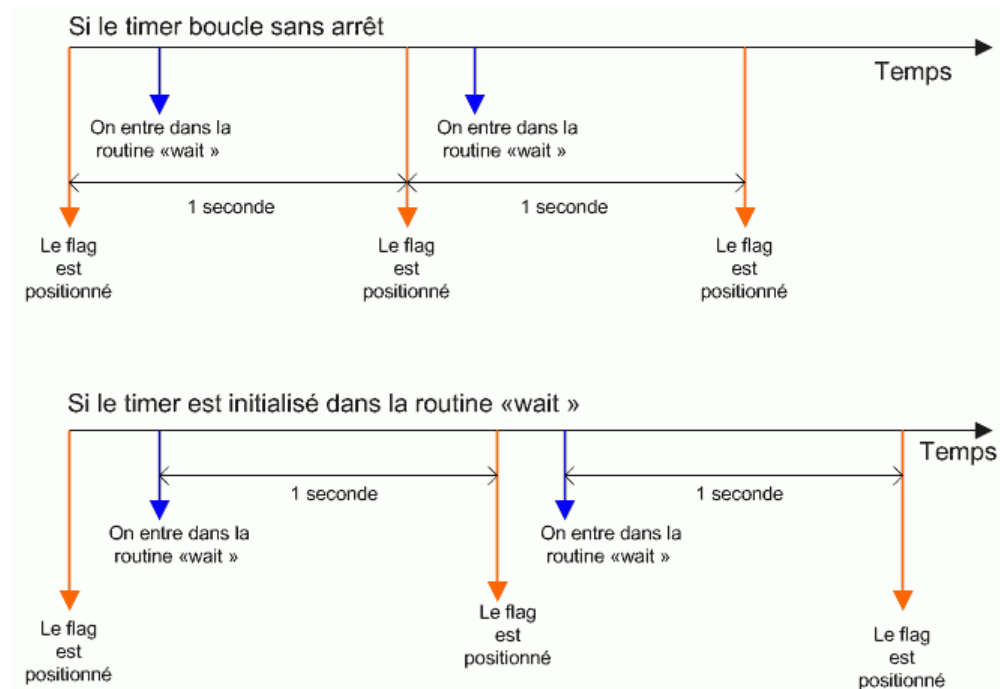
CBLOCK 0x70          ; Début de la zone (0x70 à 0x7F)
w_temp : 1           ; Sauvegarde registre W
status_temp : 1      ; sauvegarde registre STATUS
ENDC

```

La routine d'interruption est toute simple. J'ai choisi de gérer le module SPI par pooling, afin de varier les plaisirs. Nous n'aurons donc qu'une seule source d'interruption, le timer2.

Celui-ci décrémente une variable initialisée à 125. Dès que cette variable atteint 0, le flag est positionné, ce qui signale au programme principal que la durée est écoulée.

La routine d'interruption recharge elle-même le compteur, et le timer n'est pas stoppé. Cette méthode permet d'obtenir 1 seconde d'intervalle entre 2 positionnements du flag. Si on avait choisi de lancer le timer et le compteur dans la routine d'attente (wait), on aurait obtenu un temps de 1 seconde à partir de ce moment. C'est un peu compliqué à expliquer, mais plus simple à comprendre avec un petit graphique :



Le positionnement du flag signale la sortie de la routine de temporisation. La première méthode permet donc de gérer des événements à intervalles réguliers (notre cas). Le temps séparant 2 positionnements du flag est constant et ne dépend pas de l'instant où on entre dans notre routine de temporisation.

La seconde méthode permet d'attendre un temps précis dans une routine de temporisation. Elle ne comptabilise pas le temps écoulé entre le positionnement précédent du flag et l'instant où on entre dans la routine d'interruption.

```

;*****
;                               DEMARRAGE SUR RESET                               *
;*****

    org    0x000                ; Adresse de départ après reset
    goto   init                ; Initialiser

; //////////////////////////////////////
;                               I N T E R R U P T I O N S
; //////////////////////////////////////

;*****
;                               ROUTINE INTERRUPTION                               *
;*****
;-----
; La routine d'interruption timer 2 est appelée toutes les
; (0,2µs * 160 * 250) = 8ms.
; au bout de 125 passages, une seconde s'est écoulée, on positionne
; le flag
;-----

                ;sauvegarder registres
                ;-----
    org    0x004                ; adresse d'interruption
    movwf  w_temp                ; sauver registre W
    swapf  STATUS,w              ; swap status avec résultat dans w
    movwf  status_temp           ; sauver status swappé
    bcf    STATUS,RP0            ; passer banque0
    bcf    STATUS,RP1

                ; interruption timer 2
                ; -----
    bcf    PIR1,TMR2IF           ; effacer le flag d'interruption
    decfsz cmpt,f                ; décrémenter compteur de passages
    goto   restorereg            ; pas 0, fin de l'interruption
    movlw  CMPTVAL               ; valeur de recharge du compteur
    movwf  cmpt                  ; recharger compteur
    bsf    flags,0               ; positionner flag

                ;restaurer registres
                ;-----
restorereg
    swapf  status_temp,w         ; swap ancien status, résultat dans w
    movwf  STATUS                ; restaurer status
    swapf  w_temp,f              ; Inversion L et H de l'ancien W
                                ; sans modifier Z
    swapf  w_temp,w              ; Réinversion de L et H dans W
                                ; W restauré sans modifier status
    retfie                       ; return from interrupt

```

Nous arrivons maintenant dans notre routine d'initialisation. Commençons par les PORTs :

```

;*****
;                               INITIALISATIONS                               *
;*****
init
                ; initialisation PORTS
                ; -----

```

```

BANKSEL  PORTB      ; sélectionner banque 0
clrf  PORTB         ; sorties PORTB à 0
clrf  PORTC         ; sorties PORTC à 0
bsf    SELECT       ; désélectionner esclave
BANKSEL  TRISB      ; sélectionner banque 1
clrf  TRISB         ; PORTB en sortie
movlw B'10010111'   ; SDO et SCK, et select en sortie
movwf TRISC         ; direction PORTC

```

Vous remarquez la présence d'une nouvelle macro, « **BANKSEL** ». Celle-ci place RP0 et RP1 de façon similaire à nos macro « BANKx ». Cependant, on précise comme argument un nom de registre ou un emplacement mémoire. BANKSEL effectue le changement en fonction de la banque de l'adresse spécifiée. BANKSEL est une macro intégrée à MPASM. Il ne s'agit pas à proprement parler d'une directive, puisqu'elle sera remplacée par des instructions au moment de l'assemblage du programme

Dans notre exemple, « BANKSEL PORTB » détecte que PORTB se trouve en banque 0. La macro va donc créer les 2 lignes suivantes :

```

bcf    STATUS,RP0
bcf    STATUS,RP1

```

Plus loin, nous trouvons « BANKSEL TRISB ». TRISB se trouvant en banque 1, la macro va induire :

```

bsf    STATUS,RP0
bcf    STATUS,RP1

```

L'avantage apparent est que vous n'avez pas besoin de connaître la banque du registre que vous utilisez, vous ne risquez donc pas de vous tromper. L'inconvénient est que si vous ne connaissez pas cette banque, vous ne pouvez pas non plus savoir si le registre suivant que vous utiliserez fait partie ou non de la même banque, vous êtes donc contraint d'utiliser de nouveau « BANKSEL » même si ce registre ne nécessitait pas un nouveau changement de banques. A vous de voir.

Notez que BANKSEL positionne toujours les 2 bits, RP0 et RP1, même si un seul nécessitait d'être modifié. Elle sera donc toujours traduite en 2 instructions, et nécessitera donc 2 cycles d'exécution.

Le reste ne pose pas de problème, on prépare l'extinction des LEDs (dès que TRISB sera configuré), et on prépare la désélection de l'esclave en préparant le positionnement de la ligne RC6 à « 1 ».

Ensuite, on initialise et on lance notre timer 2. Le premier intervalle de 1 seconde commence donc à ce moment.

```

                ; initialiser timer 2
                ; -----
movlw PR2VAL    ; charger valeur de comparaison
BANKSEL PR2     ; passer banque 1
movwf PR2       ; initialiser comparateur
movlw B'01001110' ; timer2 on, prédiv = 16, post = 10
BANKSEL T2CON   ; passer banque 0

```

```
movwf T2CON          ; lancer timer 2
```

Viennent ensuite les initialisations des variables :

```
                ; initialiser variables
                ; -----
clrfsf num      ; on commence par l'élément 0
movlw CMPTVAL   ; pour durée initiale d'une seconde
movwf cmpt      ; dans compteur de passages
clrfsf flags    ; effacer flags
```

De nouveau, rien de particulier, on initialise le compteur de passages dans timer2, qui n'a pu encore se produire, étant donné que l'interruption timer2 n'est pas encore en service. Il n'est de ce fait pas grave de l'initialiser directement après avoir lancé le timer. Le flag est effacé, et le premier octet envoyé à l'esclave sera 0x00.

Nous allons maintenant initialiser le module SPI. Une fois de plus, toute cette théorie amène à une initialisation ultra-simple.

```
                ; initialiser SPI
                ; -----
movlw B'00100010' ; SPI maître, TOSC/64, repos = 0
movwf SSPCON      ; dans registre de contrôle
```

Nous avons décidé que le maître lirait le bit reçu au milieu du cycle, donc le bit SMP du registre SSPSTAT vaudra « 0 ».

La donnée est transmise sur le flanc « repos vers actif » de l'horloge, donc CKE du registre SSPSTAT = 0. Nous devons donc initialiser SSPSTAT avec B'00000000', ce qui est justement sa valeur au moment d'un reset. Inutile donc de l'initialiser. C'est la raison pour laquelle il n'apparaît même pas dans notre programme.

Concernant SSPCON, il faut juste mettre le SPI en service (SSPEN), et choisir la fonction maître avec FOSC/64, ce qui nous donne SSPMX = 0010. Enfin, n'est-ce pas ?

Ne reste plus qu'à mettre notre interruption du timer 2 en service :

```
                ; lancer interruption timer 2
                ; -----
bsf STATUS,RP0   ; passer banque 1
bsf PIE1,TMR2IE  ; interruption timer 2 en service
bcf STATUS,RP0   ; repasser banque 0
bsf INTCON,PEIE  ; interruptions périphériques en service
bsf INTCON,GIE   ; lancer les interruptions
```

Ceci termine notre initialisation. Le programme principal va effectuer l'envoi d'un octet toutes les secondes, réceptionner la réponse à l'envoi précédent, puis envoyer l'octet reçu sur le PORTB. Une première temporisation supplémentaire d'une seconde permet de s'assurer que l'esclave a bien terminé son initialisation et est prêt à recevoir (inutile dans ce cas-ci, car notre esclave mettra moins d'une seconde pour être prêt, mais à prendre en considération lors de l'utilisation d'un périphérique quelconque).

```

;*****
;                                     PROGRAMME PRINCIPAL                                     *
;*****
;-----
; Attend qu'une seconde soit écoulée depuis l'émission précédente
; Envoie l'octet num, et réceptionne en même temps la réponse à l'octet
; envoyé lors du précédent transfert.
; Envoie l'octet reçu sur les LEDs, et incrémente l'octet num pour le
; prochain envoi
;-----
call    wait                ; attendre 1 seconde supplémentaire pour
                           ; être sûr que l'esclave soit prêt
loop
call    wait                ; attendre 1 seconde
call    send                ; envoyer l'octet num
movf    SSPBUF,w            ; charger l'octet reçu de l'esclave
movwf   PORTB               ; l'envoyer sur les 8 LEDs
incf    num,f               ; incrémenter numéro à envoyer
goto    loop                ; boucler

```

La routine d'attente « wait » est toute simple, puisqu'elle se borne à attendre le positionnement du flag par la routine d'interruption, puis à effacer ce flag.

```

;*****
;                                     Attendre 1 seconde                                     *
;*****
;-----
; attendre qu'une seconde se soit écoulée depuis le précédent passage dans
; cette routine
;-----
wait
btfss   flags,0             ; flag positionné?
goto    wait                ; non, attendre flag
bcf     flags,0              ; reset du flag
return  ; et retour

```

Nous arrivons maintenant au cœur de notre exercice, l'émission et la réception d'un octet.

De nouveau, vous allez voir que la complexité de l'étude théorique se traduit par une énorme facilité de l'application dans un programme. C'est d'ailleurs le but des modules intégrés que de vous donner accès de façon simple à des fonctions puissantes. Essayez donc de réaliser ce transfert synchrone en pilotant vous-même SDI, SDO, et SCK. Bon courage.

De plus, vous verrez que les vitesses que vous pourrez atteindre seront nettement inférieures à celles possibles par l'utilisation du module SPI, et en monopolisant votre programme pour cette unique tâche.

```

;*****
;                                     Envoyer l'octet num                                     *
;*****
;-----
; Sélectionne l'esclave, envoie l'octet sur la liaison série et reçoit
; l'octet préparé par l'esclave.
; Attend la fin de l'échange avant de sortir, après avoir désélectionné
; l'esclave
;-----
send
bcf     SELECT                ; sélectionner l'esclave

```

```

bcf    PIR1,SSPIF    ; effacer flag
movf   num,w          ; charger octet à envoyer
movwf  SSPBUF         ; lancer le transfert

sendloop
btfss  PIR1,SSPIF    ; tester si transfert terminé
goto   sendloop      ; non, attendre
bsf    SELECT        ; désélectionner l'esclave
return ; et retour
END          ; directive fin de programme

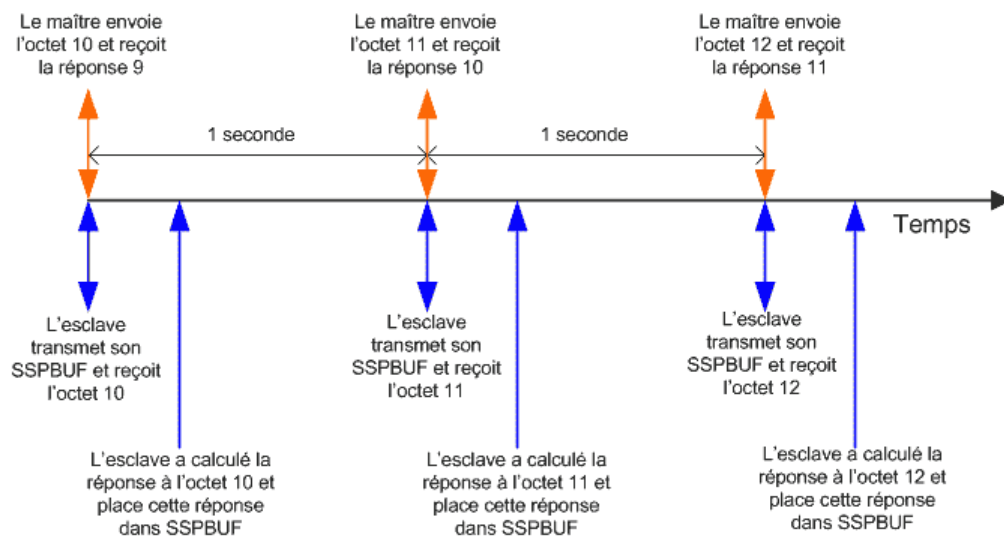
```

Attention, j'ai placé cette fois les sous-routines après le programme principal, n'oubliez pas la directive « END ».

Cette sous-routine est très simple : on place l'octet à envoyer dans SSPBUF, et on attend simplement que le bit SSPIF soit positionné, signalant que le transfert est terminé. A ce moment, l'octet reçu a remplacé celui envoyé dans SSPBUF, il ne restera plus qu'à lire ce registre dans le programme principal.

Lancez l'assemblage, et placez le fichier « .hex » obtenu dans votre PIC maître.

Pour terminer ce programme, et avant de voir le programme destiné à l'esclave, je vous donne un petit chronogramme de l'échange d'informations, afin de bien vous faire comprendre ce que vous envoyez et ce que vous recevez.



Vous constatez que le maître reçoit la réponse à sa question précédente, et non à la question en cours. Vous visualisez bien le fait qu'un transfert synchrone équivaut à l'échange entre le contenu de SSPBUF du maître et celui de l'esclave. L'échelle de temps n'est pas correcte, en réalité le temps de calcul de l'esclave est très très petit par rapport à la seconde qui sépare 2 transferts, la double et la simple flèche bleues sont donc pratiquement superposées.

Bon, passons à l'étude du programme de notre esclave. Les contraintes ont déjà été établies, effectuez un copier/coller de votre fichier maquette, et nommez cette copie « SPISlav.asm ». Créez un nouveau projet à ce nom, puis éditez l'en-tête.

```

;*****
;   Programme de communication série synchrone entre 2 PICs 16F876.      *
;   Logiciel pour l'esclave.                                           *
;                                                                           *
;   Le maître envoie le numéro de l'affichage désiré                  *
;   L'esclave renvoie l'octet correspondant de sa mémoire eeprom      *
;   Si le bouton-poussoir est activé, l'octet est inversé avant renvoi *
;   Le maître récupère l'octet et l'envoie sur son PORTB (8 LEDs)      *
;                                                                           *
;   Liaison full-duplex. L'esclave renvoie donc l'octet correspondant à la *
;   demande précédente.                                              *
;                                                                           *
;*****
;
;   NOM:      SPISlav                                                  *
;   Date:     22/06/2002                                              *
;   Version:  1.0                                                      *
;   Circuit:  platine d'expérimentation                               *
;   Auteur:   Bigonoff                                                 *
;                                                                           *
;*****
;
;   Fichiers requis: P16F876.inc                                       *
;                                                                           *
;*****
;
;   Notes: Le bouton-poussoir est connecté sur RB0                    *
;           Les 2 PICs sont interconnectés via SDO,SDI, et SCK         *
;           La fréquence des quartz est de 20 MHz                      *
;           La vitesse de communication est de Fosc/64, soit 312,5 KHz  *
;                                                                           *
;*****

```

```

LIST      p=16F876           ; Définition de processeur
#include <p16F876.inc>        ; fichier include

```

```

__CONFIG  _CP_OFF & _DEBUG_OFF & _WRT_ENABLE_OFF & _CPD_OFF & _LVP_OFF &
_BODEN_OFF & _PWRTE_ON & _WDT_OFF & _HS_OSC

```

```

;*****
;                               DEFINE                                  *
;*****

```

```

#define BP   PORTB,0          ; bouton-poussoir

```

On conservera la macro de lecture en eeprom, étant donné que nous allons y placer quelques données.

```

;*****
;                               MACRO                                  *
;*****
;
;   ; opérations en mémoire eeprom
;   ; -----

```

```

REEPROM macro                ; lire eeprom(adresse & résultat en w)
    clrwdt                   ; reset watchdog
    bcf   STATUS,RP0         ; passer en banque2

```



```

    bsf     STATUS,RP1
    movwf   EEADR           ; pointer sur adresse eeprom
    bsf     STATUS,RP0     ; passer en banque3
    bcf     EECON1,EEPGD    ; pointer sur eeprom
    bsf     EECON1,RD       ; ordre de lecture
    bcf     STATUS,RP0     ; passer en banque2
    movf    EEDATA,w        ; charger valeur lue
    bcf     STATUS,RP1     ; passer en banque0
endm

```

Aussi curieux que cela paraisse, et ce qui vous démontre la simplicité de mise en œuvre de la communication, nous n'aurons besoin d'aucune variable, exceptées celles nécessaires à la sauvegarde des registres pour la routine d'interruption.

J'ai choisi en effet de travailler par interruptions sur SPI, comme cela, vous avez un exemple avec interruptions et un exemple avec pooling.

```

;*****
;                               VARIABLES BANQUE 0                               *
;*****
; Zone de 80 bytes
; -----
; CBLOCK 0x20           ; Début de la zone (0x20 à 0x6F)
; ENDC                 ; Fin de la zone

;*****
;                               VARIABLES ZONE COMMUNE                               *
;*****

; Zone de 16 bytes
; -----

; CBLOCK 0x70           ; Début de la zone (0x70 à 0x7F)
; w_temp : 1           ; Sauvegarde registre W
; status_temp : 1      ; sauvegarde registre STATUS
; ENDC

```

Je vous ai laissé la déclaration de la zone banque 0, pour le cas où vous désireriez utiliser ce programme comme base pour une application particulière.

Nous allons ensuite programmer quelques données dans la zone eeprom. Ces données constituent les données renvoyées au maître. Celui-ci envoie le numéro (l'adresse) de l'octet souhaité, l'esclave renvoie l'octet présent à cet emplacement. Je me suis limité à 16 emplacements, le but étant de montrer les transferts, et non de créer une application pratique.

```

;*****
;                               DONNEES EEPROM                               *
;*****
; ORG0x2100      ; adresse zone data

; DE B'00000001' ; data d'allumage
; DE B'00000010'
; DE B'00000100'
; DE B'00001000'
; DE B'00010000'
; DE B'00100000'
; DE B'01000000'
; DE B'10000000'

```

```

DE B'00000001'
DE B'00000100'
DE B'00010000'
DE B'01000000'
DE B'00000010'
DE B'00001000'
DE B'00100000'
DE B'10000000'

```

Souvenez-vous que l'adresse absolue de la zone eeprom est 0x2100, et que la directive permettant d'initialiser une zone data est « DE ». Je vous mets les « 1 » en vert, vous voyez ainsi à quel allumage de LED correspond le mot déclaré.

Nous voici à notre routine d'interruption, qui ne comprend que l'interruption SPI. Notre PIC entre dans cette interruption quand le transfert est terminé. Il lit en eeprom l'octet dont le numéro est maintenant dans SSPBUF (reçu du maître), et place cet octet dans son SSPBUF, qui sera transmis lors du prochain transfert au maître.

Lorsque le bouton-poussoir est pressé, l'octet est inversé avant d'être envoyé. Les LEDs éteintes seront donc allumées et vice et versa. Ceci va s'avérer pratique pour démontrer par la pratique le décalage entre interrogation et réponse.

```

;*****
;                               DEMARRAGE SUR RESET                               *
;*****

org 0x000      ; Adresse de départ après reset
goto  init     ; Initialiser

; ////////////////////////////////////////
;                               I N T E R R U P T I O N S                               *
; ////////////////////////////////////////

;*****
;                               ROUTINE INTERRUPTION                               *
;*****
;-----
; A chaque réception d'un octet en provenance du maître, l'octet est traité
; et la réponse replacée dans SSPBUF
;-----

;sauvegarder registres
;-----

org 0x004      ; adresse d'interruption
movwf w_temp   ; sauver registre W
swapf STATUS,w ; swap status avec résultat dans w
movwf status_temp ; sauver status swappé

; Interruption SSP
; -----
BANKSEL PIR1   ; passer banque 0
bcf PIR1,SSPIF ; effacer flag interrupt
movf SSPBUF,w  ; charger numéro reçu du maître
andlw 0x0F     ; on n'a que 16 positions valides
REEPROM       ; lire l'octet correspondant en eeprom
btfss BP      ; tester si bouton pressé
xorlw 0xFF    ; oui, inverser l'octet reçu
movwf SSPBUF  ; placer réponse dans SSPBUF qui sera
; envoyée en même temps que la réception

```

```

; de l'octet suivant

; restaurer registres
; -----
restorereg
    swapf status_temp,w    ; swap ancien status, résultat dans w
    movwf STATUS           ; restaurer status
    swapf w_temp,f         ; Inversion L et H de l'ancien W
    swapf w_temp,w         ; Réinversion de L et H dans W
    retfie                 ; return from interrupt

```

Notre routine d'initialisation commence par s'occuper des PORTs.

```

; //////////////////////////////////////
;                               P R O G R A M M E
; //////////////////////////////////////

;*****
;                               INITIALISATIONS
;*****
init

    ; initialisation PORTS
    ; -----
    BANKSEL TRISC           ; passer banque 1
    bcf TRISC,5             ; SDO en sortie
    movlw 0x06              ; pour PORTA en numérique
    movwf ADCON1            ; dans registre de contrôle

```

Notez que, bien que la pin *SS* soit prise en charge par le module SPI, le fait d'oublier de configurer ADCON1 pour l'imposer comme pin numérique, la ferait détecter en priorité comme une entrée analogique.

A propos, en me relisant, je tiens à préciser que « *SS* » est en italique puisqu'il s'agit d'une pin active à l'état bas, non pour évoquer une idéologie que par ailleurs je réprouve. Je tenais à cette précision pour éviter un dangereux amalgame quant à mes intentions. Je sais que certains sont très forts pour découvrir des messages cachés dans les livres, tableaux et autres œuvres, mais je tiens à rassurer tout le monde : il s'agit d'un cours, non d'une « œuvre », et il n'y a aucun message caché. Tous mes messages sont « en clair dans le texte ».

Le mode SPI esclave ne requiert qu'une seule sortie, à savoir la pin SDO (RC5).

L'initialisation du module SPI est aussi simple que pour le maître, on choisira le mode esclave avec gestion de la pin *SS*. On initialise SSPBUF à 0, de façon que le premier transfert envoie cette valeur au maître.

```

    ; initialiser SPI
    ; -----
    bcf STATUS,RP0         ; repasser banque 0
    movlw B'00100100'      ; esclave avec SS activée
    movwf SSPCON            ; dans registre
    clrf SSPBUF             ; premier octet envoyé = 0

```

Il nous faut maintenant mettre l'interruption SPI en service :

```

; autoriser interruptions
; -----

```

```

bsf    STATUS,RP0      ; passer banque 1
bsf    PIE1,SSPIE      ; autoriser interrupts SPI
bcf    STATUS,RP0      ; repasser banque 0
bsf    INTCON,PEIE     ; autoriser interruptions périphériques
bsf    INTCON,GIE      ; valider interruptions

```

Et le reste de notre programme ? Vous allez rire :

```

;*****
;
;               PROGRAMME PRINCIPAL                      *
;*****
start
    goto start      ; boucler
END                ; directive fin de programme

```

Difficile, de nouveau, de faire plus simple.

L'esclave pouvant être mis en mode sommeil durant la réception, vous pouvez ajouter :

```

start
    sleep          ; mise en sommeil
    goto start     ; boucler
END               ; directive fin de programme

```

Attention, dans ce cas. Le réveil du PIC provoque la remise en service de l'oscillateur, qui est un signal fortement perturbateur. Si votre PIC est incorrectement découplée, si vos fils sont longs ou sujets aux perturbations, vous risquez d'obtenir un fonctionnement aléatoire.

Je vous déconseille donc d'utiliser cette instruction « sleep » sur une platine d'expérimentation, comportant, de surcroît, 2 PICs disposant de leur propre oscillateur. N'oubliez pas que les vitesses de transmission sont assez importantes, et donc facilement sensibles aux perturbations.

Lancez l'assemblage, placez les 2 PICs sur le circuit et lancez l'alimentation. Au bout de quelques secondes, les LEDs commencent à clignoter sur le maître, preuve que les transferts fonctionnent.

Si vous maintenez pressé le bouton-poussoir, vous constatez que l'allumage suivant est toujours non inversé. Ce n'est que 2 allumages plus tard que les LEDs seront inversées. Ceci vous démontre que la réponse concerne toujours l'octet précédent.

21.8 Exercice 2 : alternative de fonctionnement

Si ce fonctionnement ne nous convient pas, c'est-à-dire si nous voulons que l'allumage des LEDs corresponde à l'adresse envoyée, nous n'avons d'autre alternative que de travailler en half-duplex, c'est-à-dire, émettre l'octet, puis recevoir la réponse par une autre transmission. Comme la transmission comporte automatiquement une émission et une réception simultanées, nous ignorerons simplement le premier octet reçu.

Une transmission se fera donc de la façon suivante :

- On envoie le numéro de l'octet

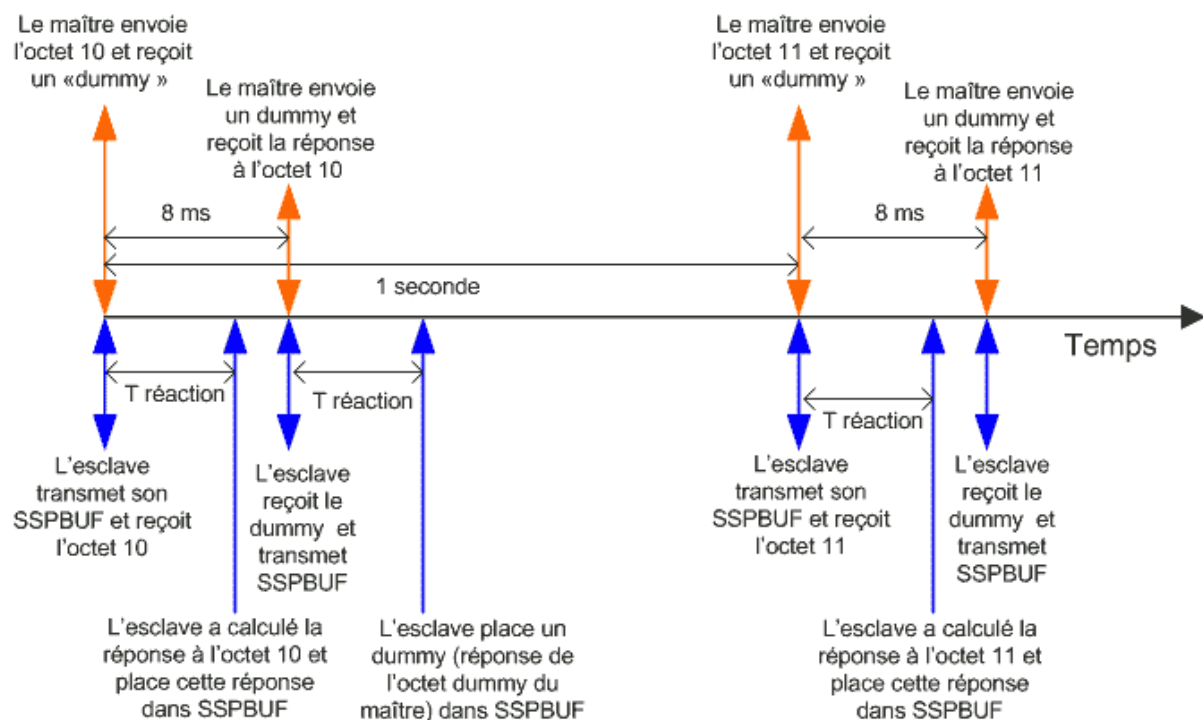
- On réceptionne l'octet correspondant à la requête précédente, on l'ignore (octet dummy)
- On attend que l'esclave ait eu le temps de placer l'octet dans SSPBUF
- On envoie n'importe quoi (octet dummy)
- On réceptionne l'octet correspondant à la requête précédente (octet courant)
- On place cet octet sur le PORTB.

Vous voyez que de la sorte, on procède en 2 étapes : l'émission d'un octet utile avec réception d'un octet ne servant à rien, suivie par émission d'un octet ne servant à rien avec réception d'un octet utile

Entre les 2 transmissions, il est impératif d'attendre que l'esclave ait eu le temps de traiter l'information. Nous allons nous servir de nouveau de notre routine d'interruption timer 2, pour positionner un autre flag. Le temps séparant 2 passages dans cette routine est de 8ms, ce qui est largement supérieur au temps de réaction de l'esclave.

Pour calculer ce dernier, il suffit de compter les cycles d'instructions survenus entre le moment où le maître est interrompu par sa routine d'interruption timer2, et le moment où l'esclave a placé sa réponse dans SSPBUF. 8ms correspondent à 40.000 cycles d'instruction, vous voyez que l'esclave a eu tout le temps nécessaire pour placer sa réponse.

Voici ce que ça donne sur un petit dessin (attention, de nouveau pas à l'échelle) :



On voit très nettement que chaque échange d'information utile nécessite à présent 2 transmissions consécutives. Par contre, l'avantage est que l'octet reçu est bien l'octet de réponse à la requête actuelle, et non plus à la précédente.

Le programme de l'esclave reste inchangé, puisqu'il se contente de répondre aux interrogations. Il lui importe peu que l'interrogation soit un « dummy » (pour rappel, octet inutile) ou un octet réellement important. En modifiant le programme de l'esclave, on aurait

pu se passer de calculer la réponse au dummy reçu, et placer directement n'importe quoi dans SSPBUF. Cela n'a cependant aucune importance pour cette application.

Par contre, le maître devra gérer cette double transmission. Effectuez un copier/coller de votre programme « SPIMast1.asm » et renommez la copie en « SPIMast2.asm ». Créez un nouveau projet à ce nom.

Le début ne pose pas problème :

```
;*****
; Programme de communication série synchrone entre 2 PICs 16F876.      *
; Logiciel pour le master.                                           *
;                                                                     *
; Le maître envoie le numéro de l'affichage désiré                  *
; L'esclave renvoie l'octet correspondant de sa mémoire eeprom      *
; Le maître récupère l'octet et l'envoie sur son PORTB (8 LEDs)      *
;                                                                     *
; Liaison half-duplex. L'esclave renvoie donc l'octet correspondant à la *
; demande actuelle. L'échange d'informations nécessite cependant 2  *
; cycles d'émission/réception                                       *
;                                                                     *
;*****
;
; NOM:      SPIMast2                                                *
; Date:     23/06/2002                                             *
; Version:  1.0                                                    *
; Circuit:  platine d'expérimentation                             *
; Auteur:   Bigonoff                                               *
;                                                                     *
;*****
;
; Fichier requis: P16F876.inc                                       *
;                                                                     *
;*****
;
; Notes: Les 8 LEDS sont connectées sur le PORTB                  *
;        Les 2 PICs sont interconnectés via SDO,SDI, et SCK       *
;        La fréquence des quartz est de 20 MHz                    *
;        La vitesse de communication est de Fosc/64, soit 312,5 KHz *
;                                                                     *
;*****

LIST      p=16F876          ; Définition de processeur
#include <p16F876.inc>       ; fichier include

__CONFIG  _CP_OFF & _DEBUG_OFF & _WRT_ENABLE_OFF & _CPD_OFF & _LVP_OFF &
_BODEN_OFF & _PWRTE_ON & _WDT_OFF & _HS_OSC

;*****
;
;          DEFINITIONS ET ASSIGNATIONS                             *
;*****

#define    SELECT_PORTC,6    ; sélection de l'esclave
PR2VAL    EQU    D'249'     ; Valeur de comparaison timer 2
CMPTVAL    EQU    D'125'    ; 125 passages dans la routine d'interruption
; durée = Tcy*(PR2+1)*prédiv*postdiv*cmptval
; = 0,2µs * 250 * 16 * 10 * 125 = 1s.
```

Ensuite, les variables. Il suffit pour nous d'utiliser un second bit comme flag dans la variable « flags », nous choisirons arbitrairement le bit 1.

```
;*****
;                               VARIABLES BANQUE 0                               *
;*****

; Zone de 80 bytes
; -----

CBLOCK 0x20          ; Début de la zone (0x20 à 0x6F)
num : 1              ; numéro à envoyer à l'esclave
cmpt : 1             ; compteur de passages d'interruption
flags : 1            ; 8 flags d'usage général
                      ; b0 : une seconde s'est écoulée
                      ; b1 : 8 ms se sont écoulées
ENDC                 ; Fin de la zone
```

Rien à dire non plus sur les variables en zone commune :

```
;*****
;                               VARIABLES ZONE COMMUNE                               *
;*****

; Zone de 16 bytes
; -----

CBLOCK 0x70          ; Début de la zone (0x70 à 0x7F)
w_temp : 1           ; Sauvegarde registre W
status_temp : 1      ; sauvegarde registre STATUS
ENDC
```

La routine d'initialisation est strictement identique, excepté le positionnement du flag 1 à chaque passage dans la routine d'interruption.

```
;*****
;                               DEMARRAGE SUR RESET                               *
;*****

org 0x000            ; Adresse de départ après reset
goto init            ; Initialiser

; ////////////////////////////////////////
;                               I N T E R R U P T I O N S                               *
; ////////////////////////////////////////

;*****
;                               ROUTINE INTERRUPTION                               *
;*****
;-----
; La routine d'interruption timer 2 est appelée toutes les
; (0,2µs * 160 * 250) = 8ms.
; au bout de 125 passages, une seconde s'est écoulée, on positionne
; le flag
;-----

;-----
;sauvegarder registres
;-----
org 0x004            ; adresse d'interruption
```

```

movwf w_temp          ; sauver registre W
swapf STATUS,w        ; swap status avec résultat dans w
movwf status_temp     ; sauver status swappé
bcf STATUS,RP0        ; passer banque0
bcf STATUS,RP1

                ; interruption timer 2
                ; -----
bcf PIR1,TMR2IF       ; effacer le flag d'interruption
bsf flags,1           ; 8 ms écoulées
decfsz cmpt,f         ; décrémenter compteur de passages
goto restorereg       ; pas 0, fin de l'interruption
movlw CMPTVAL         ; valeur de recharge du compteur
movwf cmpt            ; recharger compteur
bsf flags,0           ; positionner flag

                ; restaurer registres
                ; -----
restorereg
swapf status_temp,w   ; swap ancien status, résultat dans w
movwf STATUS          ; restaurer status
swapf w_temp,f        ; Inversion L et H de l'ancien W
                        ; sans modifier Z
swapf w_temp,w        ; Réinversion de L et H dans W
                        ; W restauré sans modifier status
retfie                ; return from interrupt

```

Rien ne change pour l'initialisation

```

;*****
;                               INITIALISATIONS                               *
;*****
init
                ; initialisation PORTS
                ; -----
BANKSEL PORTB    ; sélectionner banque 0
clrf PORTB       ; sorties PORTB à 0
clrf PORTC       ; sorties PORTC à 0
bsf SELECT       ; désélectionner esclave
BANKSEL TRISB    ; sélectionner banque 1
clrf TRISB       ; PORTB en sortie
movlw B'10010111' ; SDO et SCK, et select en sortie
movwf TRISC       ; direction PORTC

                ; initialiser timer 2
                ; -----
movlw PR2VAL     ; charger valeur de comparaison
BANKSEL PR2      ; passer banque 1
movwf PR2        ; initialiser comparateur
movlw B'01001110' ; timer2 on, prédiv = 16, post = 10
BANKSEL T2CON    ; passer banque 0
movwf T2CON      ; lancer timer 2

                ; initialiser variables
                ; -----
clrf num         ; on commence par l'élément 0
movlw CMPTVAL    ; pour durée initiale d'une seconde
movwf cmpt       ; dans compteur de passages
clrf flags       ; effacer flags

```



```

; initialiser SPI
; -----
movlw B'00100010' ; SPI maître, TOSC/64, repos = 0
movwf SSPCON      ; dans registre de contrôle

; lancer interruption timer 2
; -----
bsf STATUS,RP0    ; passer banque 1
bsf PIE1,TMR2IE   ; interruption timer 2 en service
bcf STATUS,RP0    ; repasser banque 0
bsf INTCON,PEIE   ; interruptions périphériques en service
bsf INTCON,GIE    ; lancer les interruptions

```

C'est dans le programme principal que se situe la différence de fonctionnement. Vous noterez la présence de la temporisation des 8ms, et la double émission/réception. Afin d'éviter d'utiliser une sous-routine spécifique, l'octet « dummy » envoyé sera de nouveau l'octet « num ». Ceci n'a aucune importance, l'octet « dummy » pouvant, par définition, être n'importe quoi, donc pourquoi pas l'octet « num » ?

```

;*****
;                               PROGRAMME PRINCIPAL                               *
;*****
;-----
; Attend qu'une seconde soit écoulée depuis l'émission précédente
; Envoie l'octet num et réception un dummy
; 8 ms après le début de la première transmission, envoie un dummy (ici,
; pour des raisons de facilité, envoie l'octet num comme dummy, cela
; évite d'utiliser une seconde sous-routine) et réceptionne la réponse
; à l'octet num.
; Envoie l'octet reçu sur les LEDs, et incrémente l'octet num pour le
; prochain envoi
;-----
    call wait          ; attendre 1 seconde supplémentaire pour
                        ; être sûr que l'esclave soit prêt
loop
    call wait          ; attendre 1 seconde
    call send          ; envoyer l'octet num, recevoir dummy

    bcf flags,1        ; effacer flag 1
loop2
    btfss flags,1      ; tester si 8 ms écoulées
    goto loop2         ; non, attendre

    call send          ; envoyer l'octet dummy, recevoir réponse
                        ; à l'octet num
    movf SSPBUF,w      ; charger l'octet reçu de l'esclave

    movwf PORTB        ; l'envoyer sur les 8 LEDs
    incf num,f         ; incrémenter numéro à envoyer
    goto loop

```

Lancez l'assemblage, placez votre PIC maître sur le circuit, et envoyez l'alimentation. Vous ne constatez aucune différence. C'est normal, étant donné que vous ne voyez pas les correspondances entre octet envoyé et octet reçu.

Ceci explique pourquoi j'ai ajouté un bouton-poussoir. Pressez-le et vous constatez que les LEDs seront inversés dès le clignotement suivant. Ceci démontre que l'octet reçu est bien la réponse à l'octet en cours de traitement, et non plus la réponse au précédent.

21.9 Conclusions

Voilà, vous êtes maintenant devenus des spécialistes du module SPI. Une corde de plus est donc ajoutée à votre arc de programmeur es-PICs. N'oubliez pas que les transmissions série synchrones sont utilisées, en général, pour les cas suivants :

- Liaisons à hauts débits
- Liaisons à courtes distances (le plus souvent entre 2 composants de la même carte).
- Communications avec des circuits spécifiques nécessitant des liaisons synchrones

Elles sont souvent sensibles aux perturbations, et il convient d'être soigneux dans la réalisation d'applications pratiques.

Les exercices présentés n'avaient nul besoin de gérer les conditions d'erreur. Il n'en sera pas toujours de même pour vos applications pratiques. Pensez donc à vérifier les indicateurs concernés décrits dans la partie théorique, si vous avez des raisons de penser qu'il peut arriver qu'un des indicateurs d'erreur se retrouve positionné.

De nouveau, c'est à vous de savoir s'il y a risque ou non de collision ou de débordement, d'après la structure de votre programme, et du périphérique connecté.

Notes : ...

Notes : ...

22. Le bus I²C

22.1 Introduction

J'ai décidé, plutôt que de passer directement à l'étude du module MSSP en mode I²C, de commencer par vous expliquer en quoi consiste ce bus.

En effet, bien que le module prenne en charge tous les aspects de gestion de ce bus, il m'a semblé important de comprendre ce qui se passait effectivement au niveau des lignes qui le constituent.

De plus, l'absence de ces informations vous poserait problème au moment d'établir une communication avec un composant I²C. En effet, lorsque vous rencontrerez un nouveau composant, tout ce dont vous disposerez à son sujet sera, en général, son datasheet. Sans une connaissance assez précise du bus I²C, vous vous retrouveriez confronté à quelque chose qui risquerait bien de vous paraître incompréhensible.

Et puis, mieux vaut avoir une information qu'on n'utilisera éventuellement pas qu'une absence d'information au moment où on en a besoin. Vous ne pensez pas ?

22.1 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), mais je ne parlerai pas de ce bus ici. Si cela vous intéresse, vous trouverez des informations sur Internet. Sachez simplement que nos 16F87x peuvent créer des signaux compatibles avec cette norme.

L' I²C permettait, à ses débuts, de travailler à des fréquences maximales de 100 Kbits/seconde, vitesses assez rapidement portées à 400 Kbits/seconde. Il existe maintenant des familles de circuits pouvant atteindre des vitesses de 3.4 Mbits/seconde. Il n'est pas dans mon propos d'étudier ces modèles « hi-speed » particuliers. De toute façon, le fonctionnement théorique reste identique.

Le bus I²C est constitué de 2 uniques lignes bidirectionnelles :

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

Il est important de vous souvenir 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 :

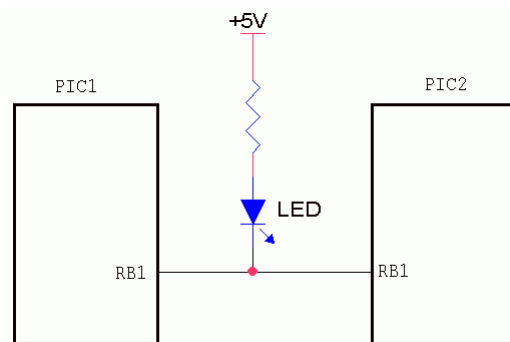
- La destruction de l'électronique des circuits, pour le cas où l'un d'entre eux impose un niveau haut et l'autre un niveau bas (court-circuit).
- La corruption des données destinées ou en provenance de l'esclave.

Fort heureusement, vous vous doutez bien que Philips (l'inventeur de l'I²C) a trouvé des solutions pour parer à ces éventualités. Je commence par les contraintes électroniques. Je parlerai des corruptions de données plus loin dans ce chapitre.

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 pics, par exemple. Imaginons le montage suivant :

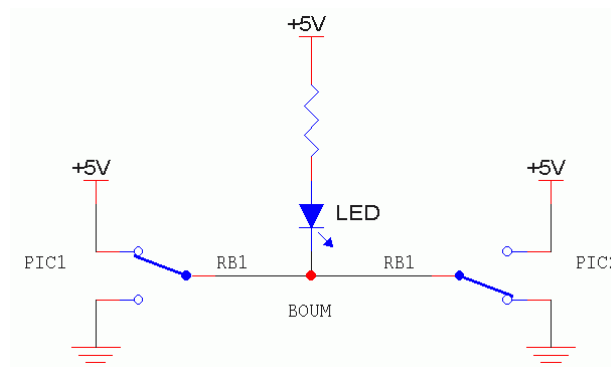


Imaginons que vous désiriez allumer la LED depuis n'importe lequel des 2 PICs. Si vous créez les sous-routines suivantes dans chaque PIC :

```
Allumage
    bcf    PORTB,1
    return
```

```
extinction
    bsf    PORTB,1
    return
```

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 PICs. Si on regarde de façon schématique ce qui se passe au niveau des 2 PICs, 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 :

```
Initialisation
    bcf    PORTB,1      ; Si RB1 en sortie, alors RB1 = 0
```

```
Allumage
    BANK1                ; passer banque 1
    bcf    TRISB,1        ; RB1 en sortie, donc allumage de la LED
    BANK0                ; repasser banque 0
```

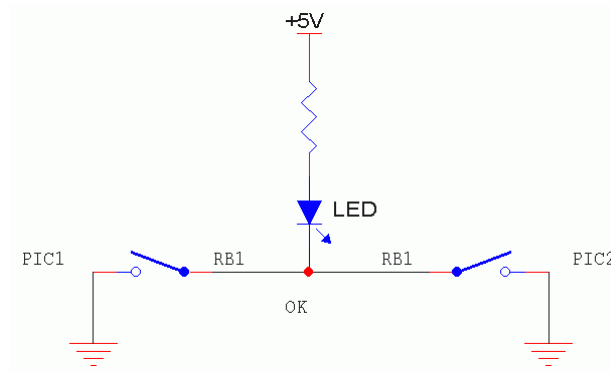
```

return

extinction
    BANK1                ; passer banque 1
    bsf    TRISB,1        ; RB1 en entrée, donc non connectée -> Led éteinte
    BANK0                ; repasser banque 0
    return

```

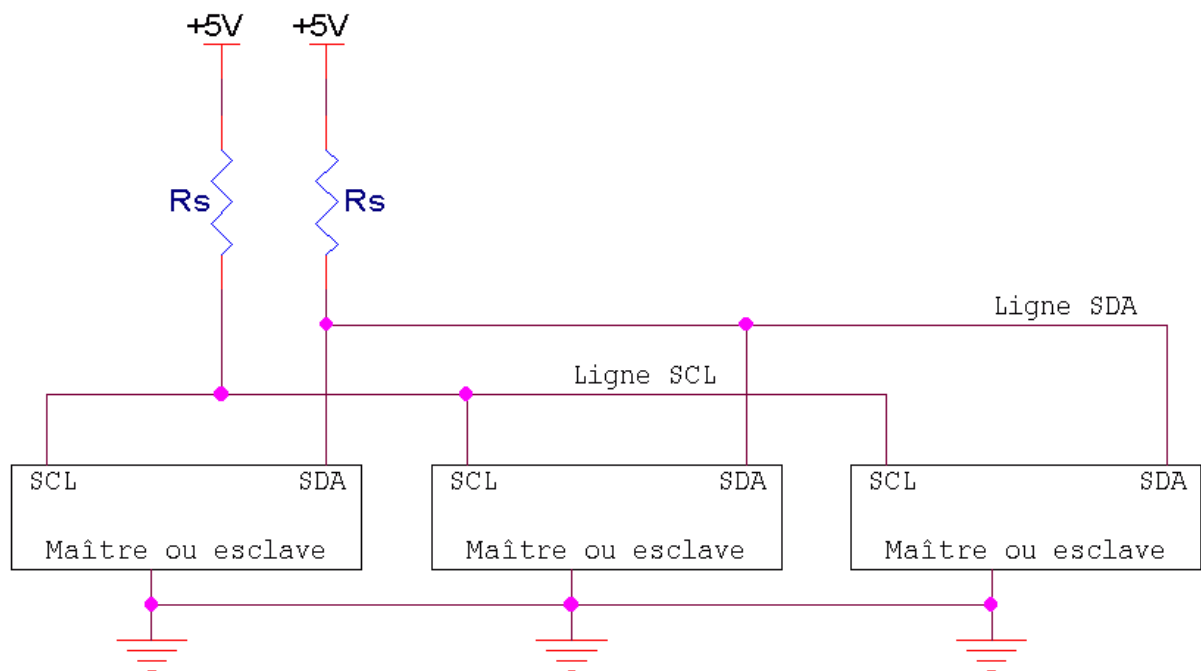
On obtient alors un schéma équivalent théorique du type :



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 PICs 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.

22.2 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.

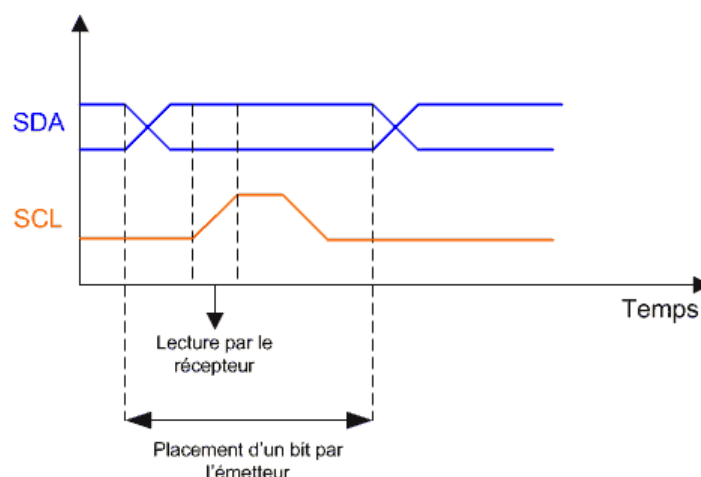
22.2.1 Le bit « ordinaire »

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 :

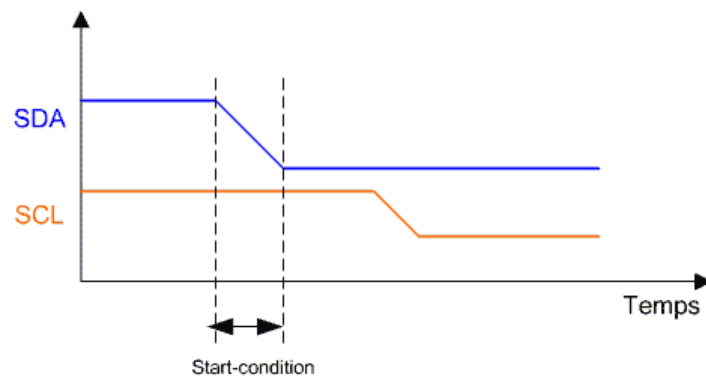


Vous constatez que le bit présent sur la ligne SDA doit être présent avant la montée du signal SCL, et continuer un certain temps avant sa redescente.

22.2.2 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.

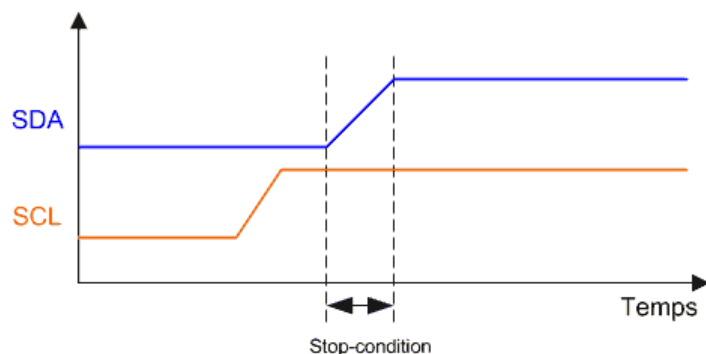


Il existe un dérivé de cette condition, appelé « repeated start condition », qui est utilisé lorsqu'un message en suit directement un autre. Il s'agit donc en fait d'un second start-condition au sein d'un même message. Nous verrons son intérêt plus tard. Sachez à ce niveau qu'un « repeated start-condition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

22.2.3 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 :

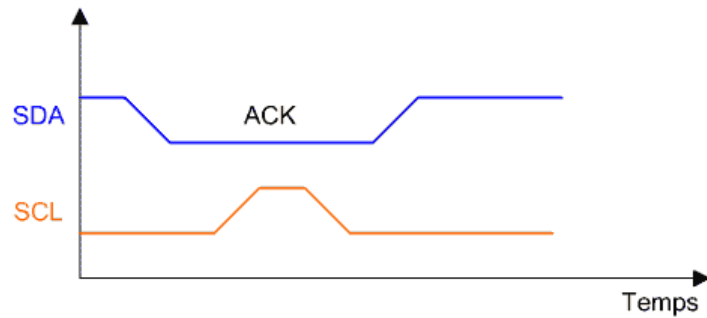
La ligne SDA est ramenée à 1, alors que la ligne SCL se trouve déjà à 1. Voici ce que cela donne :



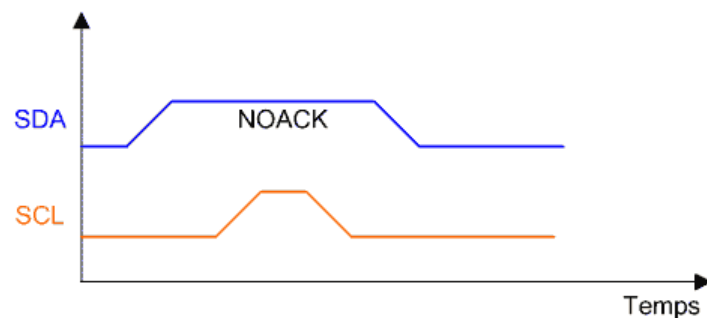
22.2.4 L'acknowledge

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).



22.2.5 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 écrits ($R/W = 0$) 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.

Etant donné que ce bit ne présente aucune particularité, je ne vous donne pas le chronogramme (identique à celui décrit pour le bit « ordinaire »).

22.3 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 :

- B'0000000' : utilisée pour adresser simultanément tous les périphériques (general call address). Les octets complémentaires précisent le type d'action souhaité (par exemple, reset).
- B'0000001' : utilisée pour accéder aux composants CBUS (ancêtre de l' I²C)
- B'0000010' : réservée pour d'autres systèmes de bus
- B'0000011' : réservée pour des utilisations futures
- B'00001xx' : pour les composants haute-vitesse
- B'11111xx' : réservée pour des utilisations futures
- B'11110xy' : 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.

22.3 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.

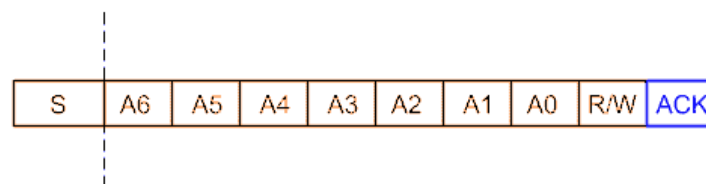
Nous savons déjà que la transmission commence par un « start-condition » (S).

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.

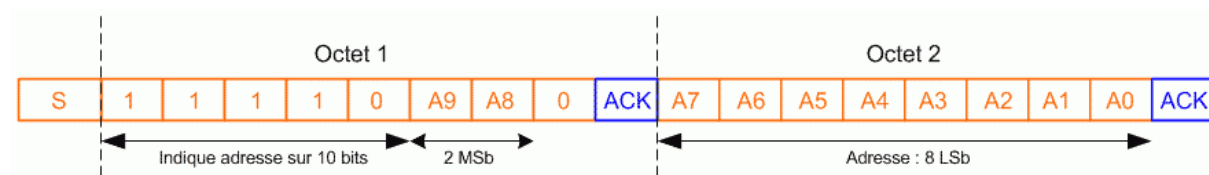
Chaque octet envoyé est toujours accompagné d'un accusé de réception de la part de celui qui reçoit. Un octet nécessite donc $8 + 1 = 9$ impulsions d'horloge sur la pin SCL.

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.



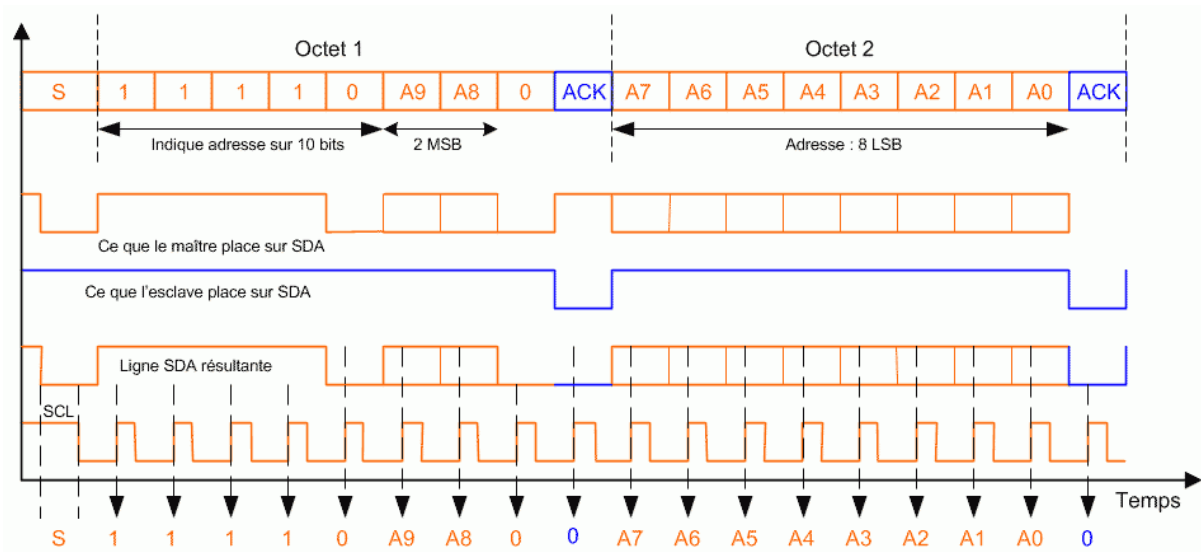
Pour coder une adresse sur 10 bits, on utilisera comme premier octet l'adresse réservée B'11110xy0' qui précise qu'un second octet est nécessaire. Voici ce que donne l'envoi d'une adresse sur 10 bits :



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 B '11110'

Afin d'être certain que vous avez tout compris, je vous donne 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.

Ceci explique que pour un néophyte, ces chronogrammes ne sont pas toujours aisés à comprendre. Nul doute qu'après avoir lu ce chapitre, vous trouverez la norme I²C comme simple à comprendre.

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 :

- Soit nous allons lui envoyer un ou plusieurs octets de donnée
- 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.

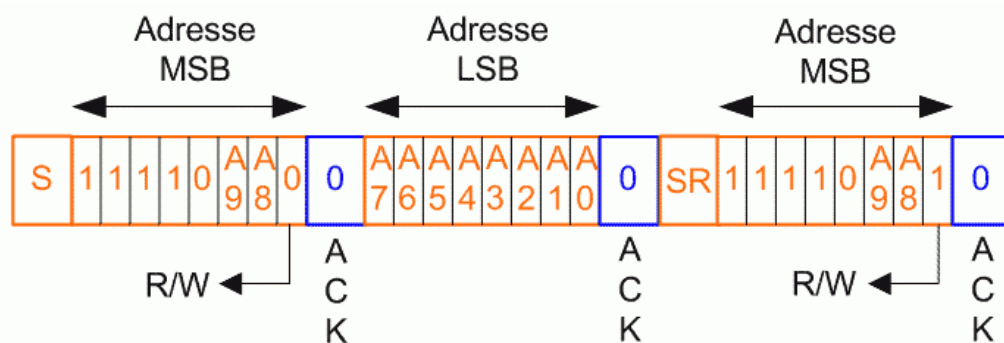
Maintenant, si vous avez suivi, vous devez vous dire : « mais alors, puisque le bit R/W du premier octet d'adresse dans le cas d'une adresse sur 10 bits vaut toujours 0, comment donc effectuer une lecture en mode 10 bits ? ».

En fait, ceci est un peu plus compliqué. 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.



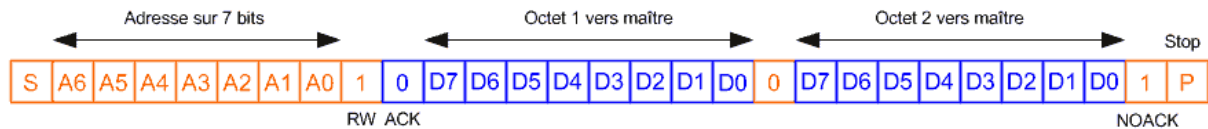
Le cas de la lecture n'est pas plus compliqué. Voici la séquence, toujours pour une adresse codée sur 7 bits :

- 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 xème 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" précédant le stop-condition.

Notez cependant que c'est toujours le maître qui pilote la ligne SCL.

22.4 Événements spéciaux

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

22.4.1 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 μ s se sont écoulées depuis l'envoi du stop-sequence, aucun nouveau start-sequence n'ayant été reçu.

22.4.2 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.

22.4.3 La pause

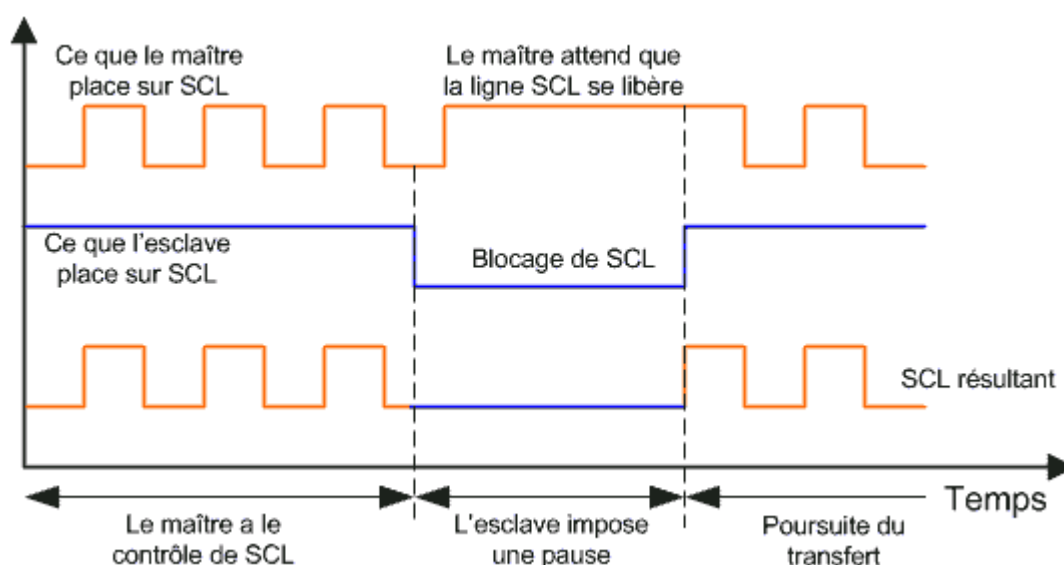
Un autre cas particulier est celui où, lors d'une écriture, l'esclave demande au maître de patienter durant un certain temps (par exemple le temps mis pour traiter l'octet précédent).

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.



22.5 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 μ s, 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 4,7 μ s pour vérifier qu'un autre maître ne dirige pas le bus. Dans la pratique, certains maîtres n'attendent même pas cette limite de 4,7 μ s pour s'assurer que le bus soit libre, ça ne pose pas de problème de fonctionnement particulier.

Vous allez me dire : « c'est bien beau, tout ça, mais si 2 nouveaux maîtres prennent le contrôle en même temps ? »

Et bien, rassurez-vous, c'est également prévu, et géré de façon automatique. Imaginons que le bus soit libre, et que les maîtres 1 et 2 s'accaparent le bus en même temps.

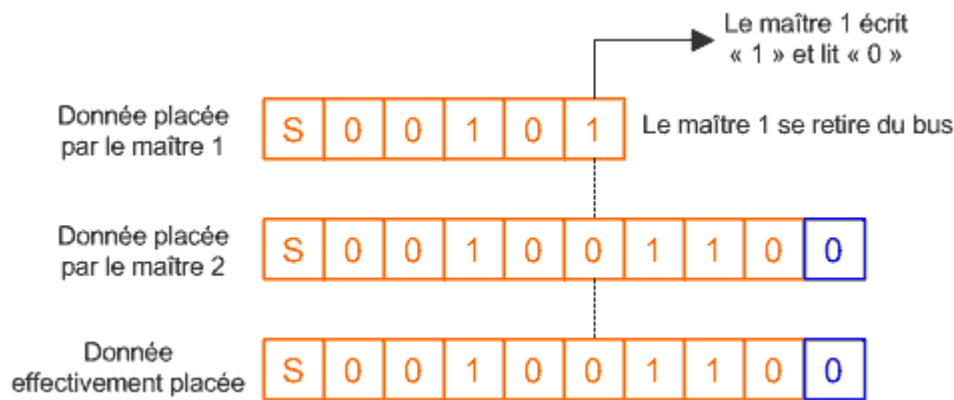
Chaque maître dispose d'une électronique qui vérifie en permanence la correspondance entre les niveaux appliqués sur SDA et SCL, et les niveaux lus en réalité sur ces lignes. S'il y a discordance, c'est qu'un tiers prend le contrôle de cette ligne. Nous en avons déjà vu un exemple lorsque l'esclave imposait 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 B'0010011', 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 vous avez tout compris, 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.

Vous allez me rétorquer qu'il se peut très bien que les 2 maîtres accèdent au même esclave. Fort bien, mais cela ne pose de nouveau aucun problème. Nous avons 2 possibilités :

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.

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.

Si vous êtes attentifs, vous pouvez encore protester, en démontrant que les 2 maîtres peuvent tenter de prendre le contrôle du bus, mais d'une façon légèrement décalée (un des deux réagit un peu plus vite que l'autre). Dans ce cas, il faut en revenir au start-condition, dans lequel le maître fait descendre la ligne SDA à « 0 ». Le maître le plus lent verra donc la ligne SDA descendre avant même qu'il ne l'ai positionnée. Il se retirera donc du bus avant même d'avoir tenté la moindre action.

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.

J'en reste là pour la théorie générale, je pourrais vous parler de l'« I²C high-speed », mais si vous avez compris ce qui précède, et que vous avez besoin d'un tel composant, il vous sera aisé de le mettre en œuvre.

Ce que je préfère faire, par contre, c'est vous donner un exemple concret de circuit courant utilisant L'I²C. J'ai choisi de vous expliquer la façon dont fonctionne une eeprom de type 24C32, 24C64 ou autre (attention, la 24C16 fait exception).

Lisez ceci, car j'utiliserai cette eeprom dans le cadre de notre exercice pratique sur L'I²C.

22.6 Les eeprom I²C

Avec ce chapitre, je vais vous montrer comment interpréter un datasheet de composant I²C. J'ai reçu énormément de courrier concernant ces composants et le mode I²C, ce qui justifie, ce cours étant destiné à rendre service au plus grand nombre, que je m'y attarde.

De plus, je suis persuadé que si vous devenez un adepte des microcontrôleurs, vous aurez un jour ou l'autre besoin d'un composant de ce type.

22.6.1 Caractéristiques générales

Pour que tout le monde parte sur les mêmes bases, je vous fournis le datasheet du 24C64 en format pdf, et en direct de chez Microchip.

Tout d'abord, un œil sur le tableau de la première page nous montre qu'il s'agit d'un circuit qui accepte une fréquence d'horloge de maximum 400KHz. Nous sommes donc en plein dans la norme I²C seconde génération.

Les tensions sont compatibles avec notre PIC. Notez que la norme I²C n'impose pas la valeur de la tension de rappel au niveau haut. Suivant les technologies, vous pourriez vous retrouver avec des composants nécessitant des tensions différentes (12V). Dans ce cas, de simples résistances de protection vous permettraient d'interfacer facilement ces composants avec votre PIC.

Un coup d'œil aux caractéristiques vous informe que vos données sont à l'abri pour une durée minimale garantie de 200 ans. Si vous perdez vos sauvegardes au bout de 100 ou 150 ans, courez donc vite vous plaindre chez Microchip.

1.000.000 de cycles d'effacement/écriture sont de plus garantis (vous avez le temps de procéder à des expérimentations).

Microchip vous informe également sur cette première page, que la 24C64 (ou 24LC64, ou 24AA64...) dispose de 64Kbits de mémoire. Attention, Kbits et non Kbytes. Ceci nous donne en fait 8Kbytes de données mémorisées.

On vous parle également de « random access » et de « sequential access ». La notion de « random » pour « aléatoire », signifie que vous pouvez accéder à n'importe quelle case mémoire, sans avoir à parcourir l'ensemble. Si vous imaginez un livre, cela signifie que vous pouvez ouvrir ce livre à n'importe quelle page.

« Sequential », pour « séquentiel » signifie exactement l'inverse. Vous pouvez écrire ou lire adresse après adresse, dans l'ordre croissant, sans avoir à vous préoccuper de définir à chaque fois l'adresse.

Ceci vous indique donc que les accès offrent beaucoup de souplesse. On vous parle de plus de « page-write ». Nous verrons que cela permet d'écrire toute une série d'octets (sur la même page de 32) en une seule opération d'écriture.

Page 2, vous trouvez les contraintes liées aux signaux, contraintes qui restent bien entendu conformes à la norme I²C. Rien d'inquiétant pour nous.

Le petit **tableau 1-1** en haut à droite, nous donne la fonction des pins de cette eeprom. On y voit que 3 pins, A0, A1, et A2 sont des pins qui sont utilisées par l'utilisateur pour définir la sélection du circuit. Nous y reviendrons. Nous trouvons en plus les 2 lignes d'alimentation, ce qui est logique, et les 2 lignes du bus, SCL et SDA, obligatoires pour tous les circuits I²C.

On notera une dernière pin, WP, qui permet, lorsqu'elle est placée au niveau haut, d'interdire les écritures dans la mémoire, tout en autorisant les lectures. Si cette pin est connectée à la masse, ou n'est pas connectée du tout, les opérations d'écritures sont autorisées.

Nous arrivons au **tableau 1-3 de la page 3**. Ce tableau poursuit les caractéristiques précédentes. Jetez un œil sur la ligne « bus free time ». Vous voyez qu'il est indiqué que le bus doit être libre depuis 4700 ns avant qu'une nouvelle transmission puisse avoir lieu. Nous retrouvons donc bien nos fameuses 4,7µs entre le stop-condition et le moment où les composants considèrent que le bus est libre.

Encore plus bas dans le tableau, vous voyez « input filter ... ». Sachez que tous les composants I²C sont munis d'un filtre destiné à éliminer les parasites. Notre PIC en est également pourvu, j'en reparlerai au moment voulu.

L'avant-dernière ligne nous indique qu'une opération d'écriture nécessite 5ms maximum, ce qui est énorme par rapport aux vitesses mises en jeu. Une opération d'écriture est l'écriture d'un seul octet, ou d'un groupe d'octets (maximum une page). **Vous avez donc intérêt à utiliser les procédures d'écriture par page lorsque vous avez plusieurs octets consécutifs à écrire.** J'expliquerai plus loin comment savoir que l'écriture est terminée.

D'ailleurs, le « plus loin » ne l'est pas tant que ça. Si vous tournez la page, vous voyez la description des pins dont j'ai déjà parlé, et les caractéristiques du bus, qui sont bien entendu celles du bus I²C. Un encadré est important, il signale que l'eprom n'envoie pas son accusé de réception (donc le maître lira un « NOACK ») si un cycle précédent d'écriture n'est pas terminé. Voici donc comment savoir si le cycle est oui ou non terminé.

La page 5 montre comment se passe le transfert, c'est-à-dire de façon strictement conforme au protocole I²C (heureusement).

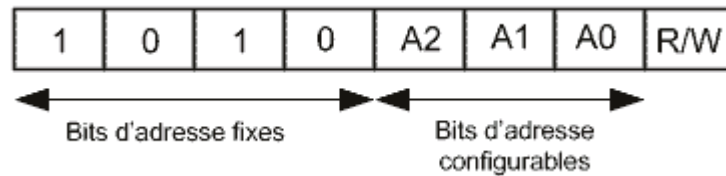
22.6.2 Octet de contrôle et adresse esclave

On arrive aux choses sérieuses avec la page 6. **La figure 5-1** vous montre le format de l'octet de contrôle. Octet de contrôle ? Qu'est-ce que c'est que cette nouveauté ?

Pas de panique, Il s'agit tout simplement de l'octet d'adressage, donc le premier octet émis par le maître et le premier reçu par l'esclave, c'est-à-dire l'eprom.

Vous allez me dire que je n'ai dit nulle part que l'eprom est un composant esclave. C'est vrai, mais il faut faire de temps en temps fonctionner sa tête. En fait, ce n'est pas l'eprom qui décide si on la lit ou on l'écrit. Ce n'est pas non plus l'eprom qui décide à quel moment envoyer ou recevoir des données. De plus, la figure 5-1 nous montre que l'eprom a une adresse, et c'est le composant esclave qui dispose d'une adresse. Tout ceci nous démontre que l'eprom est un circuit I²C esclave.

Voyons l'octet dénommé « de contrôle » par Microchip. Bien entendu, nous, nous savons qu'il s'agit tout simplement de l'octet d'adresse, envoyé directement après le start-condition :



Microchip a utilisé le terme « contrôle » probablement pour ne pas que l'utilisateur confonde l'adresse de l'eeprom (contrôle) et l'adresse de l'octet concerné dans l'eeprom.

Nous constatons immédiatement qu'on nous impose les 4 premiers bits de l'adresse de l'eeprom, les 3 bits restants étant à notre convenance. Comme toujours, cette adresse se termine par le bit R/W. Nous pouvons en déduire que :

- Notre eeprom sera adressée obligatoirement dans la plage : B'1010000' à B'1010111'.
- C'est une adresse assez élevée, les eeproms n'étant pas considérés comme des composants prioritaires
- Nous avons 8 possibilités de choix de l'adresse, fonctions de A2, A1, et A0.
- Nous pouvons de ce fait placer, sans artifice, un maximum de 8 eeproms sur notre bus.
- Nous avons besoin d'une possibilité de choisir ces 3 bits.
- Nous avons affaire à un adressage sur 7 bits.

Arrivés à ce stade, vous devez savoir que les bus I²C sont destinés à recevoir une foule de composants différents. Or, chaque composant doit avoir une adresse distincte. Les constructeurs de circuits intégrés se sont donc mis d'accord pour réserver certaines plages d'adresses à des composants particuliers. Ceci permet d'éviter les conflits malheureux qui rendraient impossibles certaines cohabitations.

Les adresses commençant par « 1010 » sont en général réservées aux eeproms. Il nous reste à définir les 3 bits d'adressage pour notre ou nos eeproms.

En fait, c'est tout simple, nous avons vu que notre eeprom dispose de 3 broches A0, A1, et A2. C'est le niveau placé sur ces pins qui définira l'adresse.

Par exemple, si vous placez du +5V (« 1 ») sur la pin A1, alors le bit A1 de l'adresse vaudra « 1 ». Si vous reliez cette même pin à la masse (« 0 »), alors le bit A1 de l'adresse vaudra « 0 ». Vous choisissez donc l'adresse finale de votre eeprom au moment de la connexion physique de celle-ci sur son circuit.

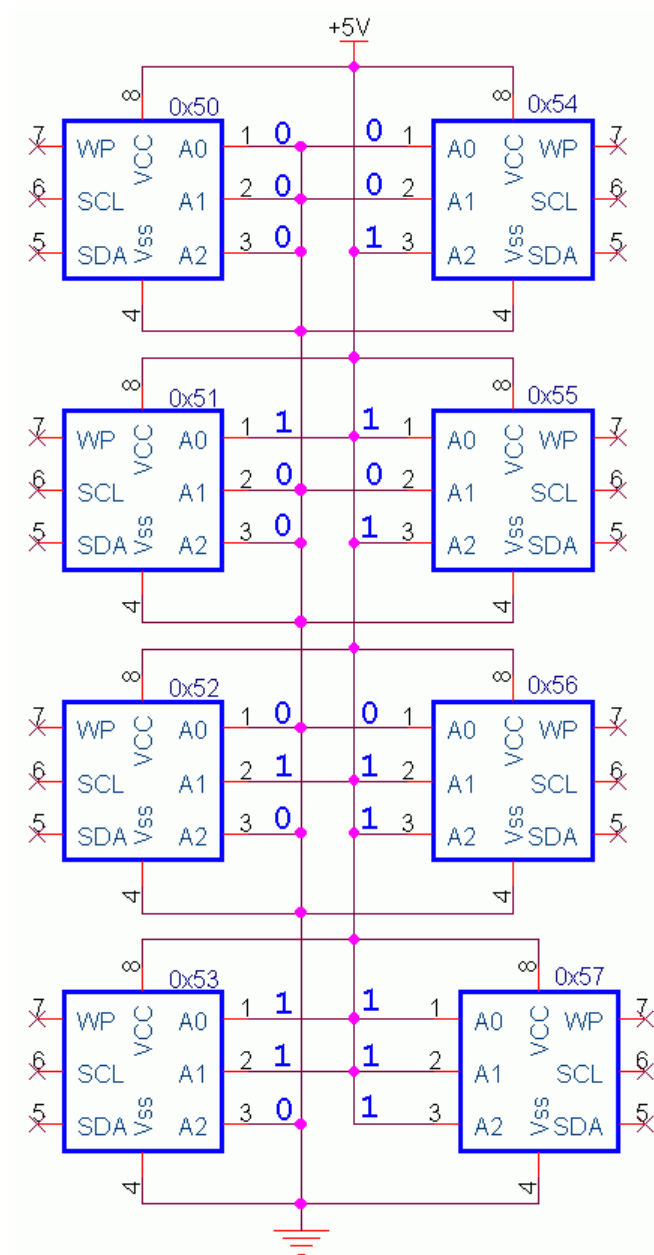
L'adresse est donc configurable au moment de la création du circuit, et ne peut pas être modifiée par logiciel.

La plus petite adresse possible sera : B'1010000', soit D'80' ou 0x50. La plus grande sera B'1010111', soit D'87 ou 0x57. Voyons donc comment interfacer nos 8 eeproms sur un circuit, et quelles seront leurs adresses respectives.

Je n'ai représenté que les lignes +5V et 0V. Les lignes SDA et SCL respectives de chaque circuit devront bien évidemment être reliées entre elles et au bus I²C.

La pin WP sera laissée non connectée, ou reliée à la masse pour une utilisation normale, et connectée au +5V pour protéger l'EEPROM. Bien entendu, il n'est pas interdit de connecter ces pins WP à une pin de votre PIC, de façon à commander la protection en écriture par logiciel.

Si vous avez besoin de plus de 8 EEPROMs, vous pouvez également utiliser des mécanismes de sélection de groupes d'EEPROM à partir de votre PIC, en laissant ou non passer l'horloge SCL, mais ceci déborde du cadre de ces explications, et du bus I²C dans son sens strict.



Vous voyez que dans cet exemple, chacune des 8 EEPROMs répondra à une adresse unique, comprise entre 0x50 et 0x57. Je n'ai pas représenté les interconnexions des lignes SCL et SDA, mais je vous en ai déjà parlé tout au début du chapitre.

22.6.3 Sélection d'une adresse interne

Nous venons de définir l'adresse I²C, c'est-à-dire l'adresse physique sur le bus de chaque circuit eeprom. Mais, lorsqu'on opère dans une mémoire, il nous faut encore savoir à quelle adresse on lit ou on écrit parmi les 8Koctets possibles qu'elle contient.

Il ne faut pas, à ce niveau, confondre l'adresse du circuit, et l'adresse de la donnée à lire ou à écrire.

Pour sélectionner une adresse dans une eeprom, il suffit, directement après le premier octet d'adressage du circuit, d'écrire dans l'eeprom sélectionnée l'adresse, codée sur 2 octets, à laquelle devra s'initialiser son pointeur interne.

Donc, même si on doit lire un emplacement mémoire, on doit écrire l'adresse (R/W = 0). J'insiste donc une nouvelle fois : l'adresse sur laquelle on travaille dans l'eeprom doit être écrite par le maître.

Toutes les opérations sur les eeprom se réalisent en effet sur l'emplacement désigné par un pointeur interne. Ceci fonctionne exactement comme pour l'adressage indirect des PICs. Avant de pouvoir utiliser la case pointée par « INDF », vous deviez en effet initialiser le pointeur « FSR », il en est de même ici.

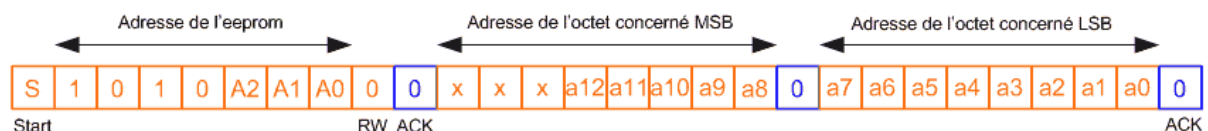
La sélection d'une adresse s'effectue donc fort logiquement de la façon suivante :

- Le maître envoie le start-condition
- Le maître envoie le premier octet avec l'adresse de l'eeprom et le bit R/W à 0 (écriture)
- L'eeprom envoie l'accusé de réception (ACK)
- Le maître envoie les 2 octets de l'adresse de l'octet concerné.

Notez que l'adresse de l'octet doit tenir dans la plage des 8Koctets disponibles, ce qui donne une adresse codée sur 13 bits (de B'0000000000000' à B'1111111111111', soit de 0x00 à 0x1FFF). Comme on envoie 2 octets, cela nous donne 16 bits.

L'adresse sera alignée vers la droite, ce qui implique que les 3 bits à gauche du premier octet (poids fort) seront inutilisés. On les placera de préférence à 0, pour toute compatibilité future avec une eeprom de capacité supérieure.

Autrement dit :



J'ai noté l'adresse de l'esclave en majuscules (Ax) et la valeur d'initialisation du pointeur de l'eeprom (adresse de l'octet) en minuscules (ax). Ceci afin de ne pas vous embrouiller avec ces différentes notions d'adressage.

A ce niveau, il faut se souvenir que le premier « ACK » envoyé par l'esclave signale si celui-ci est prêt. Si l'esclave répond à ce moment « NOACK » (donc ne répond pas), c'est

qu'il n'a pas terminé l'opération précédente, il faut donc recommencer la tentative jusqu'à ce que la réponse soit « ACK ».

Evidemment, si vous adressez un esclave qui n'existe pas, vous obtiendrez également un « NOACK », puisque personne n'aura placé la ligne SDA à « 0 ». Dans ce cas, vous risquez d'attendre longtemps.

22.6.4 Ecriture d'un octet

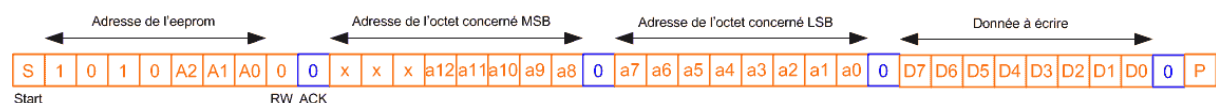
Nous savons maintenant choisir une eeprom, et sélectionner une adresse dans cette eeprom. Nous allons voir comment écrire une donnée à l'emplacement choisi.

La procédure est très simple, elle consiste à initialiser le pointeur d'adresse, puis à envoyer directement après l'octet de donnée. L'eeprom sait parfaitement que l'octet suivant devra être écrit à l'emplacement spécifié. C'est bien entendu le bit R/W qui indique à l'eeprom qu'il s'agit d'une écriture.

Donc, dans l'ordre :

- Le maître envoie le start-sequence
- Le maître envoie l'adresse de l'eeprom avec le bit R/W à 0 (écriture)
- L'eeprom, si elle est prête, répond « ACK »
- Le maître envoie le poids fort de l'adresse concernée
- L'eeprom répond « ACK »
- Le maître envoie le poids faible de l'adresse concernée
- L'eeprom répond « ACK »
- Le maître envoie la donnée à écrire en eeprom
- L'eeprom répond « ACK »
- Le maître envoie le stop-sequence.

Sous forme de chronogramme simplifié :



Pour faire un peu plus concret, imaginons que nous voulions écrire la valeur 0x3D à l'adresse 0x1A58 de notre EEPROM, sur laquelle les pins A2 et A0 sont connectées à la masse et A1 au +5V. Voici ce que nous devons envoyer :



Et voilà, rien de plus simple. Vous savez maintenant comment écrire un octet dans une eeprom 24c64 ou assimilée.

22.6.5 Ecriture par page

Nous avons vu que nous pouvions écrire un octet à n'importe quelle adresse de notre eeprom. Mais cette opération d'écriture nécessite à l'eeprom un temps de traitement interne de 5ms. Durant ces 5ms, l'eeprom répondra « NOACK » à toute nouvelle tentative d'écriture.

Pour écrire nos 8×1024 octets, il nous faudrait, en pratiquant de la sorte, $8 \times 1024 \times 5\text{ms}$, soit plus de 40 secondes. Cette méthode n'est donc guère pratique pour les écritures multiples d'octets.

Heureusement, nous avons la possibilité d'écrire des séries d'octets en une seule opération d'écriture. Nous parlerons ici du cas de la 24C64, dont chaque page fait 32 octets. Cette taille varie en fonction du type d'eeprom, une 24C512 dispose de pages de 128 octets, par exemple.

Ceci est rendu possible par un mécanisme interne à l'eeprom, qui incrémente automatiquement les 5 bits de poids faible du pointeur d'adresse (b0 à b4). Attention, il n'y a pas de report sur le bit « b5 » pour la 24C64. (en fait, l'eeprom utilise un buffer interne, mais c'est transparent pour l'utilisateur).

Il suffit donc de placer les octets de donnée les uns à la suite des autres, ils seront écrits automatiquement dans les adresses successives de l'eeprom.

Donc, nous aurons des trames du style :



Supposons que l'adresse définie soit 0x1180 :

L'octet 0 sera écrit à : $0x1180 + 0x00 = 0x1180$

L'octet 1 sera écrit à : $0x1180 + 0x01 = 0x1181$

.....

.....

L'octet 31 sera écrit à : $0x1180 + 0x1F = 0x119F$

Attention cependant au piège. Si vous définissez une adresse de base de 0x115E, par exemple :

$0x115E = B'0001\ 0001\ 0101\ 1110'$

L'octet 0 sera écrit à l'adresse 0x115E

L'octet 1 sera écrit à l'adresse 0x115F

A quelle adresse sera écrit l'octet 2 ? Si vous me répondez : « 0x1160 », vous êtes tombés dans le piège. Voyons en binaire :

L'octet 2 sera écrit à l'adresse :

$B'0001\ 0001\ 0101\ 1110'$

$$\begin{array}{r}
 + \text{ B'0000 0000 0000 0010' } \\
 \hline
 = \text{ B'0001 0001 0100 0000' }
 \end{array}$$

En effet, je vous ai dit que l'incrémentation ne concernait que les 5 bits de poids faible (b4 à b0), **le bit b5 n'est donc pas affecté par l'incrémentation.**

Ceci vous précise que l'octet 2 sera écrit à l'adresse 0x1140.

Donc, l'incrémentation ne s'effectue correctement que si l'ensemble des adresses des octets écrits se trouvent dans la même page de 32. Le cas optimal est obtenu si l'adresse passée comme second et troisième octet de la trame est une adresse dont les 5 bits de poids faible sont égaux à 0. Dans ce cas, on se trouve en début de page, et on peut donc écrire 32 octets simultanément.

Comme l'écriture de ces 32 octets ne nécessite pas plus de temps que l'écriture d'un seul, le temps nécessaire pour remplir toute notre eeprom sera divisée par un facteur de 32, soit moins de 2 secondes.

Partant de là, il vous est très simple de découvrir quelle méthode utilise votre programmeur d'eeprom.

Je résume les 2 méthodes d'écriture :

- Soit vous envoyez l'adresse d'écriture suivie par l'octet à écrire
- Soit vous envoyez l'adresse de départ suivie par un maximum de 32 octets, qui seront écrits séquentiellement dans la mémoire.

Tout ce qui précède est valable pour la 24C64. Sur une 24C512, par exemple, vous pourrez envoyer 128 octets de data, le bit 7 du pointeur d'adresse interne n'étant pas affecté par l'écriture. Renseignez-vous selon le type d'eeprom que vous utiliserez.

22.6.6 La lecture aléatoire

Aléatoire (random) est à prendre ici dans le sens de « n'importe où », et non en comme signifiant « au hasard ». Cette dernière définition n'aurait d'ailleurs aucun intérêt.

Pour lire un octet à une adresse spécifique, il faut réaliser les opérations suivantes :

- Initialiser le pointeur d'adresse de l'eeprom
- Lire l'octet concerné
- Envoyer un NOACK pour signifier la fin de la transaction.

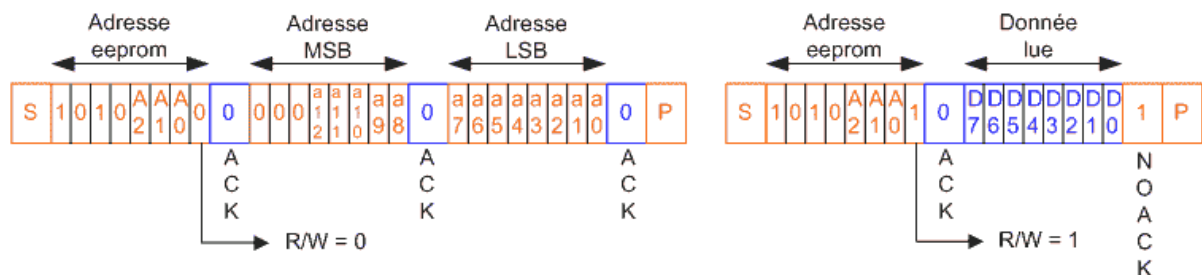
Donc, vous voyez que **ceci nécessite une opération d'écriture** (le pointeur d'adresse), **suivie par une opération de lecture** (l'octet à lire).

Comme le bit R/W définissant le sens de transfert n'existe que dans le premier octet suivant le start-condition, vous voyez déjà qu'il vous faudra envoyer 2 fois ce premier octet, donc 2 fois le start-condition.

Vous pouvez procéder en une seule étape, ou en 2 étapes successives. Si vous utilisez 2 étapes, vous aurez :

- Le maître envoie le start-condition (S)
- Le maître envoie l'adresse de l'eeprom avec le bit R/W à 0 (écriture)
- L'eeprom envoie l'accusé de réception (ACK)
- Le maître envoie l'octet fort de l'adresse à lire
- L'eeprom envoie l'accusé de réception (ACK)
- Le maître envoie l'octet faible de l'adresse à lire
- L'eeprom envoie l'accusé de réception (ACK)
- Le maître envoie le stop-condition (P)
- Le maître envoie le start-condition (S)
- Le maître envoie l'adresse de l'eeprom avec le bit R/W à 1 (lecture)
- L'eeprom envoie l'accusé de réception (ACK)
- L'eeprom envoie l'octet à lire
- Le maître envoie un « NOACK »
- Le maître envoie le stop-condition (P)

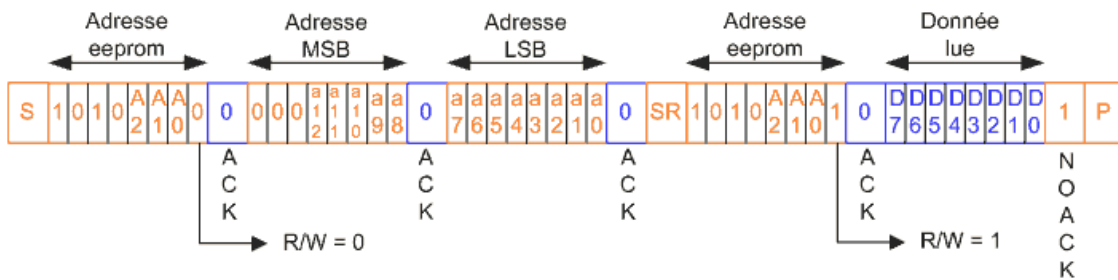
Autrement dit :



Si vous utilisez la méthode conseillée, c'est-à-dire en une seule étape, vous remplacez simplement le stop-condition suivi du start-condition par un « repeated start-condition » (SR), ce qui nous donne :

- Le maître envoie le start-condition (S)
- Le maître envoie l'adresse de l'eeprom avec le bit R/W à 0 (écriture)
- L'eeprom envoie l'accusé de réception (ACK)
- Le maître envoie l'octet fort de l'adresse à lire
- L'eeprom envoie l'accusé de réception (ACK)
- Le maître envoie l'octet faible de l'adresse à lire
- L'eeprom envoie l'accusé de réception (ACK)
- Le maître envoie le repeated start-condition (SR)
- Le maître envoie l'adresse de l'eeprom avec le bit R/W à 1 (lecture)
- L'eeprom envoie l'accusé de réception (ACK)
- L'eeprom envoie l'octet à lire
- Le maître envoie un « NOACK »
- Le maître envoie le stop-condition (P)

Ou, sous forme de chronogramme simplifié :



Donc, pour résumer, vous écrivez l'adresse, puis vous lisez la donnée. Le passage entre lecture et écriture nécessite de devoir réenvoyer le start-condition (SR).

22.6.7 La lecture de l'adresse courante

Vous avez vu que si nous décidons d'effectuer une lecture aléatoire en 2 étapes, nous commençons par envoyer l'adresse de lecture, puis nous procédons à la lecture en elle-même. En fait, si vous réfléchissez, ceci équivaut à dire :

- J'initialise le pointeur d'adresse de l'eeprom
- Je lis l'adresse courante.

En effet, le pointeur d'adresse reste mémorisé dans l'eeprom. A tout moment, vous disposez donc de la possibilité de lire l'adresse qui est actuellement pointée par le pointeur d'adresse.

REMARQUE : La lecture d'un octet provoque automatiquement l'incréméntation du pointeur d'adresse. Donc, la prochaine lecture renverra l'octet suivant, et ainsi de suite. Si le pointeur dépasse la capacité de la mémoire, il recommence à 0.

22.6.8 La lecture séquentielle

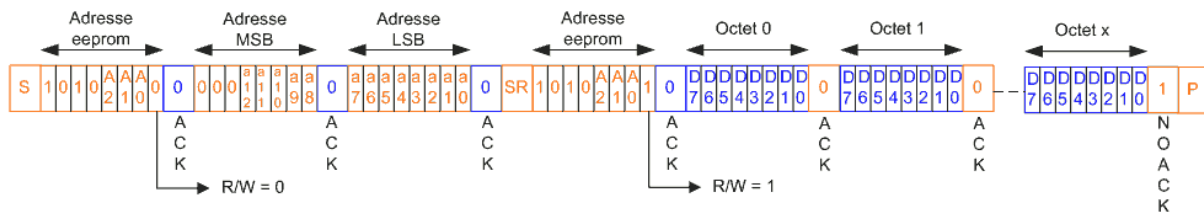
Et bien, de nouveau, c'est tout simple. Cette procédure permet de lire plusieurs octets consécutifs. Ceci fonctionne exactement comme pour l'écriture par page, excepté que nous retrouvons pas ici les contraintes de page. Nous pouvons donc lire l'intégralité de la mémoire eeprom en une seule opération, nous ne sommes pas limités à 32 octets.

Nous procédons comme pour la lecture aléatoire, mais au lieu que le maître envoie un « NOACK » après avoir lu l'octet, il envoie un « ACK », qui signale à l'eeprom qu'il souhaite lire l'octet suivant. Ce n'est qu'au dernier octet que le maître enverra un « NOACK ».

La procédure est la suivante :

- Le maître envoie le start-condition (S)
- Le maître envoie l'adresse de l'eprom avec le bit R/W à 0 (écriture)
- L'eprom envoie l'accusé de réception (ACK)
- Le maître envoie l'octet fort de l'adresse à lire
- L'eprom envoie l'accusé de réception (ACK)
- Le maître envoie l'octet faible de l'adresse à lire
- L'eprom envoie l'accusé de réception (ACK)
- Le maître envoie le repeated start-condition (SR)
- Le maître envoie l'adresse de l'eprom avec R/W à 1 (lecture)
- L'eprom envoie l'accusé de réception (ACK)
- L'eprom envoie l'octet à lire
- **Le maître envoie un « ACK »**
- L'eprom envoie l'octet suivant à lire
- **Le maître envoie un « ACK »**
- L'eprom envoie l'octet suivant à lire
-
-
- L'eprom envoie l'octet suivant à lire
- **Le maître envoie un « NOACK »**
- Le maître envoie le stop-condition (P)

Ce qui nous donne :



22.6.9 Le cas de la 24C16

Bon, juste un mot sur l'eprom 24C16, dont je vous ai dit qu'elle n'était pas compatible. J'en parle pour m'éviter du courrier, car j'ai comme une intuition à ce sujet.

Sur la 24C16, les pins A0, A1, et A2 existent, mais ne sont pas connectées à l'intérieur de l'eprom. Elles ne peuvent donc pas servir pour sélectionner l'adresse.

Première déduction : **on ne peut mettre qu'une et une seule eeprom 24C16 par bus.**

Nos bits A2, A1, et A0 du premier octet sont donc libres. Ils sont en fait utilisés comme bits de poids fort de l'adresse de l'octet concerné. L'eprom 2416 contient 16Kbits de mémoire, donc 2Koctets. L'adresse d'un octet se code de ce fait sur 11 bits. Donc, si vous me suivez toujours :

Sur une 24C64, on a :

- start-condition
- premier octet = 1010 A2 A1 A0, avec A2/A0 adresse physique de l'eprom
- second octet = MSB adresse = x x x a12 a11 a10 a9 a8
- troisième octet = LSB adresse = a7 a6 a5 a4 a3 a2 a1 a0

Sur une 24C16, on aura

- start-condition
- premier octet = 1010 a10 a9 a8, avec a10/a8 = 3 bits de poids fort de l'adresse de l'octet
- second octet = a7 a6 a5 a4 a3 a2 a1 a0

Autrement dit, l'eprom 24C16 répond en fait à 8 adresses différentes pour un seul circuit. C'est une astuce qui permet de gagner un octet en émission, mais se paye par l'occupation de 8 adresses pour un seul circuit. C'est pour ça que vous lisez souvent que la 24C16 n'est pas compatible avec les autres eprom (24c32, 24c64...).

Vous pouvez donc remplacer sans problème dans un montage une 24C32 par une 24C64, mais vous ne pouvez remplacer une 24c16 par une autre 24cxx que si vous modifiez le logiciel du maître en conséquence.

22.6.10 Conclusions

Nous venons d'étudier tout le datasheet de l'eprom 24C64. Vu le nombre important de questions que j'ai reçues à ce sujet, il m'a paru nécessaire d'effectuer cette analyse. Ceci vous ouvrira de plus les portes des nombreux autres périphériques I²C.

Ceci clôture notre étude théorique du bus I²C. Nous allons maintenant étudier comment ces options sont disponibles sur nos 16F87x.

Notes : ...

23. Le module MSSP en mode I²C

23.1 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. Je ne parlerai pas du fonctionnement de ces filtres, car cela sort du cadre de cet ouvrage et ne vous apporterait rien de plus. L'important étant que les PICs sont compatibles I²C. Pour rester simple, disons qu'en sortie, ceci permet de respecter les chronologies, et qu'en entrée cela réduit les risques de prise en compte des parasites.

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 100 KHz, 400 KHz, 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 », dont nous avons déjà parlé à plusieurs reprises, et qui permettent une détection plus sécurisée des signaux parasites.

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.

23.2 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, en cas de doute, consultez le datasheet, ou l'étude du mode SPI dans cet ouvrage.

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

b7 : SMP : Slew rate control set bit (1 pour 100 KHz et 1MHz, 0 pour 400 KHz)
b6 : CKE : Input levels set bit (0 = bus I²C, 1 = bus SMBUS)
b5 : D_A : Data / Address bit (0 = adresse, 1 = data)
b4 : P : stop bit
b3 : S : Start bit
b2 : R_W : Read/Write bit information (0 = write, 1 = read)
b1 : UA : Update adress
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 start-condition.

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 start-condition). 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 inversement).

Nous verrons en temps utile comment cela se passe. 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.

23.3 Le registre SSPCON

De nouveau, nous allons voir que tous les bits de ce registre sont utilisés.

SSPCON en mode I²C

b7 : WCOL : Write **COL**lision detect bit
b6 : SSPOV : **SSP O**verflow indicator bit
b5 : SSPEN : **SSP E**nable select bit
b4 : CKP : Pause (si 0)
b3 : SSPM3 : SSP select bit M3
b2 : SSPM2 : SSP select bit M2
b1 : SSPM1 : SSP select bit M1
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 I ² C esclave avec adresse sur 7 bits
0	1	1	1	mode I ² C esclave avec adresse sur 10 bits
1	0	0	0	mode I ² C 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 maître avec adresses 7 bits, interruptions sur événements S et P
1	1	1	1	Mode maître avec adresses 10 bits, interruptions sur événements S et P

Nous ne parlerons que des 3 premiers modes, les autres n'étant pas nécessaires. De plus, je ne sais pas ce que signifie « mode maître avec adresse », étant donné qu'un maître n'a pas d'adresse, et que rien n'empêche le même maître d'adresser des esclaves sur 7 et 10 bits d'adresse.

J'ai posé la question à Microchip, leur réponse a été que les modes I²C « firmware controlled » permettent de prendre la gestion I²C en charge par le logiciel. Avouez que, comme précision, ça nous avance beaucoup. Ma question était pourtant précise. A mon avis, le technicien n'en savait pas plus que moi à ce sujet.

23.4 Le registre SSPADD

Puisque je viens d'en parler, autant continuer par celui-ci. Son nom signifie **S**ynchronous **S**erial **P**ort **A**DDress register. Il a deux fonctions complètement distinctes, selon que l'on travaille en I²C maître ou en I²C esclave.

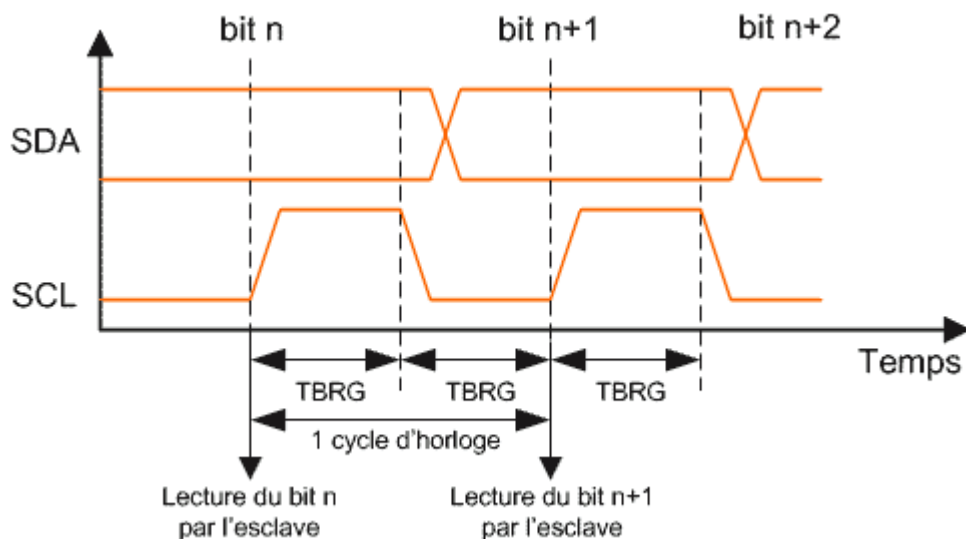
Commençons par le mode esclave. C'est dans ce registre que vous placerez l'adresse à laquelle devra répondre votre PIC. Si vous décidez de travailler en mode 10 bits, vous placerez les bits de poids fort dans SSPADD. Vous serez alors prévenus par le bit UA du registre SSPSTAT qu'il est temps d'y placer les bits de poids faible afin de compléter l'adresse, et inversement.

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 T_{BRG} durant lequel la ligne SDA est positionnée, SCL étant à 0, ensuite la ligne SCL est mise à 1 durant un temps T_{BRG} . 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 T_{BRG} . Comme BRG décrémente 2 fois par cycle d'instructions (donc tous les 2 T_{osc}), et qu'il met 2 T_{osc} supplémentaires pour se recharger, on peut dire que le temps de durée d'un bit est de :

$$\text{Durée d'un bit} = T_{osc} * 2 * 2 * (SSPADD + 1)$$

La fréquence de l'horloge est donc déterminée par la formule :

$$F_{SCL} = F_{osc} / (4 * (SSPADD + 1))$$

En général, vous connaissez la fréquence (100KHz, 400KHz, 1MHz), et vous cherchez à déterminer SSPADD. D'où :

$$SSPADD = (F_{osc} / (F_{SCL} * 4)) - 1$$

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 remplacez B'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.

23.5 Le registre SSPCON2

Voici un registre que nous n'avons pas encore examiné. Voyons donc ce qu'il contient, le bit 7 ne concerne que le mode esclave, tandis que les autres bits ne concernent que le mode maître :

SSPCON2

b7 : GCEN	: General Call ENable bit (réponse appel général)
b6 : ACKSTAT	: ACKnowledge STATus bit (état du bit ACK reçu)
b5 : ACKDT	: ACKnowledge DaTa bit (valeur transmise)
b4 : ACKEN	: ACKnowledge ENable bit (placer l'acknowledge)
b3 : RCEN	: ReCeive ENable bit (lance la réception)
b2 : PEN	: stop-condition ENable bit (générer stop-condition)
b1 : RSEN	: Repeated Start-condition ENable bit (générer SR)
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.

23.6 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.

23.7 Le module MSSP en mode I²C maître

Nous voici maintenant dans l'utilisation concrète de notre PIC configurée en I²C maître. Je ne traite pour l'instant que le cas où votre PIC est le seul maître du bus (mode mono-maître), 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.

23.7.1 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 :

- 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.
- 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

$$\text{SSPADD} = (\text{Fosc} / (\text{F}_{\text{SCL}} * 4)) - 1$$

Dans notre cas :

$$\text{SSPADD} = (20 \text{ MHz} / (400\text{KHz} * 4)) - 1$$

$$\text{SSPADD} = (20.000 / (400 * 4)) - 1$$

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 :

- On positionne le bit SSPEN pour mettre le module en service
- On choisit SSMPx = 1000 pour le mode I²C maître

Tout ceci nous donne un code du style :

```
Init
BANKSEL  TRISC      ; sélectionner banque 1
bsf      TRISC,3     ; SCL en entrée (mis par défaut sur un reset)
bsf      TRISC,4     ; SDA en entrée (idem)
movlw    B'00000000' ; slew rate control en service, mode I2C
movwf    SSPSTAT     ; dans SSPSTAT
movlw    D'12'       ; valeur de recharge du BRG
movwf    SSPADD      ; dans registre de recharge
bcf      STATUS,RP0  ; passer banque 0
movlw    B'00101000' ; module MSSP en service en mode I2C master
movwf    SSPCON      ; dans registre de contrôle
```

23.7.2 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 le pouvoir la lancer (puisque vous êtes également dans ce cas sorti de la commande précédente sans attendre).

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, car la gestion des interruptions dans ce mode n'est pas vraiment simple

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)

Voici donc un exemple de sous-routine qui attend que l'opération précédente soit terminée :

```
I2C_idle
    BANKSEL    SSPSTAT    ; passer en banque 1
I2C_idle2
```

```

    clrwdt          ; effacer watchdog
    btfsc SSPSTAT,R_W ; tester si émission en cours
    goto I2C_idle2   ; oui, attendre
i2C_idle3
    clrwdt          ; effacer watchdog
    movf SSPCON2,w    ; charger registre SSPCON2
    andlw 0x1F        ; conserver ACKEN,RCEN,PEN,RSEN,et SEN
    btfss STATUS,Z    ; tester si tous à 0 (aucune opération en cours)
    goto I2C_idle3    ; non, attendre
    bcf STATUS,RP0    ; repasser en banque 0
    return           ; rien en cours, on peut sortir

```

Bien évidemment, si vous n'avez pas activé le watchdog, vous pouvez vous passer des instructions « clrwdt ».

23.7.3 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

Voici le code correspondant :

```

I2C_start
    call I2C_idle    ; attendre fin de l'opération précédente (voir plus haut)
    BANKSEL SSPCON2 ; passer en banque 1
    bsf SSPCON2,SEN  ; lancer start-condition
    bcf STATUS,RP0   ; repasser en banque 0
    return           ; retour

```

Et son alternative, comme expliqué précédemment :

- On lance le start-condition
- On attend que le start-condition soit terminé

```

I2C_start
    BANKSEL SSPCON2 ; passer en banque 1
    bsf SSPCON2,SEN ; lancer le start-condition
I2C_start2
    clrwdt          ; effacer watchdog
    btfsc SSPCON2,SEN ; start-condition terminé ?
    goto I2C_start2 ; non, attendre
    bcf STATUS,RP0  ; oui, repasser en banque 0
    return          ; et retour

```

23.7.4 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.

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 « 0 » (écriture) dans SSPBUF, ce qui lance l'émission

I2C_adress

```
call I2C_idle ; attendre fin de l'opération précédente
movlw SLAVE*2 ; charger adresse esclave (b7 à b1) avec b0 = 0
movwf SSPBUF ; lancer l'émission de l'adresse en mode écriture
return ; et retour
```

et son alternative :

- On place l'adresse de l'esclave comme précédemment
- On attend la fin de l'émission

I2C_adress

```
movlw SLAVE*2 ; charger adresse esclave (b7 à b1) avec b0 = 0
movwf SSPBUF ; lancer l'émission de l'adresse en mode écriture
BANKSEL SSPSTAT ; passer en banque 1
```

I2C_adress2

```
clrwdt ; effacer watchdog
btfsc SSPSTAT,R_W ; tester si émission terminée
goto I2C_adress2 ; non, attendre
bcf STATUS,RP0 ; oui, repasser banque 0
return ; et retour
```

23.7.5 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

I2C_check

```
call I2C_idle ; attendre fin de l'opération précédente
BANKSEL SSPCON2 ; passer en banque 1
btfsc SSPCON2,ACKSTAT ; tester ACK reçu
goto error ; pas reçu, traiter erreur
.. .. . ; poursuivre ici si OK
.. .. .
```

error

```
.. .. . ; continuer ici si l'esclave n'est pas prêt.
.. .. .
```

Pour la seconde méthode, c'est strictement identique, si ce n'est que l'appel de la sous-routine I2C_idle n'est pas nécessaire (l'opération précédente est forcément terminée, puisqu'on attend qu'elle soit finie avant de sortir de la sous-routine).

Cette procédure ne lance aucune commande, elle ne nécessite donc pas d'attendre avant le « return ».

23.7.6 L'écriture d'un octet

Nous en sommes arrivé 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

```
I2C_send
    call  I2C_idle    ; attendre fin de l'opération précédente
    movlw OCTET      ; charger octet à envoyer
    movwf SSPBUF     ; lancer l'écriture de l'octet
    return           ; et retour
```

et son alternative :

- On place l'octet à envoyer dans SSPBUF
- On attend la fin de l'émission

```
I2C_send
    movlw OCTET      ; charger octet à envoyer
    movwf SSPBUF     ; lancer l'écriture
    BANKSEL SSPSTAT  ; passer en banque 1
I2C_send2
    clrwdt           ; effacer watchdog
    btfsc SSPSTAT,R_W ; tester si émission terminée
    goto  I2C_send2  ; non, attendre
    bcf   STATUS,RP0  ; oui, repasser banque 0
    return           ; et retour
```

23.7.7 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

Voici le code correspondant :

```
I2C_start
    call  I2C_idle    ; attendre fin de l'opération précédente
    BANKSEL SSPCON2   ; passer en banque 1
    bsf   SSPCON2,RSN ; lancer le repeated start-condition
    bcf   STATUS,RP0  ; repasser en banque 0
```

```
return          ; retour
```

Et son alternative, toujours sur le même principe :

- On lance le repeated start-condition
- On attend que le repeated start-condition soit terminé

```
I2C_start
    BANKSEL    SSPCON2    ; passer en banque 1
    bsf        SSPCON2,RSEN ; lancer le repeated start-condition
I2C_start2
    clrwdt      ; effacer watchdog
    btfsc      SSPCON2,RSEN ; start-condition terminé ?
    goto       I2C_start2 ; non, attendre
    bcf        STATUS,RP0  ; oui, repasser en banque 0
    return      ; et retour
```

Maintenant, vous allez me dire « mais qu'est-ce qui différencie un repeated start-condition d'un start-condition ordinaire ? »

En fait, vous devez vous souvenir 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 :

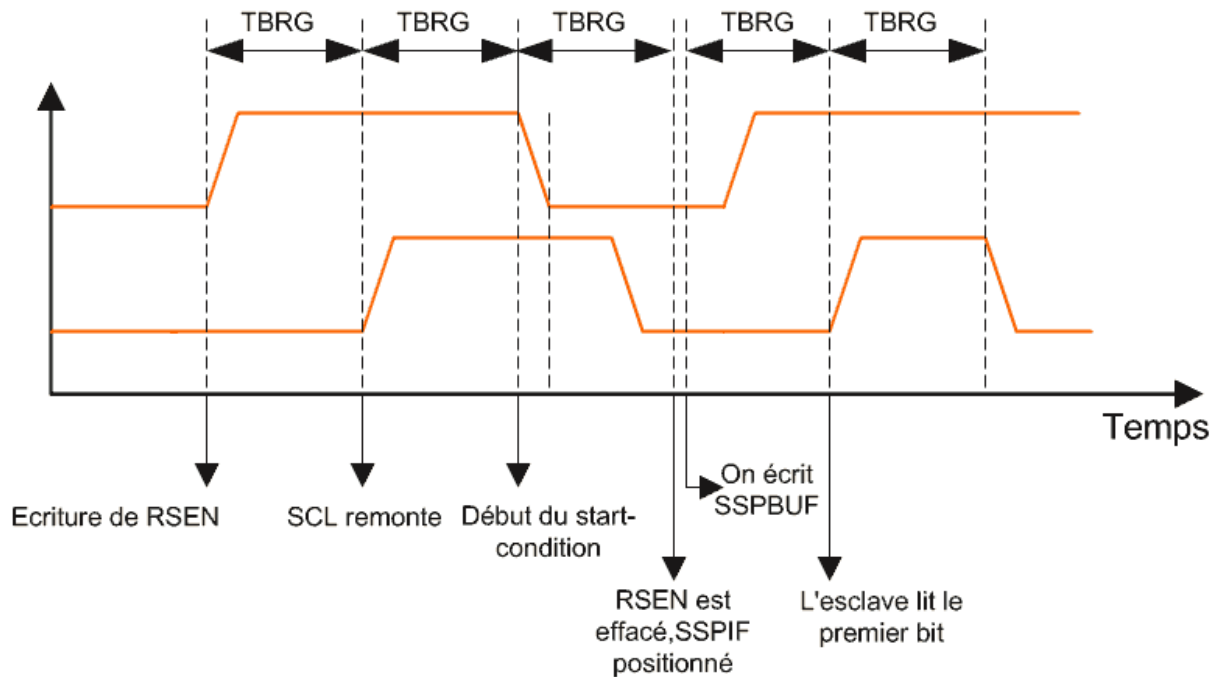
- Dès le positionnement de SREN, la ligne SDA est placée à l'état haut
- Après un temps égal à T_{BRG} , la ligne SCL est amenée également à l'état haut
- Après une nouvelle durée égale à T_{BRG} , on repasse SDA à 0, ce qui équivaut alors à un start-condition
- T_{BRG} plus tard, le bit SREN est effacé, 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 « start-condition » simple. Elles permettent de ramener les 2 lignes à l'état haut sans générer de « stop-condition ». Elles nécessitent 2 temps T_{BRG} supplémentaires.

Les 2 dernières lignes correspondent évidemment au cycle normal d'un « start-condition ».

Et hop, un petit chronogramme (mais si, je sais que vous aimez ça) :



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.

23.7.8 L'envoi de l'adresse de l'esclave

Nous allons maintenant devoir envoyer encore une fois l'adresse de l'esclave. Nous devons 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

```
I2C_adress
call I2C_idle ; attendre fin de l'opération précédente
movlw (SLAVE*2) |1 ; charger adresse esclave (b7 à b1) avec b0 = 1
; on pouvait écrire également (SLAVE*2)+1
movwf SSPBUF ; lancer l'émission de l'adresse en mode écriture
return ; et retour
```

et son alternative :

- On place l'adresse de l'esclave comme précédemment
- On attend la fin de l'émission

```
I2C_adress
movlw (SLAVE*2) |1 ; charger adresse esclave (b7 à b1) avec b0 = 1
; SLAVE*2 = adresse décalée, |1 = « OR 1 »
```

```

movwf SSPBUF      ; lancer l'émission de l'adresse en mode écriture
BANKSEL  SSPSTAT  ; passer en banque 1
I2C_adress2
clrwdt            ; effacer watchdog
btfsc SSPSTAT,R_W ; tester si émission terminée
goto I2C_adress2 ; non, attendre
bcf  STATUS,RP0   ; oui, repasser banque 0
return           ; et retour

```

23.7.9 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

```

I2C_read
call I2C_idle ; attendre fin de l'opération précédente
BANKSEL  SSPCON2 ; passer banque 1
bsf  SSPCON2,RCEN ; lancer la réception
bcf  STATUS,RP0   ; repasser banque 0
return           ; et retour

```

et l'alternative :

- On lance la lecture
- On attend la fin de l'opération pour sortir

```

I2C_read
BANKSEL  SSPCON2 ; passer banque 1
bsf  SSPCON2,RCEN ; lancer la réception
I2C_read2
btfsc SSPCON2,RCEN ; réception terminée ?
goto I2C_read2 ; non, attendre
bcf  STATUS,RP0   ; repasser banque 0
return           ; et sortir

```

23.7.10 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

```

I2C_NOACK
call I2C_idle ; attendre fin de l'opération précédente
BANKSEL  SSPCON2 ; passer banque 1
bsf  SSPCON2,ACKDT ; le bit qui sera envoyé vaudra « 1 »
bsf  SSPCON2,ACKEN ; lancer l'acknowledge (= ACKDT = 1 = NOACK)
bcf  STATUS,RP0   ; repasser banque 0
return           ; et retour

```

et l'alternative :

- On envoie NOACK
- On attend que l'envoi soit terminé

```

I2C_NOACK
    BANKSEL    SSPCON2    ; passer banque 1
    bsf        SSPCON2,ACKDT ; le bit qui sera envoyé vaudra « 1 »
    bsf        SSPCON2,ACKEN ; lancer l'acknowledge (= ACKDT = 1 = NOACK)

I2C_NOACK2
    btfsc      SSPCON2,ACKEN ; tester si NOACK terminé
    goto       I2C_NOACK2    ; non, attendre
    bcf        STATUS,RP0    ; repasser banque 0
    return
    ; et retour

```

23.7.11 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

Voici le code correspondant :

```

I2C_stop
    call       I2C_idle    ; attendre fin de l'opération précédente
    BANKSEL    SSPCON2    ; passer en banque 1
    bsf        SSPCON2,PEN ; lancer le stop-condition
    bcf        STATUS,RP0  ; repasser en banque 0
    return
    ; retour

```

Et son alternative, comme expliqué plus haut :

- On lance le stop-condition
- On attend que le repeated start-condition soit terminé

```

I2C_stop
    BANKSEL    SSPCON2    ; passer en banque 1
    bsf        SSPCON2,PEN ; lancer le stop-condition

I2C_start2
    clrwdt     ; effacer watchdog
    btfsc      SSPCON2,PEN ; stop-condition terminé ?
    goto       I2C_start2 ; non, attendre
    bcf        STATUS,RP0  ; oui, repasser en banque 0
    return
    ; et retour

```

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.

Remarquez également que le flag SSPIF est positionné pour chaque opération terminée. Vous pouvez donc remplacer le test de fin d'exécution par un simple test sur ce flag :

```

Wait
    bcf    STATUS,RP0        ; passer banque 0
    bcf    PIR1,SSPIF        ; reset du flag
Wait2
    btfss  PIR1,SSPIF        ; flag positionné ?
    goto   Wait2             ; non, attendre
    return                    ;return

```

23.8 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.

Vous aurez alors dans votre routine d'interruption SSP, un sélecteur dans le style (cmpt est un compteur) :

```

    incf   cmpt,f            ; incrémenter compteur de position
    movlw  HIGH tablesaut    ; poids fort adresse de la table de saut
    movwf  PCLATH            ; dans PCLATH
    movf   cmpt,w            ; charger compteur de position
    addlw  LOWtablesaut      ; ajouter poids faible adresse table de saut
    btfsc  STATUS,C          ; tester si on débordé
    incf   PCLATH,f          ; oui, incrémenter PCLATH
tablesaut
    movwf  PCL               ; provoquer un saut dans la table qui suit :
    goto   I2C_start         ; envoyer start-condition (cmpt valait 0 en entrant
                                ; dans la routine, et vaut 1 maintenant)
    goto   I2C_sendaddress   ; envoyer adresse esclave (cmpt = 2)
    goto   I2C_testAck       ; tester acknowledge
    goto   I2C_sendByte     ; envoyer octet
    goto   I2C_testAck       ; tester acknowledge
    goto   I2C_stop          ; envoyer stop-condition

```

Vous voyez que suivant l'état de cmpt, vous allez sauter à des routines différentes en fonction de l'avancée de votre séquence. J'ai illustré une séquence pour l'écriture. Celle-ci commencera en initialisant cmpt à 0. Il sera incrémenté à chaque appel. Ainsi, la première adresse de saut sera « etiquettesaut + 1 », donc le premier « goto ». Vous écrirez bien entendu également, si vous en avez besoin, une seconde suite de « goto » correspondant à la lecture.

Si la première adresse du « goto » concernant la lecture vaut, par exemple, 22, vous initialiserez cmpt à 21 pour une lecture.

Ensuite, vous remarquez ici une astuce de programmation. Je vous avais dit que lorsque vous créez un tableau de saut, vous ne pouviez pas déborder sur 2 « pages » de 256 octets. Votre tableau était donc de fait limité.

L'astuce utilisée ici permet de dépasser cette limite. En effet, au lieu d'ajouter « w » à PCL directement, on opère l'addition dans le registre « W ». Si on détecte que le résultat débordera sur l'autre page, on incrémente tout simplement PCLATH. Ne reste plus qu'à placer le résultat dans PCL.

De cette façon, non seulement vous pouvez placer votre tableau à n'importe quel endroit, mais, en plus, vous pouvez utiliser un compteur de plus de 8 bits, ce qui vous permet de créer des tableaux de sauts de plus de 256 éléments.

La même technique est évidemment autorisée pour les tableaux de « retlw ». Vous voyez que vous commencez à devenir des « pros ».

23.8 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.

23.8.1 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 :

- Durant le start-condition
- Durant l'envoi de l'adresse de l'esclave
- Durant l'émission d'un NOACK
- Durant l'émission d'un octet
- Durant un repeated start-condition.

23.8.2 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 recommence 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. En effet, si « P » est positionné, alors « S » est effacé. Donc, il est inutile de tester « P », c'est de la logique combinatoire.

J'ai interrogé Microchip à ce sujet, la réponse a été que j'avais effectivement raison, mais qu'il était plus sûr de tester comme indiqué. Je soupçonne donc fortement que cette réponse qui ne veut rien dire ne cache un bug de fonctionnement concernant le fonctionnement de ces bits. Je vous encourage donc à procéder comme recommandé.

- Si le bit « P » est positionné, alors le bus est libre
- Si « P » et « S » sont effacés, alors le bus est libre.

Ceci nous donne :

```
I2C_free
    BANKSEL SSPSTAT      ; passer en banque 1
    btfsc SSPSTAT,P      ; tester P positionné
    goto busfree         ; oui, traiter bus libre
    btfsc SSPSTAT,S      ; Tester si P et S libres
    goto I2C_free        ; S positionné, on recommence les tests
busfree
    suite...             ; suite du traitement, le bus est libre
```

23.8.3 La détection de la perte de contrôle

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. Il vous suffira alors, par exemple, si une interruption de ce type est générée, de remettre votre variable « cmpt » à 0, afin de reprendre les opérations au début.

23.9 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.

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 a lieu.

Donc, puisque la transmission peut avoir lieu à n'importe quel moment, la meilleure méthode d'exploitation sera d'utiliser les interruptions. Je vous donnerai donc des exemples de routines dans le chapitre correspondant.

Néanmoins, 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'**

23.9.1 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 ».

```
I2C_init
    BANKSEL    SSPADD        ; passer en banque 1
    movlw     ADRESSE*2      ; charger valeur adresse (bits 7 à 1)
    movwf     SSPADD        ; dans registre d'adresse
    clrf      SSPSTAT        ; slew rate ON, compatible I2C
    movlw     B'10000000'    ; appel général en service (facultatif)
    movwf     SSPCON2        ; dans registre SSPCON2
    bcf       STATUS,RP0     ; repasser banque 0
    movlw     B'00110110'    ; MSSP en service, mode I2C 7 esclave 7 bits, SCL OK
    movwf     SSPCON         ; dans registre de contrôle
```

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 :

23.9.2 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 et sait maintenant que suit l'adresse de l'esclave concerné.

23.9.3 La réception de l'adresse en mode écriture

Deux cas peuvent maintenant se produire. Soit l'adresse qui vient d'être émise par le maître est celle de votre PIC, 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 (je vais en parler), 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. Mais vous allez me dire que cela ne sert à rien, puisque l'adresse lue doit forcément être l'adresse de votre esclave, que vous connaissez.

En réalité, pas tout à fait. 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.

```
I2C_adress
    movf    SSPBUF,w      ; charger adresse reçue (efface BF)
    btfsc   STATUS,Z      ; tester si 0 (adresse générale)
    goto    adressGen     ; oui, traiter adresse générale (toujours en écriture)
    suite..               ; non, traiter une commande via l'adresse de SSPADD
I2C_adressGen
    Suite..               ; ici, il s'agit de l'adresse générale.
```

23.9.4 La génération du « ACK »

Comme je viens de vous le dire, elle est automatique. On distingue 2 cas :

- 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.
- Soit le bit BF ou le bit SSPOV ou les deux étaient 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.

23.9.5 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.

23.9.6 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 POSITIONNÉ**
- 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.

23.9.6 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.

```
I2C_Send
    movlw DATA          ; charger la donnée à envoyer
    movwf SSPBUF          ; dans le buffer
    bsf    SSPCON,CKP     ; met fin à la pause, permet le transfert
```

ou, en alternative, si vous désirez tester toutes les conditions d'erreur :

```
I2C_Send
    bsf    STATUS,RP0     ; passer en banque 1
    btfsc  SSPSTAT,BF     ; envoi précédent envoyé ?
    goto   $-1            ; non, attendre
    bcf    STATUS,RP0     ; repasser banque 0
I2C_send2
    bcf    SSPCON,WCOL     ; effacer bit de collision
    movlw DATA           ; charger donnée à envoyer
```

```

movwf SSPBUF      ; dans le buffer
btfsc SSPCON, WCOL ; collision ?
goto I2C_send2    ; oui, on recommence l'écriture
bsf SSPCON, CKP    ; met fin à la pause, permet le transfert

```

J'ai utilisé volontairement l'expression « goto \$-1 », car vous la rencontrerez parfois dans certains programmes. L'explication est la suivante :

« \$ » est une directive qui représente l'adresse courante. Donc, « \$-1 » l'adresse précédente. Comme dans les PICs 16Fxxx, chaque instruction nécessite un mot de programme, « goto \$-1 » veut simplement dire « sauter à la ligne précédente », tout comme « goto \$-3 » signifiera « sauter 3 lignes plus haut ». De même « goto \$+4 » indiquera qu'il faut sauter 4 lignes plus bas.

Si vous décidez d'utiliser cette directive, limitez-vous à 2 ou 3 lignes au maximum. Evitez les « goto -35 » qui rendraient votre programme parfaitement illisible. C'est pourquoi j'ai attendu longtemps avant d'introduire cette notion, afin de vous habituer à utiliser des étiquettes.

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. Comme je vais vous l'indiquer plus bas, 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.

23.9.7 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 :

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.

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.

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.

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.

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 resetter 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.

La routine d'interruption

Nous pouvons maintenant en venir à la façon de traiter les interruptions. Nous allons imaginer que nous complétons la sous-routine d'interruption int_ssp. Nous allons considérer que la zone des variables contient les éléments suivants :

- La zone de stockage des octets reçus démarre en bufin
- La zone de stockage des octets à émettre démarre en bufout
- ptrin contient un pointeur sur le prochain emplacement libre de bufin
- ptrout contient un pointeur sur le prochain emplacement libre de bufout
- flagout est un flag qui est positionné lorsque tous les octets ont été envoyés
- flaggen est un flag qui indique s'il s'agit d'une commande générale

Le traitement des flags est du ressort du programme principal.

Je ne gère pas les conditions d'erreur (ex : débordement des buffers) afin de ne pas alourdir les routines, l'important est de comprendre la façon de procéder.

```

;*****
;                                INTERRUPTION SSP                                *
;*****
;-----
; Gestion de l'interruption en mode esclave.
; cas 1 : réception de l'adresse en mode écriture
; cas 2 : réception d'une donnée
; cas 3 : réception de l'adresse en mode lecture
; cas 4 : envoi d'une donnée en mode lecture
; cas 5 : réception du NOACK de fin de lecture
; erreur : tout autre cas est incorrect, on provoque le reset du PIC par
;          débordement du watchdog (à vous de traiter de façon appropriée)
;-----
intssp
        ; conserver les bits utiles
        ; -----
        BANKSEL    SSPSTAT        ; passer en banque 1
        movf    SSPSTAT,w        ; charger valeur SSPSTAT
        andlw   B'00101101'      ; garder D_A, S, R_W, et BF
        bcf     STATUS,RP0       ; repasser en banque 0
        movwf   ssptemp          ; sauvegarder dans variable temporaire

        ; cas 1 : réception adresse écriture
        ; -----
        xorlw   B'00001001'      ; buffer plein, contient une adresse
        btfss   STATUS,Z        ; condition réalisée ?
        goto    intssp2         ; non, examiner cas 2

        movlw   bufin            ; charger adresse du buffer de réception
        movwf   ptrin            ; le prochain octet sera le premier du buffer in
        bcf     flaggen          ; par défaut, pas une commande générale
        movf    SSPBUF,w        ; efface BF, teste l'adresse
        btfsc   STATUS,Z        ; adresse générale reçue ?
        bsf     flaggen          ; oui, positionner le flag
        return                    ; fin du traitement

        ; cas 2 : réception d'une donnée
        ; -----
intssp2
        movf    ssptemp,w        ; charger bits utiles de SSPSTAT
        xorlw   B'00101001'      ; buffer plein, contient une donnée
        andlw   B'11110111'      ; éliminer le bit S (stop-condition déjà reçu)
        btfss   STATUS,Z        ; condition réalisée ?
        goto    intssp3         ; non, examiner cas 3

        movf    ptrin,w          ; charger pointeur d'entrée
        movwf   FSR              ; dans pointeur d'adresse
        movf    SSPBUF,w        ; charger donnée reçue
        movwf   INDF            ; la placer dans le buffer d'entrée
        incf    ptrin,f         ; incrémenter le pointeur d'entrée
        return                    ; fin du traitement

        ; cas 3 : émission de la première donnée
        ; -----
intssp3
        movf    ssptemp,w        ; charger bits utiles de SSPSTAT

```

```

    xorlw B'00001100'      ; demande d'envoi, on vient de recevoir l'adresse
    btfss STATUS,Z         ; condition réalisée ?
    goto  intssp4          ; non, examiner cas 4
    movlw bufout+1         ; charger adresse du buffer d'émission + 1
    movwf ptrout           ; le prochain octet sera le second du buffer out
    movf  bufout,w         ; charger le premier octet du buffer d'émission
    movwf SSPBUF           ; dans le buffer I2C
    bsf   SSPCON,CKP       ; met fin à la pause, permet le transfert
    return                ; fin du traitement

; cas 4 : émission d'une donnée quelconque
; -----
intssp4
    movf  ssptemp,w        ; charger bits utiles de SSPSTAT
    xorlw B'00101100'     ; demande d'envoi qui suit un autre envoi
    btfss STATUS,Z        ; condition réalisée ?
    goto  intssp5         ; non, examiner cas 5

    movf  ptrout,w         ; charger pointeur buffer de sortie
    movwf FSR             ; dans pointeur d'adresse
    movf  INDF,w          ; charger octet à envoyer
    movwf SSPBUF          ; le mettre dans le buffer de sortie
    bsf   SSPCON,CKP       ; libère l'horloge, permet le transfert
    incf  ptrout,f         ; incrémenter pointeur de sortie
    return                ; fin du traitement

; cas 5 : réception du NOACK
; -----
intssp5
    movf  ssptemp,w        ; charger bits utiles de SSPSTAT
    xorlw B'00101000'     ; NOCAK reçu
    btfss STATUS,Z        ; condition réalisée ?
    goto  $               ; non, reset PIC par débordement watchdog

    bsf   flagout          ; signaler fin de lecture
    return                ; et retour

```

Vous voyez que tout est logique. Vous allez peut-être penser que cette routine de sélection est un peu longue. Mais, de toute façon, je viens de l'écrire pour vous, pas vrai? Je vous conseille un excellent exercice si vous comptez utiliser l'I²C en mode esclave : essayez de réécrire cette routine vous-même, vous saurez vite si vous avez tout compris ou pas.

23.10 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 Address) 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 resetté, 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** : on est alors prêt pour une prochaine réception, le bit UA est resetté 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 ACK
- 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 : 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 resetté, 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 : on est alors prêt pour une prochaine réception, le bit UA est resetté 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 : on ne met à jour SSPADD que si UA est positionné. Sinon, on poursuit le traitement ordinaire de notre interruption.

23.11 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.

Vous voici devenu des spécialistes de l'I²C, à moins que vous ne ressentiez brutalement un important mal de tête. Rassurez-vous, en prenant une pause et en relisant quelques fois, vous allez voir que c'est plus simple à comprendre qu'à expliquer (heureusement pour vous, malheureusement pour moi).

Vous constatez par vous-même qu'il ne m'est guère possible de donner un exemple concret pour chaque mode, c'est pourquoi je vous ai donné dans la théorie des exemples de traitement des événements I²C. Je ne réaliserai donc qu'un exercice, que j'ai choisi volontairement comme un cas des plus courants.

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 I2C_free

- Si vous avez absolument besoin de libérer le PIC le plus rapidement possible, vous devrez utiliser les interruptions.

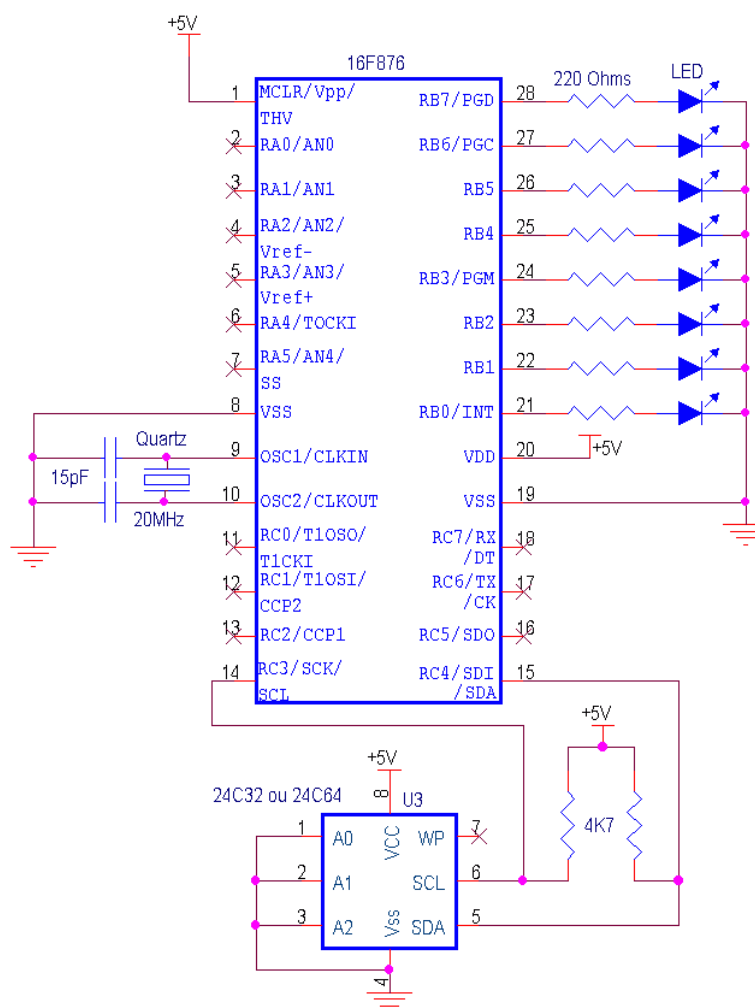
Pour le mode esclave :

- **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.

23.12 Exercice pratique : pilotage d'une 24c64

Enfin, nous arrivons à la pratique. Pour réaliser cet exercice, nous allons ajouter une eeprom 24C32 ou 24C64 à notre PIC. Nous aurons également besoin de 8 LEDs, afin de visualiser ce que contient l'eeprom.

Voici le montage :



Notez sur ce schéma la présence des **2 résistances de rappel au +5V** indispensables sur les lignes SCL et SDA.

Nous allons réaliser un programme qui écrit 40 valeurs dans l'EEPROM. Ces valeurs vont ensuite être lues pour provoquer les allumages des LEDs. En somme, le programme va fonctionner comme notre programme « lum », mais les données seront écrites et lues dans notre EEPROM, ce qui est bien entendu le but de cet exercice.

Commencez par effectuer un copier/coller de votre fichier « m16F876.asm » et renommez cette copie « lum_eep.asm ». Construisez un nouveau projet sur base de ce fichier.

Comme toujours, on commence par l'en-tête et la configuration. J'ai choisi de mettre le watchdog en service :

```
;*****
; Exercice sur les accès I2C concrétisés par les échanges avec une 24C64. *
; Le programme écrit 40 octets dans l'EEPROM, puis les lit en boucle et *
; envoie les octets lus à intervalle de 0,5 seconde sur le PORTB (LEDs) *
; *
;*****
;
; NOM: lum_eep *
; Date: 06/07/2002 *
; Version: 1.0 *
; Circuit: Platine d'expérimentation *
; Auteur: Bigonoff *
; *
;*****
;
; Fichier requis: P16F876.inc *
; *
;*****
;
; Notes: L'EEPROM est connectée sur le bus I2C (SCL + SDA) *
; A2, A1, et A0 de l'EEPROM sont connectées à la masse *
; 8 LEDs sont connectées sur le PORTB *
; LE quartz travaille à 20MHz. *
; *
;*****
```

```
LIST p=16F876 ; Définition de processeur
#include <p16F876.inc> ; fichier include
```

```
_CONFIG _CP_OFF & _DEBUG_OFF & _WRT_ENABLE_OFF & _CPD_OFF & _LVP_OFF &
_BODEN_OFF & _PWRTE_ON & _WDT_ON & _HS_OSC
```

```
;_CP_OFF Pas de protection
;_DEBUG_OFF RB6 et RB7 en utilisation normale
;_WRT_ENABLE_OFF Le programme ne peut pas écrire dans la flash
;_CPD_OFF Mémoire EEPROM déprotégée
;_LVP_OFF RB3 en utilisation normale
;_BODEN_OFF Reset tension hors service
;_PWRTE_ON Démarrage temporisé
;_WDT_ON Watchdog en service
;_HS_OSC Oscillateur haute vitesse (4Mhz<F<20Mhz)
```

```
;*****
```

```
;
; ***** ASSIGNATIONS SYSTEME *****
;
```

```
; REGISTRE OPTION_REG (configuration)
; -----
OPTIONVAL EQU B'10000000' ; RBP b7 : 1= Résistance rappel +5V hors service
```

Ensuite, nos assignations. Nous allons avoir besoin de la valeur de recharge du timer2, calculée comme dans notre programme « spimast1.asm ». On prendra une valeur de recharge de PR2 de 249, ce qui, associé à un nombre de passages de 125 et une combinaison pré et postdiviseurs donnant une division totale de 80, nous donnera un temps total de 0.5S.

Comme nous avons câblé notre eeprom avec ses lignes A2,A1, et A0 à la masse, son adresse : 1010 A2 A1 A0 R/W nous donnera en pratique : B'10100000' (0xA0). Vue sous cette forme, l'adresse est déjà prête à placer dans SSPADD, donc nous n'aurons pas besoin de la décaler vers la gauche.

```
; *****
; ***** ASSIGNATIONS PROGRAMME *****
; *****

ADRESS EQU B'10100000' ; adresse eeprom = 1010 A2 A1 A0 R/W(0)
; l'adresse est déjà décalée (0xA0)
PR2VAL EQU D'249' ; Valeur de comparaison timer 2
CMPTVAL EQU D'125' ; 125 passages dans la routine d'interruption
; durée = Tcy*(PR2+1)*prédiv*postdiv*cmptval
; = 0,2µs * 250 * 16 * 5 * 125 = 0.5s.
```

Nous allons devoir écrire 40 valeurs dans notre mémoire RAM. Afin de réduire un peu notre fichier source (mais pas l'exécutable), nous allons créer une petite macro, qui place l'octet passé en paramètre dans le buffer, à la position précisée comme second paramètre.

```
; *****
; ***** MACRO *****
; *****

WBUF macro octet,offset ; place l'octet "octet" dans buffer+offset
    movlw octet ; charger octet
    movwf buffer+offset ; placer dans le buffer
endm
```

Maintenant, notre zone de variables, qui contiendra notre buffer de 32 octets, une variable contenant le nombre d'octets à envoyer, une variable sur 16 bits qui contiendra l'adresse à envoyer dans le pointeur d'adresse de l'eeprom (l'adresse de lecture ou d'écriture), et, enfin, notre flag et notre compteur pour l'interruption du timer2.

```
; *****
; ***** VARIABLES BANQUE 0 *****
; *****

; Zone de 80 bytes
; -----

CBLOCK 0x20 ; Début de la zone (0x20 à 0x6F)

buffer : 0x20 ; 32 octets de buffer
```

```

buflen : 1          ; longueur utilisée du buffer
cmpt : 1            ; compteur de passages d'interruption
flags : 1          ; 8 flags d'usage général
                ; b0 : 0.5s s'est écoulée
eepa : 2            ; valeur pour le pointeur d'adresse eeprom
ENDC              ; Fin de la zone

```

La zone de variables communes ne contient que les variables de sauvegarde nécessaires :

```

;*****
;                               VARIABLES ZONE COMMUNE                               *
;*****

; Zone de 16 bytes
; -----

CBLOCK 0x70          ; Début de la zone (0x70 à 0x7F)
w_temp : 1          ; Sauvegarde registre W
status_temp : 1     ; sauvegarde registre STATUS
ENDC

```

La routine d'interruption, tout droit tirée de notre programme « lum », se contente de positionner le flag au bout de 125 passages correspondants à 0.5 s.

```

;*****
;                               DEMARRAGE SUR RESET                               *
;*****

org 0x000          ; Adresse de départ après reset
goto init         ; Initialiser

; //////////////////////////////////////

;                               I N T E R R U P T I O N S

; //////////////////////////////////////

;*****
;                               ROUTINE INTERRUPTION                               *
;*****
;*****
;                               ROUTINE INTERRUPTION                               *
;*****
;-----
; La routine d'interruption timer 2 est appelée toutes les
; (0,2µs * 80 * 250) = 4ms.
; au bout de 125 passages, une demi-seconde s'est écoulée, on positionne
; le flag
;-----

                ;sauvegarder registres
;-----

org 0x004          ; adresse d'interruption
movwf w_temp       ; sauver registre W
swapf STATUS,w     ; swap status avec résultat dans w
movwf status_temp  ; sauver status swappé
bcf STATUS,RP0     ; passer banque0
bcf STATUS,RP1

```

```

                ; interruption timer 2
                ; -----
bcf    PIR1,TMR2IF    ; effacer le flag d'interruption
bsf    flags,1        ; 8 ms écoulées
decfsz cmpt,f        ; décrémenter compteur de passages
goto   restorereg     ; pas 0, fin de l'interruption
movlw  CMPTVAL        ; valeur de recharge du compteur
movwf  cmpt           ; recharger compteur
bsf    flags,0        ; positionner flag

                ; restaurer registres
                ; -----
restorereg
    swapf status_temp,w    ; swap ancien status, résultat dans w
    movwf STATUS          ; restaurer status
    swapf w_temp,f        ; Inversion L et H de l'ancien W
                        ; sans modifier Z
    swapf w_temp,w        ; Réinversion de L et H dans W
                        ; W restauré sans modifier status
    retfie                ; return from interrupt

```

On en arrive aux initialisations. En premier lieu, les PORTs. Comme au moment de la mise sous tension, les ports sont tous en entrée, on ne forcera que les pins qui devront être configurées en sortie. Dans notre cas, le PORTB, qui, pilotera les LEDs.

On en profite pour couper les résistances de rappel au +5V (pullup) du PORTB, qui seront ici inutiles, via le registre d'option.

```

; //////////////////////////////////////
;                                     P R O G R A M M E
; //////////////////////////////////////

;*****
;                               INITIALISATIONS
;*****
init

    ; initialisation PORTS
    ; -----
    BANKSEL    PORTB        ; passer banque 0
    clrf    PORTB          ; sorties PORTB à 0
    bsf    STATUS,RP0      ; passer en banque1
    clrf    TRISB          ; PORTB en sortie, les autres en entrée
    movlw  OPTIONVAL       ; charger masque
    movwf  OPTION_REG      ; initialiser registre option

```

Vient maintenant notre module I²C. Nous allons indiquer que nous mettons le « slew-rate control » en service et que nous travaillons avec des niveaux compatibles I²C. Ceci nous amène à n'écrire que des « 0 » dans SSPSTAT.

La fréquence est déterminée par la valeur placée dans SSPSTAT. Nous avons déjà calculé cette valeur pour une fréquence de quartz de 20MHz et un débit de 400KBauds. Cette valeur vaut 12.

Il ne reste plus qu'à mettre le module en service et à choisir le mode I²C maître (tout ceci via SSPCON)

```

; initialiser I2C
; -----
clrf  SSPSTAT      ; slew rate control en service, mode I2C
movlw D'12'        ; valeur de recharge du BRG (400 Kbauds)
movwf SSPADD       ; dans registre de recharge
bcf   STATUS,RP0   ; passer banque 0
movlw B'00101000'  ; module MSSP en service en mode I2C master
movwf SSPCON       ; dans registre de contrôle

```

On initialise ensuite nos variables, ce qui se limite ici à resetter le flag et à initialiser le compteur de passage pour la première durée de 0.5 s.

```

; initialiser variables
; -----
clrf  flags        ; reset flags
movlw CMPTVAL      ; pour 125 passages
movwf cmpt         ; dans compteur de passages

```

Reste à initialiser notre timer2 (avec pré et postdiviseurs) et de mettre l'interruption correspondante en service. C'est du déjà-vu.

```

; initialiser timer 2
; -----
movlw PR2VAL       ; charger valeur de comparaison
BANKSEL PR2        ; passer banque 1
movwf PR2          ; initialiser comparateur
movlw B'00100110'  ; timer2 on, prédiv = 16, post = 5
bcf   STATUS,RP0   ; passer banque 0
movwf T2CON        ; lancer timer 2

; lancer interruption timer 2
; -----
bsf   STATUS,RP0   ; passer banque 1
bsf   PIE1,TMR2IE  ; interruption timer 2 en service
bcf   STATUS,RP0   ; repasser banque 0
bsf   INTCON,PEIE  ; interruptions périphériques en service
bsf   INTCON,GIE   ; lancer les interruptions
goto  start        ; programme principal

```

Notre sous-routine d'interruption se contente de tester le positionnement du flag, positionnement effectué par la routine d'interruption. Bien entendu, il ne faudra pas oublier de resetter ce flag avant de sortir :

```

;*****
;
; Attendre 0.5 seconde
;*****
;-----
; attendre que 0.5s se soit écoulée depuis le précédent passage dans
; cette routine
;-----
wait
  clrwdt          ; effacer watchdog
  btfss flags,0   ; flag positionné?
  goto wait       ; non, attendre flag
  bcf  flags,0    ; reset du flag
  return          ; et retour

```

Maintenant, nous arrivons à notre programme principal, qui commence par remplir le buffer avec des données à mémoriser en eeprom.

Bien entendu, c'est idiot de procéder au remplissage d'une eeprom de cette façon, puisque nous allons placer en eeprom des données qui se trouvent déjà dans notre PIC. Mais il s'agit d'un exercice. Comme j'ai limité les composants au maximum, je n'ai pas d'échange avec l'extérieur, et donc, je n'ai pas d'autre moyen de vous proposer à la fois lecture et écriture.

Dans le chapitre sur les communications asynchrones, je modifierai ce programme pour le rendre utile. En attendant, revenons-en à nos moutons (pardon, à nos eeproms).

Nous allons utiliser notre macro pour écrire 32 valeurs dans le buffer :

```

;*****
;
;          PROGRAMME PRINCIPAL
;*****
start

        ; remplir le buffer d'émission
        ; -----
WBUF    0x01,0x00      ; placer 1'octet dans buffer
WBUF    0x02,0x01      ; placer 1'octet dans buffer + 1
WBUF    0x04,0x02      ; placer 1'octet dans buffer + 2
WBUF    0x08,0x03      ; placer 1'octet dans buffer + 3
WBUF    0x10,0x04      ; placer 1'octet dans buffer + 4
WBUF    0x20,0x05      ; placer 1'octet dans buffer + 5
WBUF    0x40,0x06      ; placer 1'octet dans buffer + 6
WBUF    0x80,0x07      ; placer 1'octet dans buffer + 7
WBUF    0x40,0x08      ; placer 1'octet dans buffer + 8
WBUF    0x20,0x09      ; placer 1'octet dans buffer + 9
WBUF    0x10,0x0A      ; placer 1'octet dans buffer + 10
WBUF    0x08,0x0B      ; placer 1'octet dans buffer + 11
WBUF    0x04,0x0C      ; placer 1'octet dans buffer + 12
WBUF    0x02,0x0D      ; placer 1'octet dans buffer + 13
WBUF    0x01,0x0E      ; placer 1'octet dans buffer + 14
WBUF    0x00,0x0F      ; placer 1'octet dans buffer + 15
WBUF    0x01,0x10      ; placer 1'octet dans buffer + 16
WBUF    0x03,0x11      ; placer 1'octet dans buffer + 17
WBUF    0x07,0x12      ; placer 1'octet dans buffer + 18
WBUF    0x0F,0x13      ; placer 1'octet dans buffer + 19
WBUF    0x1F,0x14      ; placer 1'octet dans buffer + 20
WBUF    0x3F,0x15      ; placer 1'octet dans buffer + 21
WBUF    0x7F,0x16      ; placer 1'octet dans buffer + 22
WBUF    0xFF,0x17      ; placer 1'octet dans buffer + 23
WBUF    0xFE,0x18      ; placer 1'octet dans buffer + 24
WBUF    0xFC,0x19      ; placer 1'octet dans buffer + 25
WBUF    0xF8,0x1A      ; placer 1'octet dans buffer + 26
WBUF    0xF0,0x1B      ; placer 1'octet dans buffer + 27
WBUF    0xE0,0x1C      ; placer 1'octet dans buffer + 28
WBUF    0xC0,0x1D      ; placer 1'octet dans buffer + 29
WBUF    0x80,0x1E      ; placer 1'octet dans buffer + 30
WBUF    0x00,0x1F      ; placer 1'octet dans buffer + 31

```

Ensuite, nous allons envoyer ce buffer dans l'eeprom, via le bus I²C. Nous allons imaginer que la routine «eep_sendbuf» se charge d'envoyer le buffer. Nous écrivons cette routine plus loin.


```

; envoyer buffer dans eeprom
; -----
clr    eepa          ; adresse d'écriture = 00
clr    eepa+1        ; idem poids faible
movlw  0x20          ; 32 octets présents
movwf  buflen        ; placer dans le compteur d'octets
call   eep_sendbuf   ; envoyer le buffer dans l'eeprom

```

On avait convenu qu'on écrirait 40 octets. Comme on a déjà écrit 32 octets (limite autorisée en une seule écriture, souvenez-vous), il nous reste 8 octets à placer dans le buffer et à envoyer. Naturellement, on va écrire ces données dans l'eeprom à la suite des précédents, soit à partir de l'adresse 0x0020, ceci nous impose donc d'écrire la nouvelle adresse du pointeur dans notre variable eepa (qui sera utilisée dans notre sous-routine).

```

; placer 8 nouveaux octets dans le buffer
; -----
WBUF   0x55,0x00      ; placer l'octet dans buffer
WBUF   0xAA,0x01      ; placer l'octet dans buffer + 1
WBUF   0x55,0x02      ; placer l'octet dans buffer + 2
WBUF   0xAA,0x03      ; placer l'octet dans buffer + 3
WBUF   0xF0,0x04      ; placer l'octet dans buffer + 4
WBUF   0x0F,0x05      ; placer l'octet dans buffer + 5
WBUF   0xF0,0x06      ; placer l'octet dans buffer + 6
WBUF   0x0F,0x07      ; placer l'octet dans buffer + 7

; envoyer buffer dans eeprom
; -----
clr    eepa          ; adresse d'écriture = 0x0020
movlw  0x20          ; adresse poids faible
movwf  eepa+1        ; initialiser
movlw  8             ; 8 octets présents
movwf  buflen        ; placer dans le compteur d'octets
call   eep_sendbuf   ; envoyer le buffer dans l'eeprom

```

Maintenant, nos 40 octets sont écrits dans l'eeprom, reste à les lire et à les envoyer dans le PORTB.

Nous commençons donc par initialiser eepa avec l'adresse du premier octet à lire en eeprom, soit 0x00. Comme cette variable ne sera pas modifiée par les sous-routines, on ne l'initialisera qu'une fois :

```

; traitement des lectures
; -----
clr    eepa          ; adresse de lecture = 00
clr    eepa+1        ; idem poids faible

```

Maintenant, il nous reste à procéder aux lectures suivies par des envois sur PORTB. On a 40 octets à envoyer, donc :

- Pour chaque octet
- On lit l'octet en eeprom
- On l'envoie sur le PORTB
- On attend 0.5 seconde
- Si pas dernier octet, suivant.

Bien entendu, on doit répéter cette séquence à l'infini, d'où une seconde boucle extérieure :

- On positionne le pointeur d'adresse de l'eprom sur 0x0000 (premier octet)
- On initialise le compteur d'octets
- Pour chaque octet
- On lit l'octet en eeprom
- On l'envoie sur le PORTB
- On attend 0.5 seconde
- Si pas dernier octet, suivant.
- Si dernier, on reprend le tout au début

Nous avons besoin d'un compteur d'octets. Comme nous ne nous servons plus de notre buffer (on lit un octet à la fois), on récupérera notre variable « buflen ».

Nous aurons besoin de quelques sous-routines. Nous allons les nommer, nous les construirons plus loin :

eep_address : Initialise le pointeur d'adresse de l'eprom
i2c_stop : envoie le stop-condition sur le bus I²C.
eep_readc : lit l'octet courant sur l'eprom (lecture courante).

Souvenez-vous que la lecture de l'octet courant incrémente automatiquement le pointeur d'adresse interne de l'eprom. Voici ce que ça nous donne :

```
loop1
    call eep_address      ; initialiser pointeur d'adresse
    call i2c_stop         ; fin transaction
    movlw D'40'           ; 40 octets à lire
    movwf buflen          ; dans compteur d'octets
loop2
    call eep_readc        ; lire l'octet courant
    movwf PORTB           ; envoyer sur PORTB
    call wait             ; attendre 0.5s
    decfsz buflen,f       ; décrémente compteur d'octets
    goto loop2            ; pas dernier, suivant
    goto loop1            ; dernier, on recommence
```

Notre programme principal est terminé. Voyez qu'il est très simple. Bien entendu, il reste à écrire les sous-routines, mais ces sous-routines peuvent être récupérées d'un programme à l'autre.

En général, dans un programme de ce type, on aura :

- Un programme principal qui utilise des sous-routines de gestion du périphériques I2C.
- Des sous-routines de gestion du périphérique qui utilisent des sous-routines du bus I2C.
- Des sous-routines du bus I2C qui gèrent l'hardware.

En sommes, un fonctionnement par couche. En travaillant ainsi (et c'est valable pour des périphériques autres que les I²C), vous avez des avantages non négligeables :

- Vous pouvez ajouter un nouveau périphérique sans devoir réécrire les routines de gestion hardware
- Si vous changez de PIC ou de processeur, vous ne devez modifier que les routines de gestion hardware.
- Vous ne faites donc qu'une seule fois le travail pour le hardware, et une seule fois par périphérique ajouté.

Donc, notre programme principal peut être considéré comme un programme de haut niveau (il se rapproche du pseudo-code), nos routines de gestion du périphérique comme un pilote de périphérique, et nos routines de gestion du bus comme les routines de plus bas niveau (hardware ou bios).

Ceci pour vous démontrer une fois de plus qu'on peut programmer en assembleur et rester structuré.

Voyons donc comment cela se passe en pratique. Nous commencerons donc par nos routines de gestion de l'eeprom. Et tout d'abord, la routine de lecture de l'octet courant. On supposera que le pointeur d'adresse est positionné par une autre sous-routine. Les étapes nécessaires sont les suivantes :

- On envoie le start-condition
- On écrit l'adresse du circuit (attention, pas l'adresse du pointeur d'adresse) avec le bit R/W à 1 (lecture). L'adresse du circuit, pour une eeprom, est 1010 A2 A1 A0 R/W.
- On procède à la lecture de l'octet courant
- On envoie un stop-condition

Si vous avez tout compris, le start, stop, etc. seront traitées par les sous-routines du bus I²C. On décide que l'octet sera retourné au programme principal via le registre « w ».

```
; *****
; *****
;                               ROUTINES EEPROM                               *
; *****
; *****

; *****
;                               LIRE L'OCTET COURANT                               *
; *****
; *****
; -----
; Lit l'octet pointé par le pointeur eeprom et le retourne dans w
; ensuite, clôture la communication
; -----
eep_readc
    call    i2c_start          ; envoyer start-condition
    movlw   ADRESS+1          ; charger adresse eeprom en lecture
    call    i2c_write          ; envoyer adresse eeprom
    call    i2c_read           ; lire l'octet
    call    i2c_stop           ; fin du transfert
    return                    ; et retour
```

Vous voyez, c'est tout simple : en procédant de la sorte, il vous suffit du datasheet de l'eprom pour écrire votre routine.

Une autre routine nécessaire, est notre routine qui va écrire le contenu du buffer dans l'eprom. Nous aurons besoin du nombre d'octets à envoyer (passé dans buflen), de l'adresse d'écriture en eprom (passée dans eepa), et des données à écrire (contenues dans buffer). La routine devra effectuer les opérations suivantes :

- Initialiser le pointeur d'adresse (routine déjà appelée par notre programme principal)
- Pour chaque octet
- Ecrire l'octet sur le bus I2C
- Octet suivant
- Envoyer le stop-condition (on ne peut plus rien ajouter)

```
;*****
;
;                               ENVOYER LE BUFFER DANS L'EEPROM
;*****
;-----
; Envoie le buffer dans l'eprom
; buflen contient le nombre d'octets à envoyer (détruit après l'exécution)
; eepa contient l'adresse d'écriture
;-----
eep_sendbuf
    call eep_adress      ; initialiser pointeur adresse
    movlw buffer         ; charger adresse buffer
    movwf FSR           ; dans pointeur
eep_sendb1
    movf INDF,w         ; charger un octet
    call i2c_write      ; l'envoyer
    incf FSR,f          ; incrémenter pointeur buffer
    decfsz buflen,f     ; décrémenter nbre d'octets restants
    goto eep_sendb1     ; pas fini, suivant
    call i2c_stop       ; fin du transfert
    return              ; et retour
```

C'est toujours aussi simple, on ne s'occupe nullement de l'électronique. Il nous reste maintenant notre procédure d'initialisation du pointeur d'adresse. L'adresse à placer dans le pointeur se trouve dans « eepa » Les opérations sont un peu plus longues :

- On envoie un start-condition
- Label 1
- On envoie l'adresse du circuit en écriture (écriture de « 1010 A2 A1 A0 0 »)
 - On lit l'ack reçu.
 - Si NOACK, on envoie un repeated start-condition et on recommence à Label 1
 - Si ACK :
 - On écrit l'octet d'adresse de poids fort (eepa)
 - On écrit l'octet d'adresse de poids faible (eepa+1).

Une fois de plus (j'insiste), ne confondez pas l'adresse de l'eprom (1010 A2 A1 A0 R/W), avec l'adresse de lecture ou d'écriture de l'octet dans l'eprom (2 octets eepa).

```
;*****
;
;                               INITIALISER LE POINTEUR D'ADRESSE
;*****
```

```

;-----
; envoie le start-condition, puis l'adresse de l'eeprom
; ensuite, teste le ACK
; Si NOACK, on envoie le repeated start-condition, puis de nouveau l'adresse
; si toujours NOACK, on recommence
; Si ACK, on envoie les 2 octets d'adresse contenus dans eepa
;-----
eep_adress
    ; envoyer adresse circuit tant que pas ACK
    ; -----
    call i2c_start          ; envoyer start-condition
eep_adress1
    movlw ADDRESS           ; charger adresse eeprom + écriture
    call i2c_write          ; écrire adresse eeprom
    bsf STATUS,RP0         ; passer en banque 1
    btfss SSPCON2,ACKSTAT   ; tester ACK reçu
    goto eep_adressok       ; oui, poursuivre OK
    call i2c_rstart        ; non, envoyer repeated start-condition
    goto eep_adress1        ; recommencer test

    ; placer 2 octets d'adresse dans pointeur
    ; -----
eep_adressok
    bcf STATUS,RP0         ; repasser banque 0
    movf eepa,w            ; charger poids fort adresse
    call i2c_write         ; écrire poids fort
    movf eepa+1,w          ; charger poids faible adresse
    call i2c_write         ; écrire poids faible
    return                ; et retour

```

Et voilà : concernant notre eeprom, nous savons maintenant la sélectionner et initialiser le pointeur d'adresse, lire l'octet pointé, et écrire un buffer. A vous de compléter si vous avez besoin d'autres sous-routines.

Maintenant, il nous faut gérer notre hardware, c'est-à-dire piloter réellement notre bus I²C. J'ai choisi ici la méthode qui consiste à envoyer une commande, et à attendre la fin de son exécution pour sortir. Il s'agit donc simplement ici de recopier ce que nous avons vu dans la théorie.

Afin de faciliter la lecture, j'ai créé une petite macro qui attend l'effacement du bit passé en second paramètre dans le registre passé en premier paramètre.

```

;*****
;*****
;
ROUTINES I2C
;*****
;*****
;-----
; On attend que chaque commande soit terminée avant de sortir de la
; sous-routine correspondante
;-----
IWAIT macro REGISTRE,BIT    ; attendre effacement du bit du registre
    clrwdt                 ; effacer watchdog
    btfsc REGISTRE,BIT     ; bit effacé?
    goto $-2               ; non, attendre
    bcf STATUS,RP0         ; repasser en banque 0
endm                       ; fin de macro

```

Il suffit donc d'inscrire IWAIT registre ,bit pour attendre que le bit soit effacé avant de continuer. Le watchdog est géré, et on repasse en banque 0 avant de poursuivre(en effet, j'entre et je termine toujours une routine en étant en banque 0 (convention personnelle que je vous conseille). Comme ce test précédera la sortie de la sous-routine, autant l'inclure.

Remarquez le « \$ -2 » dont je vous ai déjà parlé, et qui signifie simplement « 2 lignes plus haut ».

Maintenant, on entre dans le vif de notre sujet, la gestion de L'I²C. On commence par l'envoi du start-condition :

```
;*****
;
;          ENVOYER LE START-CONDITION
;*****
i2c_start
    bsf    STATUS,RP0        ; passer en banque 1
    bsf    SSPCON2,SEN        ; lancer le start-condition
    IWAIT  SSPCON2,SEN        ; attendre fin start-condition
    return                    ; et retour
```

Plus simple me paraît impossible, pas vrai ?

Les « repeated start-condition » et « stop-condition » sont strictement similaires :

```
;*****
;
;          ENVOYER LE REPEATED START-CONDITION
;*****
i2c_rstart
    bsf    STATUS,RP0        ; passer en banque 1
    bsf    SSPCON2,RSEN      ; lancer le repeated start-condition
    IWAIT  SSPCON2,RSEN      ; attendre fin repeated start-condition
    return                    ; et retour

;*****
;
;          ENVOYER LE STOP-CONDITION
;*****
i2c_stop
    bsf    STATUS,RP0        ; passer en banque 1
    bsf    SSPCON2,PEN       ; lancer le stop-condition
    IWAIT  SSPCON2,PEN       ; attendre fin stop-condition
    return                    ; et retour
```

L'envoi d'un « ACK » ou d'un « NOACK » ne diffèrent que par la valeur placée dans ACKDT.

```
;*****
;
;          ENVOYER LE ACK
;*****
i2c_ack
    bsf    STATUS,RP0        ; passer en banque 1
    bcf    SSPCON2,ACKDT     ; le bit qui sera envoyé vaudra " 0 "
    bsf    SSPCON2,ACKEN     ; lancer l'acknowledge (= ACKDT = 0 = ACK)
    IWAIT  SSPCON2,ACKEN     ; attendre fin ACK
    return                    ; et retour

;*****
;
;          ENVOYER LE NOACK
;*****
```

```
;*****
i2c_noack
    bsf    STATUS,RP0        ; passer en banque 1
    bsf    SSPCON2,ACKDT     ; le bit qui sera envoyé vaudra " 1 "
    bsf    SSPCON2,ACKEN     ; lancer l'acknowledge (= ACKDT = 1 = NOACK)
    IWAIT  SSPCON2,ACKEN     ; attendre fin NOACK
    return                    ; et retour
```

Vous pouviez combiner les 2 sous-routines pour gagner de la place, avec 2 points d'entrée et un « goto » (optimisatoin en taille, mais pas en temps d'exécution), mais le but ici était de rester explicite.

```
i2c_noack
    bsf    STATUS,RP0        ; passer en banque 1
    bsf    SSPCON2,ACKDT     ; le bit qui sera envoyé vaudra " 1 "
    goto   ic2_acknoack      ; poursuivre le traitement
i2c_ack
    bsf    STATUS,RP0        ; passer en banque 1
    bcf    SSPCON2,ACKDT     ; le bit qui sera envoyé vaudra " 0 "
ic2_acknoack
    bsf    SSPCON2,ACKEN     ; lancer l'acknowledge (= ACKDT = 0 = ACK)
    IWAIT  SSPCON2,ACKEN     ; attendre fin ACK
    return                    ; et retour
```

Nous avons besoin également de pouvoir écrire un octet. De nouveau, c'est très simple, l'octet sera passé par l'intermédiaire du registre « w » :

```
;*****
;                               ENVOYER UN OCTET                               *
;*****
;-----
; L'octet est passé dans W
;-----
i2c_write
    movwf  SSPBUF            ; lancer l'émission de l'adresse en mode écriture
    bsf    STATUS,RP0        ; passer en banque 1
    IWAIT  SSPSTAT,R_W       ; attendre émission terminée
    return                    ; et retour
```

Il ne nous reste plus qu'à pouvoir lire un octet, étant entendu que l'écriture de l'adresse du circuit est strictement identique à l'écriture d'un octet de donnée :

```
;*****
;                               LIRE UN OCTET                               *
;*****
;-----
; L'octet est retourné dans W
;-----
i2c_read
    bsf    STATUS,RP0        ; passer en banque 1
    bsf    SSPCON2,RCEN      ; lancer la lecture
    IWAIT  SSPCON2,RCEN      ; attendre réception terminée
    movf   SSPBUF,w          ; charger octet reçu
    return                    ; et retour

END                            ; directive fin de programme
```

Vous voyez la directive « end » de fin de programme, ce qui vous confirme que c'est terminé. Vous constatez que bien que cela semble un peu long, tout ceci est extrêmement simple. De plus, je vous le rappelle, vous n'écrirez jamais ces routines qu'une seule fois.

Lancez l'assemblage, placez votre fichier « lum_eep.hex » dans votre eeprom, et lancez l'alimentation. Vos LEDs clignotent suivant les données inscrites en eeprom.

23.13 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.

23.14 Conclusions

Ceci termine ce très long chapitre consacré au bus I²C. J'ai beaucoup développé, effectué plusieurs répétitions, mais ce bus est un bus très fréquemment utilisé, et il est plus que probable que si vous continuez l'aventure des PICs, vous serez amenés à l'utiliser de façon répétée.

23.15 Annexe : errata sur le I²C

Avant de pouvoir terminer cet ouvrage, Microchip vient de publier un errata, daté du 07/2002, concernant le fonctionnement du module I²C. Cet errata concerne les versions actuellement disponibles des PICs. A vous de vérifier le cas échéant si cet errata a été corrigé au moment où vous utiliserez cet ouvrage dans les PICs qui seront à ce moment en votre possession. Ce bug est dérangerant si vous utilisez votre PIC dans un environnement multi-maître. Voici de quoi il s'agit :

Lorsqu'un start-condition est placé sur la ligne, le bit « S » est positionné dans le registre SSPSTAT. Tant que ce bit est positionné, le module n'arrive plus à détecter un nouveau start-condition qui interviendrait par erreur.

Le bit « S » reste positionné jusque :

- La réception d'un stop-condition
- Le reset du module MSSP
- L'accusé de réception du second octet d'une adresse sur 10 bits.

Autrement dit :

Si, PAR ERREUR, un start-condition est rencontré au milieu d'une trame, il ne sera pas détecté comme une erreur, et sera interprété comme étant un bit de donnée.

De plus, un NOACK généré par l'esclave ne provoquera pas non plus le reset du module.

Malheureusement, ce bug affecte actuellement toutes les versions de PICs, y compris les tout nouveaux PIC 18xxx. Microchip annonce qu'il va tenter de corriger ce bug sur les nouvelles versions.

Dans l'attente, la solution proposée est la suivante :

- lancer un timer lors de la réception d'un start-condition. Le temps de débordement sera calculé en fonction du temps maximum séparant la réception d'un start-condition de l'effacement normal du bit « S ».
- Si le timer déborde (temps à établir par l'utilisateur), c'est que le stop-condition n'a pas été reçu dans le délai correct. L'utilisateur devra alors resetter le module, ce qui peut être fait en changeant provisoirement la valeur des bits SSPMx, puis les remettre à la valeur correcte pour l'application.

23.16 Bug en mode I²C esclave

Microchip a remplacé le précédent document concernant le bug en mode multimâtre par un nouveau document qui indique cette fois un bug en mode I²C esclave.

Ce bug intervient quand :

- Lorsque par erreur un repeated start-condition (SR) est reçu à l'intérieur d'un bit de donnée ou d'adresse. Dans ce cas, ce SR indésirable pourrait être interprété comme un bit de donnée.
- Lorsque un SR est reçu par l'esclave, en mode lecture, alors que la trame précédente s'adressait au même esclave et également en mode lecture. Ceci est beaucoup plus gênant, car correspond à un fonctionnement possible normal du mode I²C. Vous êtes dès lors contraints de prendre les précautions indiquées. Sans les dites précautions, vous risquez de voir votre PIC bloquer définitivement votre bus I²C.

La méthode de protection contre ce bug est identique à celle décrite précédemment, c'est-à-dire lancement d'un timer dès réception du start-condition, le débordement du timer indiquant que le stop-condition n'a pas été reçu dans les délais prévus. Ce délai dépend de la longueur des trames échangées, et donc de l'application.

Une fois l'erreur détectée, le module I²C devra être resetté. Ceci peut se faire de 2 façons :

- En changeant provisoirement la valeur des bits SSPM3 à SSPM0 du registre SSPCON1.
- En effaçant puis repositionnant le bit SSPEN du registre SSPCON1.

N'oubliez pas, si vous utilisez le module I²C, de vérifier toute éventuelle évolution de ce (ces ?) bug directement chez Microchip.

Notes : ...

24. Le module USART en mode série synchrone

24.1 Introduction

Et oui, nous avons 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. Son étude est donc loin d'être inutile, je dirai même qu'elle est indispensable.

USART signifie « **U**niversal **S**ynchronous **A**synchronous **R**eceiver **T**ransmitter ». 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 « **S**erial **C**ommunications **I**nterface ».

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 dédiée à l'horloge, il ne vous reste donc qu'une seule ligne pour transmettre les données.

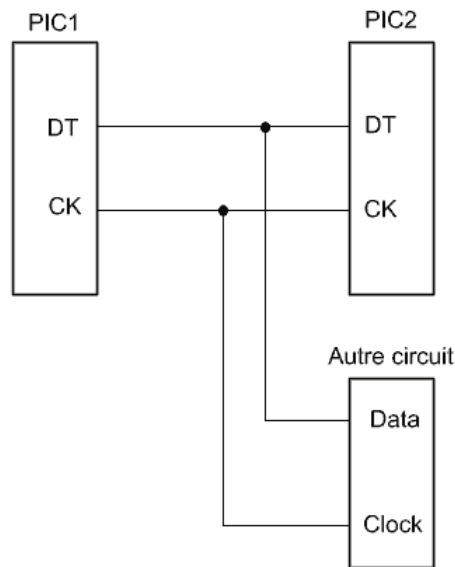
On en déduit que le PIC ne pourra émettre et recevoir en même temps en utilisant l'USART en mode synchrone. On parlera donc de liaison « half-duplex ». Par contre, le mode asynchrone n'a pas besoin de ligne d'horloge (voir cours sur les 16F84, chapitre sur la norme 7816), il nous restera alors 2 lignes pour communiquer, chacune étant dédiée à un sens de transfert. Nous pourrions 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 :

- Mode asynchrone full duplex : émission sur TX et réception sur RX
- Mode asynchrone half-duplex sur 2 lignes (TX et RX) ou sur une ligne (TX/RX reliées)
- Mode synchrone maître : émission horloge sur CK et émission/réception données sur DT
- Mode synchrone esclave : réception horloge sur CK et émission/réception données sur DT

Je vais donc, pour votre facilité, scinder de nouveau les explications en 2 chapitres. Ces chapitres auront pas mal de points communs, comme, par exemple, les registres utilisés. Je commence, pour rester dans la logique de ce que nous venons de voir, par le fonctionnement en mode synchrone.

24.2 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 :

- 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 ».
- Le « token-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).
- 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).
- 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 « token-ring ». Supposons qu'à la mise sous tension, le maître est le PIC1. 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 (token). 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'aaaaaaa', B'bbbbbbb'
- Le PIC 2 (adresse = 2) sait que les octets lui sont destinés, il n'a pas droit de réponse
- Le PIC1 envoie : B'01000010', B'ccccccc', B'ddddddd'
- 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 PIC2 : B'00000001', B'eeeeeee', B'ffffff'
- Le PIC1 envoie : B'10000011', B'ggggggg', B'hhhhhhh'
- Le PIC1 a donné le jeton (bit 7 à 1) au PIC3 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 « token-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).

24.3 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

- b7 : CSRC : Clock Source select bit (1 = master, 0 = slave)
- b6 : TX9 : TRANSMITT 9 bits enable bit (1 = 9 bits, 0 = 8 bits)
- b5 : TXEN : TRANSMITT ENable bit
- b4 : SYNC : SYNChronous mode select bit (1 = synchrone, 0 = asynchrone)
- b3 : N.U. : Non Utilisé : lu comme « 0 »
- b2 : non : non utilisé en mode synchrone
- b1 : TRMT : TRAnsMiT shift register status bit (1 = TSR vide)
- 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 (nous en verrons un exemple dans la partie dédiée à la communication asynchrone).

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 9^{ème} 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.

24.4 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

b7 : SPEN	: Serial Port ENable bit
b6 : RX9	: RECEIVE 9 bits enable bit (1 = 9 bits, 0 = 8 bits)
b5 : SREN	: Single Receive ENable bit
b4 : CREN	: Continuous Receive ENable bit
b3 : non	: non utilisé en mode synchrone
b2 : non	: non utilisé en mode synchrone
b1 : OERR	: Overflow ERRor bit
b0 : RX9D	: RECEIVE 9 th 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. J'expliquerai comment survient cette erreur. Sachez déjà qu'elle survient si vous n'avez pas traité de façon assez rapide les octets précédemment reçus.

RX9D contient le 9^{ème} 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.

24.5 Le registre SPBRG

Ce registre « **S**erial **P**ort **B**aud **R**ate **G**enerator » permet de définir la fréquence de l'horloge utilisée pour la transmission, et donc de fixer son débit. Le terme BRG doit maintenant vous être familier. Bien entendu, il ne s'agit pas du même BRG utilisé dans le module MSSP. Ces 2 modules sont complètement indépendants.

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.

Attention, je traite ici le cas de la liaison synchrone. Les formules sont différentes entre mode synchrone et mode asynchrone.

La formule qui donne le débit pour le mode synchrone est :

$$D = F_{osc} / (4 * (SPBRG + 1))$$

Comme vous aurez souvent besoin de calculer SPBRG en fonction du débit souhaité, voici la formule transformée :

$$SPBRG = (F_{osc} / (4 * D)) - 1$$

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 ?

$$\text{SPBRG} = (20.000.000 / (4 * 19200)) - 1 = 259,4.$$

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 $\text{SPBRG} = 255$, ce qui nous donne un débit réel de :

$$D = 20.000.000 / (4 * (255+1)) = 19531 \text{ bauds.}$$

L'erreur sera de :

$$\text{Erreur} = (19531 - 19200) / 19200 = 1,72\%$$

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 ».

$$D_{\text{max}} = 20000000 / (4 * (0+1)) = 5.000.000 = 5 \text{ MBauds.}$$

Vous constatez une fois de plus que les liaisons synchrones sont prévues pour travailler avec de très grandes vitesses.

24.6 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.

Exemple d'initialisation en mode maître :


```

Init
    bsf     STATUS,RP0      ; passer en banque 1
    movlw   B'10010000'    ; mode maître synchrone, émission 8 bits
    movwf   TXSTA           ; dans registre TXSTA
    movlw   BRGVAL         ; valeur calculée de SPBRG
    movwf   SPBRG          ; dans baud rate generator
    bcf     STATUS,RP0      ; repasser banque 0
    movlw   B'10000000'    ; module en service, réception 8 bits
    movwf   RCSTA          ; dans registre RCSTA

```

Exemple d'initialisation en mode esclave :

```

    bsf     STATUS,RP0      ; passer en banque 1
    movlw   B'00010000'    ; mode esclave synchrone, émission 8 bits
    movwf   TXSTA           ; dans registre TXSTA
    bcf     STATUS,RP0      ; repasser banque 0
    movlw   B'10000000'    ; module en service, réception 8 bits
    movwf   RCSTA          ; dans registre RCSTA

```

24.7 L'émission en mode maître

Je vais vous expliquer ici comment se passe une émission du point de vue du 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 9^{ème} 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 inversée, 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, si vous examinez les chronogrammes des figures 10-9 et 10-10 page 106 du datasheet, de gagner un peu de temps, surtout sur les communications à faibles débits.

Vous remarquerez en effet que la transmission commence, sur la figure 10-10, directement après positionnement du bit TXEN, alors qu'il y a un délai sur la figure 10-9 concernant la première solution, entre le chargement de TXREG et le début effectif de la transmission.

Un petit mot concernant cette différence. Elle est due au fait que le Baud Rate Generator commence à compter dès qu'un des bits TXEN, CREN ou SREN est positionné. Par contre, il est forcé à « 0 » dans le cas contraire.

Si on adopte la première méthode, vous voyez tout de suite que lorsque vous chargez TXREG, le BRG peut contenir n'importe quelle valeur. La transmission ne commencera donc que lors du prochain débordement du BRG. Ceci explique le délai.

Par contre, sur la seconde méthode, le BRG étant forcé à « 0 », dès qu'on positionne TXEN, il est prêt à envoyer la première donnée.

Ne vous tracassez pas outre mesure pour ceci, cela ne vous concernera principalement qu'en cas de temps de réponse critique avec des débits faibles, ce qui est relativement peu fréquent.

24.8 L'émission en mode esclave

En fait, c'est extrêmement simple. 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. Nous verrons une autre différence quand nous parlerons du mode sleep un peu plus loin.

24.9 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 le pendant pour la réception de 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).

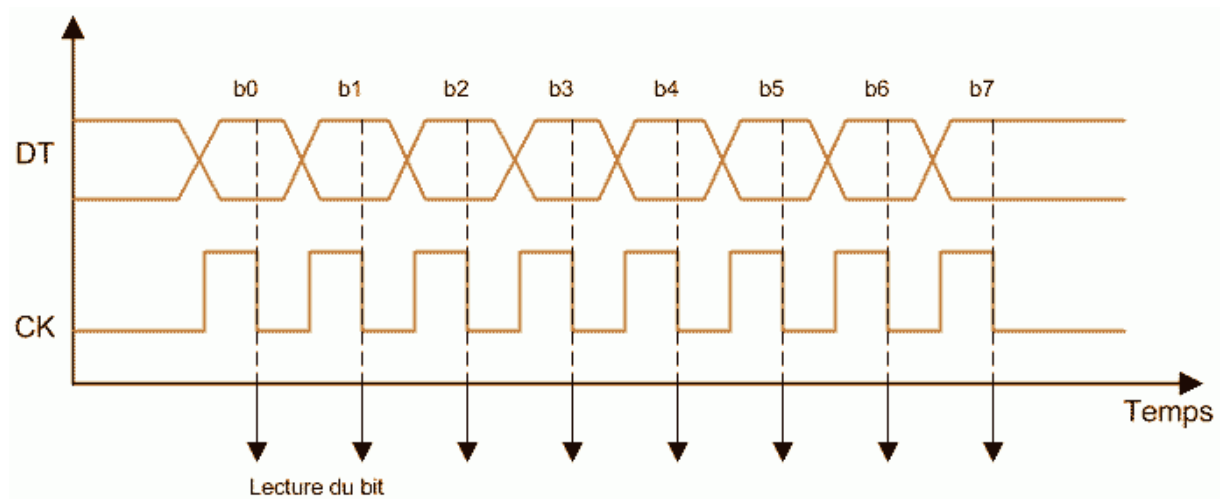
Dès que RSR est plein, son contenu est transféré dans **RCREG**, qui est le seul des 2 registres accessible par votre programme. Il contient la donnée effectivement reçue, complétée éventuellement (si vous travaillez sur 9 bits) par le bit **RX9D**.

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 9^{ème} 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 stoppé. 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 :

24.9.1 La file FIFO de RCREG

FIFO, FIFO ? Qu'est-ce donc ceci ? En fait, **FIFO** est l'abréviation de « **F**irst **I**n **F**irst **O**ut », ou, en français, « Premier Entré, Premier Sorti ». Il s'agit **d'une file**, à interpréter comme la file d'attente d'un magasin (pas celle de la poste, notre PIC ne dispose pas d'assez de place).

Dans une file de ce type, les nouveaux événements (clients pour le magasin, données pour notre PIC), arrivent en bout de file. Le premier événement à traiter sera bien entendu le premier arrivé (sauf si vous aimez les agitations musclées).

Tout ce que l'agent (La caissière, ou notre programme) aura à savoir, c'est s'il reste au moins un événement (client ou donnée) à traiter.

Et bien, **notre 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 9^{ème} 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 9^{ème} 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 9^{ème} 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.

24.9.2 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.

24.10 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.

24.11 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.

24.12 Différences entre MSSP et USART

Vous disposez maintenant de 2 modules capables de gérer une liaison série synchrone, à savoir le module MSSP en mode SPI, et le module USART en mode synchrone. Quand devez-vous utiliser un ou l'autre de ces modules ?

Peut-être êtes-vous déjà capables de répondre à tout ou partie de cette question d'après ce que je vous ai expliqué. Le fonctionnement de ces modules est assez dissemblable, et donc induit des contraintes et des possibilités différentes.

- **Le module MSSP émet les bits en commençant par le bit 7 jusqu'au bit 0**, alors que **l'USART procède de façon inverse**. Il va de soi que si vous interfacez 2 PICs dont vous écrivez vous-même le programme, cela ne posera pas de problème (comme pour plusieurs des autres différences). Par contre, si vous vous connectez dans un circuit existant, cette différence prendra tout son sens.
- **Une liaison SPI est toujours une liaison full-duplex**, qui nécessite donc **3 fils de liaison**. Ceci exclut le fonctionnement multi-maître (sans artifice), puisque vous devez croiser les fils d'émission et de réception. Par contre, **l'USART travaille sur 2 fils et en mode half-duplex**, l'inconvénient de la transmission unidirectionnelle étant compensée par l'absence de croisement des fils, donc de la possibilité de travailler en multi-maîtres.
- **Le MSSP dispose d'une pin de validation « SS »** qui n'est pas présente dans le module USART. Cette pin n'est d'ailleurs pas vraiment nécessaire dans ce module, puisque le mode half-duplex, qui n'implique pas une réponse simultanée, permet de recevoir et de traiter une adresse du destinataire.

- Le module MSSP, à fréquence de quartz égale du PIC, permet de travailler avec des vitesses plus basses que pour le module USART.

Voici déjà quelques différences flagrantes entre ces 2 modules, qui ne devraient pas vous laisser dans l'expectative au moment du choix dans une application spécifique.

24.13 Exercice pratique

Nous voici arrivé au stade de notre exercice pratique. Je vous propose un exemple qui met en œuvre les 4 modes décrits, afin de vous permettre de voir comment utiliser votre PIC dans toutes les situations.

Nous allons donc connecter ensemble 2 PICs en utilisant leur module USART. Le premier sera le maître permanent, le second l'esclave. Le fonctionnement sera le suivant :

- Le PIC maître envoie à l'esclave un mot de 9 bits , dont les 8 bits principaux représentent une adresse relative, le 9^{ème} bit indiquant quelle zone de données a été choisie .
- L'esclave lit l'octet dans son buffer dont l'adresse est donnée par les 8 bits reçus. Le 9eme bit précise si le buffer utilisé est celui de la banque 0 ou de la banque 1.
- L'esclave envoie l'octet lu au maître sur 8 bits. Le 9eme bit est utilisé pour dire au maître si la fin de la zone est atteinte ou non (dernier élément).
- Le maître réceptionne l'octet, et place la donnée sur les LEDs connectées sur le PORTB.

Bien entendu, cet exercice n'est guère utile, mais il présente l'avantage de mettre en œuvre réception et émission, en mode esclave et maître, et en plus sur 9 bits. Après tout, c'est ce qu'on lui demande, non ?

Voici le schéma retenu :


```
; positionné, indique que l'adresse envoyée est la dernière du buffer      *
;                                                                           *
;*****
LIST      p=16F876                ; Définition de processeur
#include <p16F876.inc>             ; fichier include

__CONFIG  _CP_OFF & _DEBUG_OFF & _WRT_ENABLE_OFF & _CPD_OFF & _LVP_OFF &
_BODEN_OFF & _PWRTE_ON & _WDT_ON & _HS_OSC
```

Nous allons avoir besoin de 3 valeurs. Les deux premières seront utilisées pour établir le délai de 0.5 seconde, exactement comme dans l'exercice précédent. Je n'y reviendrai pas. La troisième valeur est celle que nous devons entrer dans SPREG, afin d'établir la vitesse de transmission. Comme j'ai choisi 38400 bauds, ceci donnera :

$$SPBRG = (Fosc / (4 * D)) - 1$$

Soit, avec notre quartz de 20 MHz :

$$SPBRG = (20.000.000 / (4 * 38400)) - 1 = 129,2$$

Nous choisirons de mettre la valeur 129, ce qui nous donnera un débit réel de :

$$D = Fosc / (4 * (SPBRG + 1))$$

Soit :

$$D = 20.000.000 / (4 * (129+1)) = 38461 \text{ bauds.}$$

L'erreur n'a ici pas la moindre importance, vu que nous travaillons en synchrone, et que, de plus, les 2 interlocuteurs sont des PICs. Souvenez-vous que l'esclave n'a même pas besoin de savoir à quel débit on travaille. Il se contente de réagir aux signaux d'horloge reçus. Ceci nous donnera les assignations suivantes :

```
;*****
;                               ASSIGNATIONS PROGRAMME                      *
;*****

PR2VAL    EQU    D'249'        ; Valeur de comparaison timer 2
CMPTVAL    EQU    D'125'        ; 125 passages dans la routine d'interruption
                                   ; durée = Tcy*(PR2+1)*prédiv*postdiv*cmptval
                                   ; = 0,2µs * 250 * 16 * 5 * 125 = 0.5s.
BRGVAL     EQU    D'129'        ; pour un débit de 38461 bauds
```

Dans notre zone de variables en banque 1, nous allons retrouver notre compteur de passages dans la routine d'interruption du timer 2 :

```
;*****
;                               VARIABLES BANQUE 0                          *
;*****

; Zone de 80 bytes
; -----
```

```

CBLOCK 0x20      ; Début de la zone (0x20 à 0x6F)
cmtpt : 1        ; compteur de passages d'interruption
ENDC            ; Fin de la zone

```

Dans la zone commune, on retrouve nos variables de sauvegarde, mais également d'autres variables utilisées dans notre programme, comme :

- Des flags
- L'octet à envoyer à l'esclave
- Un compteur de boucles.

J'ai choisi de placer ces variables en zone commune, car cela diminuera le nombre de changements de banques dans le programme. En effet, plusieurs registres utilisés continuellement (TXSTA) se trouvent en banque 1, et se retrouvent impliqués dans des boucles. Comme nos variables se retrouvent dans certaines de ces boucles, le fait de pouvoir y accéder depuis n'importe quelle banque se révélera pratique. Nous obtenons donc :

```

;*****
;                               VARIABLES ZONE COMMUNE                               *
;*****

; Zone de 16 bytes
; -----

CBLOCK 0x70      ; Début de la zone (0x70 à 0x7F)
w_temp : 1       ; Sauvegarde registre W
status_temp : 1  ; sauvegarde registre STATUS
FSR_temp : 1     ; sauvegarde FSR (si indirect en interrupt)
flags : 1        ; 8 flags d'usage général
                ; b0 : une demi-seconde s'est écoulée
                ; b1 : 4ms se sont écoulées
                ; b2 : état du 9eme bit à envoyer
                ; b3 : état du 9eme bit reçu

numoctet : 1     ; numéro d'octet envoyé à l'esclave
numloop : 1      ; nombre de boucles à effectuer
ENDC

#define TIME05S flags,0 ; 0.5 secondes écoulées
#define TIME4MS flags,1 ; 4ms écoulées
#define EMI9 flags,2   ; valeur du 9eme bit à envoyer
#define REC9 flags,3   ; valeur du 9eme bit reçu

```

Remarquez les définitions de bits, qui nous éviteront de traîner des « flags,x » dans le programme. C'est du reste plus parlant.

Le démarrage sur reset se passe de commentaires (excepté le présent commentaire qui indique qu'on se passe de commentaires) :

```

;*****
;                               DEMARRAGE SUR RESET                               *
;*****

org 0x000      ; Adresse de départ après reset
goto init      ; Initialiser

```

Notre routine d'interruption ne va gérer que l'interruption du timer 2. J'ai suffisamment parlé de cette interruption dans plusieurs exercices précédents, je n'y reviens donc pas :

```

; //////////////////////////////////////
;                               I N T E R R U P T I O N S
; //////////////////////////////////////

;*****
;                               ROUTINE INTERRUPTION
;*****
;-----
; La routine d'interruption timer 2 est appelée toutes les
; (0,2µs * 80 * 250) = 4ms.
; au bout de 125 passages, une demi-seconde s'est écoulée, on positionne
; le flag
;-----
;sauvegarder registres
;-----
org 0x004                ; adresse d'interruption
movwf w_temp             ; sauver registre W
swapf STATUS,w           ; swap status avec résultat dans w
movwf status_temp        ; sauver status swappé
movf FSR , w              ; charger FSR
movwf FSR_temp           ; sauvegarder FSR

; interruption timer 2
; -----
BANKSEL PIR1             ; passer banque 0
bcf PIR1,TMR2IF           ; effacer le flag d'interruption
bsf TIME4MS               ; positionner flag pour 4ms
decfsz cmpt,f            ; décrémenter compteur de passages
goto restorereg           ; pas 0, fin de l'interruption
movlw CMPTVAL             ; valeur de recharge du compteur
movwf cmpt               ; recharger compteur
bsf TIME05S               ; positionner flag pour 0.5 seconde

;restaurer registres
;-----
restorereg
movf FSR_temp,w          ; charger FSR sauvé
movwf FSR                 ; restaurer FSR
swapf status_temp,w      ; swap ancien status, résultat dans w
movwf STATUS             ; restaurer status
swapf w_temp,f           ; Inversion L et H de l'ancien W
swapf w_temp,w           ; Réinversion de L et H dans W
retfie                   ; return from interrupt

```

Nous arrivons aux initialisations, en commençant par les PORTs. Comme nous n'utilisons en tant que tel que le PORTB, et que les bits utilisés pour la transmission sont dans le bon état par défaut (en entrée), cette partie est très courte. Notez la bonne pratique qui consiste à couper les résistances de rappel au +5V lorsqu'on ne s'en sert pas, cela diminue la consommation générale.

```

; //////////////////////////////////////
;                               P R O G R A M M E
; //////////////////////////////////////

;*****
;                               INITIALISATIONS
;*****
init
; initialisation PORTS (banque 0 et 1)
; -----

```

```

BANKSEL PORTA      ; sélectionner banque0
clrf  PORTB        ; sorties PORTB à 0
bsf   STATUS,RP0   ; passer en banque1
clrf  PORTB        ; PORTB en sortie (LEDs)
bsf   OPTION_REG,7 ; couper résistances de rappel

```

Nous en sommes à la partie qui nous intéresse, à savoir l'initialisation de notre USART . Ceci s'effectue sans problème, en suivant la théorie expliquée. Nous choisirons d'utiliser ce PIC en maître, émission et réception sous 9 bits, module USART en service, et BRG correspondant à 38400 bauds (D'129').

```

; initialiser USART
; -----
movlw B'11010000'   ; mode maître synchrone, émission 9 bits
movwf TXSTA         ; dans registre TXSTA
movlw BRGVAL        ; valeur calculée de SPBRG
movwf SPBRG         ; dans baud rate generator
bcf   STATUS,RP0    ; repasser banque 0
movlw B'11000000'   ; module en service, réception 9 bits
movwf RCSTA         ; dans registre RCSTA

```

Le reste concerne l'initialisation du timer 2, des variables, et la mise en service de l'interruption timer 2.

```

; initialiser timer 2
; -----
movlw PR2VAL        ; charger valeur de comparaison
BANKSEL PR2         ; passer banque 1
movwf PR2           ; initialiser comparateur
movlw B'00100110'   ; timer2 on, prédiv = 16, post = 5
BANKSEL T2CON       ; passer banque 0
movwf T2CON         ; lancer timer 2

; initialiser variables
; -----
movlw CMPTVAL        ; pour durée initiale de 0.5s.
movwf cmpt           ; dans compteur de passages
clrf  flags          ; effacer flags
clrf  numoctet       ; on commence par l'élément 0

; lancer interruption timer2
; -----
bsf   INTCON,PEIE    ; interruptions périphériques en service
bsf   STATUS,RP0     ; passer banque 1
bsf   PIE1,TMR2IE    ; lancer interruptions timer2
bcf   STATUS,RP0     ; repasser banque 0
bsf   INTCON,GIE     ; lancer les interruptions
goto  start          ; démarrer programme principal

```

Au tour de nos différentes fonctions (j'ai voulu dire « sous-routines »). Tout d'abord une attente de 0.5 seconde :

```

;*****
;                               Attendre 0.5 seconde                               *
;*****
;-----
; attendre qu'une seconde se soit écoulée depuis le précédent passage dans
; cette routine
;-----

```

```
wait
    clrwdt                ; effacer watchdog
    btfss flags,0         ; flag positionné?
    goto wait             ; non, attendre flag
    bcf TIME05S           ; reset du flag pour 0.5 seconde
    bcf TIME4MS           ; et reset du flag de 4ms
    return                ; et retour
```

Très simple, comme vous voyez. Le flag TIME4S, remis à 0 ici, et positionné à chaque passage dans la routine d'interruption timer 2 (toutes les 4ms), est destiné à nous donner un temps intermédiaire de 4ms. Si vous comprenez le mécanisme :

- On attend la fin des 0.5 secondes : les 2 flags sont effacés
- 4ms plus tard, le flag TIME4MS est positionné
- 0.5 seconde après l'attente des 0.5 précédente, le flag TIME05S est positionné.

Donc, en toute logique, voici notre routine qui attend le positionnement du flag TIME4MS :

```
;*****
;                               Attendre 4 ms                               *
;*****
;-----
; attendre que 4ms se soient écoulées depuis le précédent passage dans
; la routine wait
;-----
wait4ms
    clrwdt                ; effacer watchdog
    btfss TIME4MS         ; flag positionné?
    goto wait4ms          ; non, attendre flag
    return                ; et retour
```

Inutile ici de resetter notre flag TIME4MS, il serait repositionné 4ms plus tard, et nous n'en avons plus besoin qu'après la prochaine attente de 0.5 seconde.

Voici maintenant une routine plus intéressante. Elle envoie la valeur contenue dans la variable « numoctet » sur le port série.

```
;*****
;                               ENVOYER UN OCTET                               *
;*****
;-----
; Envoie l'octet numoctet vers l'esclave.
; attend la fin de la transmission pour sortir
; le 9eme bit du mot à envoyer est dans EMI9
;-----
sendusart
    bsf STATUS,RP0        ; passer en banque 1
    bcf TXSTA,TX9D        ; par défaut, 9eme bit = 0
    btfsc EMI9            ; 9eme bit doit valoir 1?
    bsf TXSTA,TX9D        ; oui, alors le positionner
    movf numoctet,w        ; charger octet à envoyer
    bcf STATUS,RP0        ; passer banque 0
    movwf TXREG            ; octet dans registre de transmission
    bsf STATUS,RP0        ; repasser banque 1
    bsf TXSTA,TXEN        ; envoyer l'octet
```

```

clrwdt          ; effacer watchdog
btfss TXSTA,TRMT ; transmission terminée?
goto $-2        ; non, attendre
bcf TXSTA,TXEN   ; oui, fin de transmission
bcf STATUS,RP0   ; repasser banque 0
return          ; et retour

```

Vous notez que le 9eme bit envoyé sera le reflet du flag EMI9, positionné par notre programme principal. J'ai choisi d'attendre la fin de l'émission avant de sortir de cette routine. Comme notre watchdog est en service, nous le resettons dans la boucle d'attente.

On envoie, et on doit recevoir, voici la sous-routine qui le permet :

```

; *****
;                                     RECEVOIR UN OCTET                                     *
; *****
; -----
; Reçoit un octet de l'esclave.
; l'octet reçu sera retourné dans W
; le 9eme bit du mot reçu sera retourné dans REC9
; Comme on effectue une lecture simple (SREN), le bit SREN est resetté
; automatiquement en fin d'émission. On se sert de cette particularité pour
; tester si la transmission, est terminée. On pouvait aussi tester le passage
; à "1" du bit RCIF.
; En réalité, vu que l'émission suivante a lieu 0.5s plus tard, on pouvait
; fort bien se passer d'attendre la fin de la réception. J'ai inclus le test
; à titre d'exemple.
; -----
recusart
    bsf  RCSTA,SREN      ; un seul octet à recevoir
    clrwdt              ; effacer watchdog
    btfsc RCSTA,SREN     ; réception terminée?
    goto $-2            ; non, attendre
    bcf  REC9            ; par défaut, 9eme bit reçu = 0
    btfsc RCSTA,RX9D     ; 9eme bit reçu = 1?
    bsf  REC9            ; oui, signaler
    movf RCREG,w         ; lire octet reçu
    return              ; et retour

```

Comme il est indiqué dans les commentaires, 2 remarques ici :

- La détection de la fin de transmission peut se faire soit sur le relâchement du bit « SREN », vu qu'il s'agit ici d'une émission unique, soit sur le positionnement du bit RCIF qui intervient une fois que l'octet reçu se trouve dans RCREG.
- Il n'était pas nécessaire d'attendre la fin de la réception avant de continuer, vu que, de toute façon, notre émission suivante s'effectuera pratiquement 0.5 seconde plus tard, l'émission sera donc terminée à ce moment.

Voici maintenant la sous-routine qui appelle les précédentes, elle réalise les opérations suivantes :

- On attend la fin de la demi-seconde courante
- On envoie le numéro de l'octet souhaité à l'esclave
- On attend que l'esclave ait eu le temps de lire et de préparer sa réponse.
- On lance la lecture de l'octet renvoyé par l'esclave

- On place l'octet lu (bits 0 à 7) sur le PORTB qui allume les LEDs
- On incrémente le numéro de l'octet souhaité pour la prochaine boucle
- On examine le bit 8 (9^{ème} bit) reçu de l'esclave.
- S'il vaut 0, on recommence au début
- S'il vaut 1, c'était le dernier, dans ce cas :
 - On remet le numéro de bit à 0
 - On regarde si on doit encore boucler
 - Si oui, on recommence une série de boucles, si non, on sort

Nous allons devoir estimer le temps que met notre esclave à préparer sa réponse :

- Si on part du principe que nous traiterons la réception sur notre esclave en mode interruption
- Si on pose que le PIC esclave était en mode sleep au moment de la réception
- Si on se souvient que le temps commence à compter à partir de la sortie de la routine « wait », on peut estimer :

Temps de réaction = temps de l'émission vers l'esclave + temps de réveil de l'esclave + temps nécessaire à l'esclave pour entrer dans la routine d'interruption, lire l'octet reçu, chercher la donnée demandée et la placer dans le registre d'émission.

On va donc compter « à la louche », faute de ne pas avoir encore le logiciel de l'esclave :
Temps d'émission = durée d'un bit * nombre de bits émis

Temps d'émission = $(1s / 38400 \text{ bauds}) * 9 \text{ bits} = 234 \mu s$

Temps de réveil de l'esclave = $1024 T_{osc} = 1024 * (1/20 \text{ MHz}) = 51,2 \mu s$.

Si on estime grossièrement que 200 cycles d'instructions seront suffisants pour traiter l'information, nous aurons encore besoin de $200 * 4 / 20\text{MHz} = 40 \mu s$.

Le temps total de réaction sera de : $234 + 51,2 + 40 = 325 \mu s$.

Nous passons dans notre routine timer 2 toutes les 4ms. Comme nous ne sommes pas spécialement pressé dans cette application, nous en profitons donc pour attendre 4ms au lieu des 325 μs minimales requises. Nous voyons que notre PIC esclave dispose de tout son temps pour répondre. Ceci explique la sous-routine « wait4ms ».

Dans une application PIC, il faut toujours pouvoir établir des ordres de grandeur concernant les temps mis en jeu, ça facilite la réalisation des programmes.

Il faut toujours également se poser la question de l'utilité ou non des calculs précis et savoir si cela vaut la peine de compliquer le programme pour optimiser certains délais.

Dans ce cas, si nous avons opté pour fabriquer un délai de 325 μs , notre programme aurait fonctionné exactement de la même façon, les octets défilant toutes les 0.5 seconde. Il n'est donc pas utile de compliquer le programme pour fabriquer ce délai précis. Autant utiliser une ressource déjà implémentée, à savoir un délai de 4ms.

Tout ceci nous donne :


```

;*****
;                               CHENILLARD SUR UNE SERIE DE DATA                               *
;*****
;-----
; le numéro du buffer de l'esclave est dans BIT8 (9ème bit)
; le numéro de l'octet demandé est dans numoctet
; Le 9eme bit positionné de l'octet reçu indique que le buffer de l'esclave
; ne contient plus de données
; Le nombre de boucles à réaliser est dans w
;-----
chenill
    movwf numloop          ; sauver dans compteur de boucles
chenloop
    call wait              ; attendre 0.5 seconde
    call sendusart         ; envoyer le numéro de l'octet
    call wait4ms           ; attendre que l'esclave soit prêt
    call recusart          ; réceptionner octet
    movwf PORTB            ; placer octet reçu sur les LEDs
    incf numoctet,f        ; incrémenter numéro de l'octet
    btfss REC9             ; reçu 9eme bit = 1? (fin de data)
    goto chenloop          ; non, octet suivant
    clrf numoctet          ; oui, reprendre depuis le début
    decfsz numloop,f       ; décrémenter compteur de boucles
    goto chenloop          ; pas dernière, suivante
    return                ; fini, retour

```

A ce stade, il ne reste plus qu'à construire notre programme principal. Celui-ci va se contenter, afin qu'on puisse voir si tout fonctionne bien, de lancer 3 fois la précédente routine avec le 9eme bit à « 0 », suivi de 4 fois la même routine, mais avec le 9eme bit à « 1 ».

Ceci se traduira par 3 séquences d'allumage des LEDs correspondant à la première zone de données stockées dans l'esclave, suivies par 4 séquences correspondant à la seconde zone de données. On reprendra ensuite le tout au début.

```

;*****
;                               PROGRAMME PRINCIPAL                               *
;*****
start
    clrwdt                 ; effacer watch dog
    bcf EMI9               ; signaler 9eme bit = 0 (première série de data)
    movlw 0x03             ; 3 boucles à effectuer
    call chenill           ; exécuter les 3 boucles
    bsf EMI9               ; signaler 9eme bit = 1 (seconde série de data)
    movlw 0x04             ; 4 boucles à effectuer
    call chenill           ; exécuter les 4 boucles
    goto start             ; boucler
    END                   ; directive fin de programme

```

Toujours la méthode habituelle de travailler en programmation dite « structurée », avec des routines courtes et hiérarchisées et un programme principal réduit à l'appel de fonctions réalisées sous forme de sous-routines.

Nous en avons terminé avec le programme à placer dans le PIC maître. Nous allons nous attaquer à celui de l'esclave, encore plus simple. Commencez par effectuer un copier/coller de votre fichier « **m16F876.asm** », renommez cette copie « **UsartS.asm** », et créez un nouveau projet du même nom.

Selon notre habitude, commençons par l'en-tête et les bits de configuration. Comme j'ai l'intention cette fois d'utiliser le mode « sleep », nous ne validerons pas notre watchdog, qui réveillerait la PIC trop tôt, et donc à un moment inopportun.

```
;*****
; Exercice de communication série synchrone sur base du module USART      *
; Ce programme est le programme de l'esclave                             *
; Utilisation d'une liaison sur 9 bits à 38400 bauds                        *
;                                                                           *
;*****
;
; NOM:      USARTS                                                         *
; Date:     17/07/2002                                                     *
; Version:  1.0                                                            *
; Circuit:  Platine d'expérimentation                                       *
; Auteur:   Bigonoff                                                       *
;                                                                           *
;*****
;
; Fichier requis: P16F876.inc                                              *
;                                                                           *
;*****
;
; Le PIC reçoit une adresse en provenance du maître.                      *
; Le 9eme bit contient la zone de data concernée (0 ou 1).                *
; Le PIC renvoie l'octet correspondant à l'adresse reçue.                 *
; Ce cet octet est le dernier de la zone, le 9eme bit sera positionné.    *
;                                                                           *
;*****
;
LIST      p=16F876                ; Définition de processeur
#include <p16F876.inc>             ; fichier include

__CONFIG  _CP_OFF & _DEBUG_OFF & _WRT_ENABLE_OFF & _CPD_OFF & _LVP_OFF &
_BODEN_OFF & _PWRTE_ON & _WDT_OFF & _HS_OSC
```

Pour les mêmes raisons que dans le programme du maître, j'ai choisi de placer mes variables dans la zone RAM commune. Si vous voulez, à titre d'exercice, réalisez ce programme en plaçant les variables en banque 1, vous verrez alors l'avantage de procéder comme je l'ai fait dans ce cas.

Les variables sont d'ailleurs réduites à leur plus simple expression. En effet, pour l'esclave, nul besoin de temps d'attente puisqu'il doit répondre dès l'interrogation du maître. J'utiliserai donc seulement 2 « flags » ou indicateurs, pour signaler l'état du 9^{ème} bit reçu et celui du 9^{ème} bit à émettre.

Etant donné qu'on n'envoie et qu'on ne reçoit qu'un seul octet à la fois, les données présentes dans RX9D et dans TX9D ne seront pas écrasées entre 2 échanges. J'aurais donc pu utiliser directement les valeurs de ces bits. Mais ce ne sera pas toujours le cas dans vos programmes, aussi ai-je préféré vous montrer une méthode plus générale.

```
;*****
;                               VARIABLES ZONE COMMUNE                      *
;*****
; Zone de 16 bytes
; -----
```

```

CBLOCK 0x70          ; Début de la zone (0x70 à 0x7F)
w_temp : 1           ; Sauvegarde registre W
status_temp : 1      ; sauvegarde registre STATUS
FSR_temp : 1         ; sauvegarde FSR (si indirect en interrupt)

flags : 1            ; 8 flags d'usage général
                        ; b0 : état du 9eme bit à envoyer
                        ; b1 : état du 9eme bit reçu
octet : 1            ; octet reçu ou à envoyer
ENDC

#define EMI9 flags,0   ; valeur du 9eme bit à envoyer
#define REC9 flags,1   ; valeur du 9eme bit reçu

```

Le démarrage sur le reset, toujours identique :

```

;*****
;                               DEMARRAGE SUR RESET                               *
;*****

org 0x000          ; Adresse de départ après reset
goto  init         ; Initialiser

```

Nous voici aux interruptions. J'ai choisi de traiter la réception d'un octet par interruption. En effet, le temps séparant 2 réceptions consécutives est très long (0,5 seconde). J'en profiterai donc pour placer le PIC en mode « sleep » dans l'attente de l'octet en provenance du maître. Il sera réveillé par l'interruption. Bien entendu, je pouvais intégrer le traitement dans le programme principal (il me suffisait de ne pas activer le bit GIE). Mais, ce traitement est plus général, et donc plus didactique.

La routine d'interruption de réception sur USART se contente de lire et de sauver l'octet reçu (ce qui efface le flag RCIF), et de positionner le flag REC9 en fonction de la valeur du 9^{ème} bit reçu. On en profite pour mettre fin au mode de réception.

```

; //////////////////////////////////////
;                               I N T E R R U P T I O N S                               *
; //////////////////////////////////////

;*****
;                               ROUTINE INTERRUPTION                               *
;*****

                ;sauvegarder registres
                ;-----
org 0x004          ; adresse d'interruption
movwf w_temp      ; sauver registre W
swapf STATUS,w    ; swap status avec résultat dans w
movwf status_temp ; sauver status swappé
movf FSR , w      ; charger FSR
movwf FSR_temp    ; sauvegarder FSR

                ; Interruption réception USART
                ; -----
BANKSEL RCSTA     ; passer banque 0
bcf REC9          ; par défaut, 9eme bit reçu = 0
btfsc RCSTA,RX9D  ; tester si 9eme bit reçu = 1
bsf REC9          ; oui, signaler 9eme bit = 1

```

```

movf   RCREG,w           ; charger octet reçu
movwf  octet             ; le sauver
bcf    RSTA,CREN         ; fin de réception

                ; restaurer registres
                ; -----
restorereg
movf   FSR_temp,w        ; charger FSR sauvé
movwf  FSR               ; restaurer FSR
swapf  status_temp,w     ; swap ancien status, résultat dans w
movwf  STATUS            ; restaurer status
swapf  w_temp,f          ; Inversion L et H de l'ancien W
                ; sans modifier Z
swapf  w_temp,w          ; Réinversion de L et H dans W
                ; W restauré sans modifier status
retfie                    ; return from interrupt

```

La routine d'initialisation se contente d'initialiser et de mettre en service le module USART en mode esclave asynchrone, et de valider les interruptions de la réception USART.

```

; //////////////////////////////////////
;                                     P R O G R A M M E
; //////////////////////////////////////

;*****
;                                     INITIALISATIONS
;*****
init
                ; Registre d'options
                ; -----
BANKSEL  OPTION_REG    ; passer banque 1
bsf      OPTION_REG,7   ; couper résistances de rappel

                ; initialiser USART
                ; -----
movlw   B'01010000'     ; mode esclave synchrone, émission 9 bits
movwf   TXSTA           ; dans registre TXSTA
bcf     STATUS,RP0      ; repasser banque 0
movlw   B'11000000'     ; module en service, réception 9 bits
movwf   RCSTA           ; dans registre RCSTA

                ; lancer interruption USART
                ; -----
bsf     INTCON,PEIE      ; interruptions périphériques en service
bsf     STATUS,RP0      ; passer banque 1
bsf     PIE1,RCIE       ; autoriser interrupts RX USART
bcf     STATUS,RP0      ; repasser banque 0
bsf     INTCON,GIE      ; lancer les interruptions
goto    start           ; démarrer programme principal

```

Il fallait bien mettre les données à envoyer quelque part. J'ai choisi de les ranger en mémoire programme. Il nous faut donc une routine qui puisse lire ces données en mémoire flash. De plus, nous aurons 2 zones, la zone concernée sera désignée par le maître, notre routine devra donc effectuer la sélection.

Il faudra de plus indiquer au maître si l'octet demandé est le dernier de la zone. Comme les données rangées en mémoire programme sont des mots de 14 bits, et que nous n'en

utiliserons que 8, j'ai choisi d'utiliser le bit 8 (b0 du poids fort de la donnée) pour indiquer que nous nous trouvons en présence du dernier mot de la zone. Il nous suffira alors de renvoyer ce bit comme 9^{ème} bit du mot à destination du maître.

```
;*****
;                               LIRE UN OCTET EN MEMOIRE PROGRAMME                               *
;*****
;-----
; Lit 1'octet à l'adresse "octet" en mémoire programme.
; Positionne EMI9 suivant le 9eme bit du mot de 14 bits correspondant
;-----
readflash
    ; pointer sur début de la zone de données
    ; -----
    BANKSEL    EEADRH        ; positionner banque 2
    movlw HIGH zonedat1      ; par défaut, MSB zone de données 1
    btfsc REC9               ; zone 2 demandée?
    movlw HIGH zonedat2      ; oui, MSB zone de données 2
    movwf EEADRH             ; positionner pointeur adresse MSB
    movlw LOW zonedat1       ; par défaut, LSB zone de données 1
    btfsc REC9               ; zone 2 demandée?
    movlw LOW zonedat2       ; oui, LSB zone de données 2
    movwf EEADR              ; positionner pointeur adresse MSB

    ; Ajouter offset demandé
    ; -----
    movf octet,w             ; charger octet reçu = numéro de l'octet
    addwf EEADR,f            ; ajouter au pointeur d'adresse LSB
    btfsc STATUS,C           ; débordement?
    incf EEADRH,f            ; oui, incrémenter pointeur poids fort

    ; Lire l'octet pointé
    ; -----
    bsf STATUS,RP0           ; passer banque 3
    bsf EECON1,EEPGRD        ; pointer sur zone flash
    bsf EECON1,RD            ; lire l'octet pointé
    nop                      ; attendre
    nop                      ; attendre
    bcf STATUS,RP0           ; passer banque 2
    movf EEDATA,w            ; charger 8 bits de données
    movwf octet              ; sauver l'octet lu
    bcf EMI9                 ; par défaut, pas dernier octet
    btfsc EEDATH,0           ; 9eme bit de donnée = 1?
    bsf EMI9                 ; oui, le signaler
    bcf STATUS,RP1          ; repasser banque 0
    return                   ; et retour
```

Nous en arrivons déjà à notre programme principal. Ce dernier va devoir réaliser les opérations suivantes :

- Lancer la lecture d'un octet
- Attendre que le maître décide d'envoyer un octet, et que ce dernier soit reçu
- Lire l'octet et stopper la réception
- Sélectionner la zone de données concernée, et lire l'octet demandé en mémoire flash
- Préparer l'émission de l'octet concerné
- Attendre que le maître se décide à lire l'octet préparé (approximativement 4ms plus tard)
- Stopper le mode d'émission et reprendre au début

Voyons tout ceci en détail, et tout d'abord :

- Lancer la lecture d'un octet
- Attendre que le maître décide d'envoyer un octet, et que ce dernier soit reçu

```
;*****  
;  
;          PROGRAMME PRINCIPAL          *  
;*****  
start  
        ; lecture sur port série  
        ; -----  
  
bsf     RCSTA,CREN      ; lancer la réception  
sleep   ; attendre un octet reçu
```

J'ai choisi dans l'attente (très longue) de la réception d'un octet, de placer le PIC en mode « sleep ». Ceci présente l'avantage de ne pas devoir « spooler » en permanence l'état d'un flag. Si nous avions utilisé le watchdog, nous aurions du insérer une boucle supplémentaire, notre watchdog réveillant le PIC à intervalles réguliers. Rien de bien dramatique, donc.

Pour rappel, puisque RCIE est positionné, une fois RCIF positionné (ce qui intervient lorsqu'un octet a été reçu et placé dans RCREG, le PIC sera réveillé. Comme les bit GIE et PEIE sont positionnés, l'instruction suivante sera exécutée, puis le PIC se connectera sur notre routine d'interruption.

Ensuite, nous avons :

- Lire l'octet et stopper la réception

Comme nous l'avons vu, ceci est entièrement géré par notre routine d'interruption.

Puis nous trouvons :

- Sélectionner la zone de données concernée, et lire l'octet demandé en mémoire flash

Ceci est géré par notre routine « readflash », qu'il nous suffit donc d'appeler.

```
        ; lecture de l'octet demandé en flash  
        ; -----  
  
call    readflash      ; lire l'octet demandé dans la mémoire flash
```

Maintenant, préparer l'émission de l'octet concerné est du ressort de la portion de code suivante, qui positionne d'abord le bit 9 à envoyer, puis place l'octet dans le registre TXREG, et ensuite lance l'émission. N'oubliez pas que l'émission n'aura effectivement lieu que lorsque le maître décidera d'envoyer ses 9 impulsions d'horloge.

```
        ; envoyer l'octet demandé  
        ; -----  
  
bsf     STATUS,RP0      ; passer en banque1  
bcf     TXSTA,TX9D      ; par défaut, 9eme bit à envoyer = 0  
btfsc   EMI9            ; tester si 9eme bit doit valoir 1
```

```

bsf    TXSTA,TX9D      ; oui, positionner 9eme bit
movf   octet,w         ; charger octet à envoyer
bcf    STATUS,RP0      ; repasser banque 0
movwf  TXREG           ; placer octet à envoyer dans registre d'envoi
bsf    STATUS,RP0      ; repasser banque 1
bsf    TXSTA,TXEN      ; lancer l'émission

```

Il nous faut maintenant attendre la fin de l'émission, caractérisée par le positionnement du flag TRMT. Ensuite, nous stoppons l'émission, afin de repasser en réception en sautant au début de notre programme principal :

```

                ; attendre la fin de l'émission
                ; -----
btfss   TXSTA,TRMT    ; émission terminée?
goto    $-1           ; non, attendre

                ; clôture de l'émission
                ; -----
bcf     TXSTA,TXEN     ; fin de l'émission
bcf     STATUS,RP0     ; repasser banque 0
goto    start         ; boucler

```

Il reste à écrire quelques données en mémoire flash, réparties en 2 zones distinctes :

```

;*****
;                                DATA                                *
;*****
;-----
; 2 zones contenant les données à envoyer, codées sur 9 bits
; le bit8 indique le dernier octet de la zone
;-----

```

```

zonedat1
DA B'000000001'
DA B'000000010'
DA B'000000100'
DA B'000001000'
DA B'000010000'
DA B'000100000'
DA B'001000000'
DA B'010000000'
DA B'001000000'
DA B'000100000'
DA B'000010000'
DA B'000001000'
DA B'000000100'
DA B'100000010' ; le 9ème bit vaut 1, donc fin de zone

```

```

zonedat2
DA B'000000001'
DA B'000000011'
DA B'000000111'
DA B'000001111'
DA B'000011111'
DA B'000111111'
DA B'001111111'
DA B'011111111'
DA B'001111111'
DA B'000111111'
DA B'000011111'

```

```

DA B'000001111'
DA B'000000111'
DA B'000000011'
DA B'000000001'
DA B'100000000'      ; le 9ème bit vaut 1, donc fin de zone

END                  ; directive fin de programme

```

Notez que la directive « END » doit se trouver en fin de la zone des données, et non en fin du programme principal.

Programmez les 2 PICs à l'aide des fichiers obtenus, et lancer l'alimentation. Vos LEDs effectuent 3 séquences d'aller-retour suivies par 4 séquences de remplissage. Vous avez réussi votre communication série synchrone basée sur l'USART.

24.14 Remarque sur le mode sleep

Les lecteurs particulièrement attentifs, et surtout ceux qui cherchent à anticiper ce qu'ils vont lire, n'auront pas manqué de faire la réflexion suivante :

Pourquoi diable place-t-il le PIC en mode sommeil durant l'attente de la réception du mot, et ne le fait-il pas dans l'attente de l'émission ?

Si vous êtes dans ce cas, bravo. Si en plus, vous avez la solution, alors vous êtes devenu un expert. Je m'explique donc :

Pour réveiller le PIC, il faut impérativement le positionnement d'un flag d'interruption. Or, qu'avons-nous à notre disposition concernant l'USART ?

Tout d'abord le flag RCIF que nous avons utilisé. Il permet de signaler que la réception d'un mot est terminée, et que l'octet reçu a été chargé dans le registre RCREG (avec positionnement éventuel de RX9D). C'est exactement ce qu'il nous fallait attendre. Aucun problème, donc, pour la réception.

Au niveau de l'émission, nous avons l'indicateur TXIF. Malheureusement pour notre exercice, ce flag est positionné, non pas quand la transmission est terminée, mais quand le registre TXREG est vide. Ce flag est prévu pour opérer une émission continue de plusieurs octet, et donc réveillerait le PIC dans ce cas à chaque fois qu'une place est libre dans le registre d'émission. L'arrêt de la transmission au moment du positionnement de TXIF empêcherait donc la transmission du dernier octet. Or, nous n'en avons qu'un à transmettre.

On pourrait donc penser qu'il n'y a pas de solution à notre problème si on voulait absolument forcer notre PIC à entrer en sommeil durant l'attente de la fin de l'émission. Et bien si, il y a une astuce possible.

- Vous écrivez l'octet à envoyer dans TXREG.
- Au cycle suivant, TXREG est transféré dans TSR, TXREG est vide, donc TXIF est positionné

- Vous écrivez une seconde fois l'octet dans TXREG. Comme TSR n'est pas encore vide (l'émission n'est pas terminée, et peut-être même pas commencée), TXREG n'est pas transféré, donc TXIF n'est pas positionné.
- Vous validez les interruptions d'émission par TXIE.
- Vous placez votre PIC en mode sommeil.
- Le premier octet a été transmis, TXREG est transféré dans TSR, TXREG est vide, donc TXIF est positionné
- Votre PIC se réveille, il vous suffit de couper TXEN pour mettre fin à l'émission . L'émission du second octet n'aura donc pas lieu. La coupure de TXEN provoque le reset de l'USART. Vous en profitez pour effacer le bit TXIE.

Vous voyez donc qu'en réfléchissant, on trouve des solutions simples pour tous les cas de figure. Dans le cas présent, il suffisait simplement d'écrire 2 valeurs au lieu d'une dans TXREG.

24.15 Conclusions

Nous venons d'étudier l'USART dans son fonctionnement synchrone, nul doute que vous ne trouviez nombre d'applications le concernant. En effet, l'USART est non seulement puissant et rapide, mais également très simple d'emploi.

Je termine avec une petite remarque. Faites attention de ne pas confondre, dans l'écriture de vos programmes, CREN avec RCEN. Comme ce dernier existe, MPASM ne vous signalerait pas d'erreur, mais votre programme ne pourrait pas fonctionner correctement.

C'est le genre d'erreurs de distraction difficiles à détecter du premier coup d'œil. Vous pensez bien que si je vous en parle, c'est que cela m'est déjà arrivé.

Notes : ...

25. Le module USART en mode asynchrone

25.1 Le mode série asynchrone

J'ai déjà parlé de ce mode dans la première partie du cours, au chapitre concernant la norme ISO 7816. Vous pouvez commencer par aller relire ce chapitre.

Cependant, je vais reprendre ici les caractéristiques de ce mode, agrémentées de quelques explications complémentaires. Ceci vous évitera de sauter d'un livre à l'autre.

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 à 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.

25.1.1 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, comme par exemple l'utilisation d'un driver RS485 (par exemple le MAX485), ou un pilote RS232 (comme le MAX232) qui

permettra à votre liaison série de devenir compatible électriquement avec le port RS232 de votre PC (j'y reviendrai).

Donc, notre ligne se trouve à « 1 ». Or, nous n'avons qu'une seule ligne dédiée 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 » (start-bit).

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.

25.1.2 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.

$$T_b = 1 / \text{Débit}$$

Par exemple, si on a affaire à une connexion à 9600 bauds, nous aurons :

$$T_b = 1 / 9600 \text{ bauds} = 104,16 \mu\text{s}.$$

25.1.3 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 à « 1 » à valoir « 4 », ce qui est bien pair.

On enverra donc :

B'001101001' : 4 bits à « 1 » = parité paire

25.1.4 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.

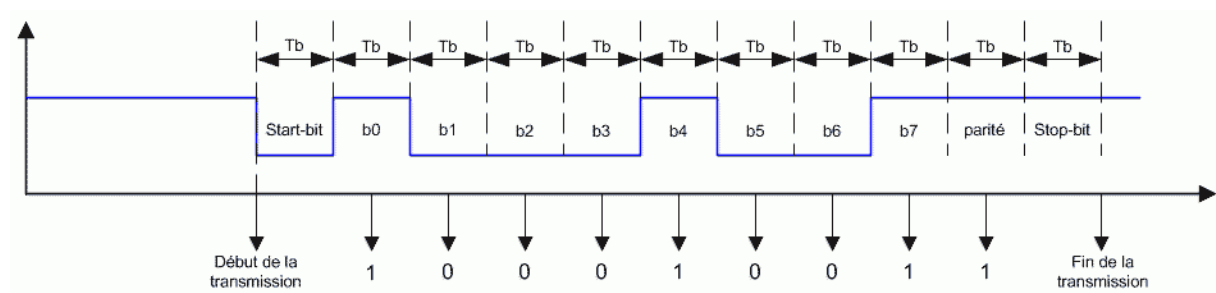
Le stop-bit n'est pas comptabilisé (pas plus que le start-bit) pour le calcul du bit de parité.

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 équivalente 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 T_b 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 votre information, que la lecture d'un bit par le PIC ne s'effectue pas en une seule fois. En réalité, votre 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.

25.1.5 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. Je vais maintenant vous montrer comment exploiter les différentes possibilités avec votre PIC en fonction de ses possibilités.

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, par la réception 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).

Comme second stop-bit ou pause entre 2 émissions sont identiques (la ligne reste à l'état haut), le récepteur n'y verra que du feu.

La réception est par contre impossible. En effet, si l'interlocuteur envoie ses données de façon jointive, le start-bit de la donnée suivant 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. En fait, je ne l'ai jamais rencontré.

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.

Fort 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 placés 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 (9^{ème} 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 donnée 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 (9^{ème} 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 9^{ème} bit, soit :

- Les 8 bits de donnée dans les bits 0 à 7
- Le bit de parité dans le bit 8 (9^{ème} bit)

C'est un grand classique que vous rencontrerez souvent (et que nous traiterons dans notre exemple).

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**. Nous nous en sortons par une pirouette en **mode donnée sur 9 bits**:

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 (9^{ème} 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.

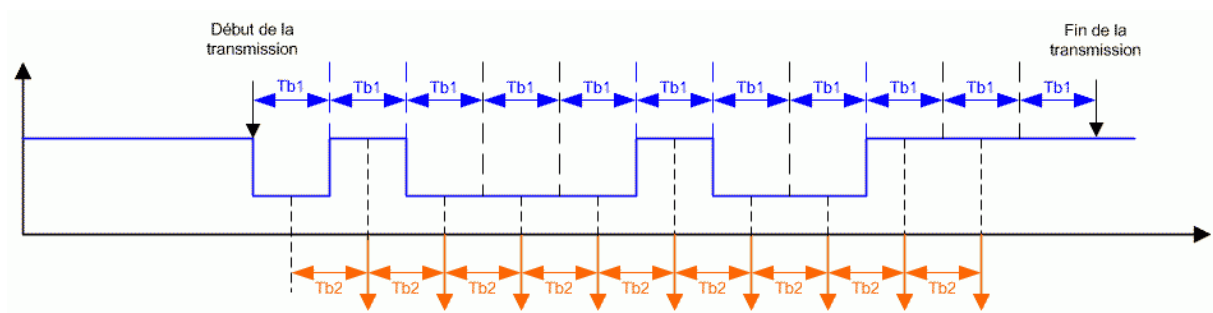
En résumé, en émission, vous créez le second stop-bit en introduisant un délai entre l'émission de 2 données.

25.1.6 Les erreurs de synchronisation

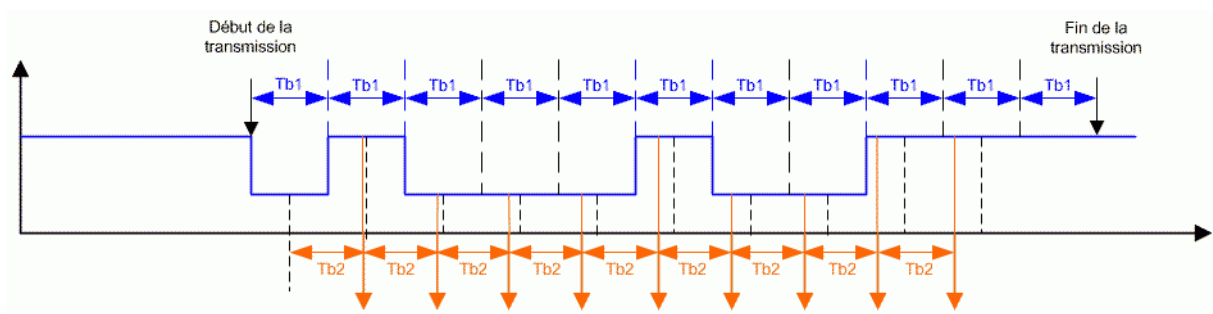
Je vous en ai déjà touché un mot, vous savez donc que émetteur et récepteur adoptent une vitesse d'horloge qui est sensée être identique sur les 2 composants. Seulement, **ces horloges ne sont jamais exactement identiques**, ce qui entraîne une **erreur qui se cumule** au fur et à mesure de la réception 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 :

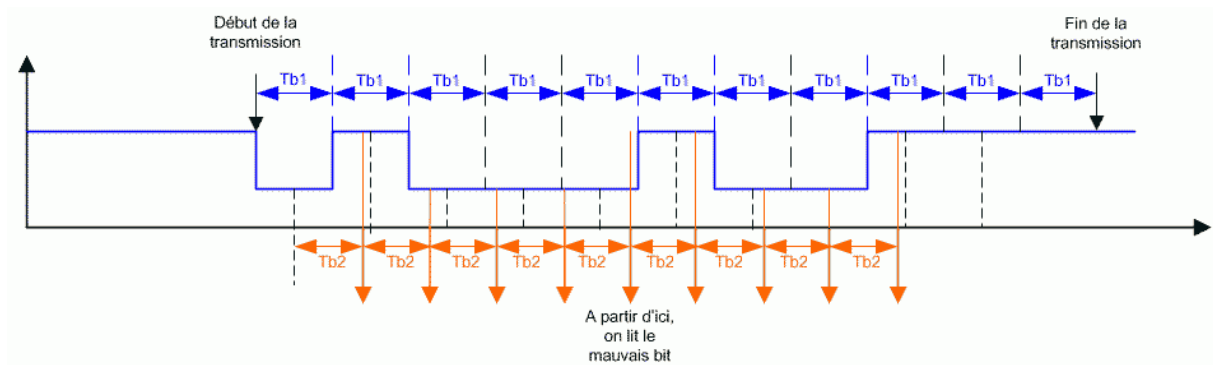


Maintenant, voyons le cas où l'horloge du récepteur n'est pas tout à fait identique. Imaginons qu'elle tourne légèrement plus vite, donc $tb2$ est inférieur à $Tb1$:



Vous voyez 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, vous voyez 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 :

- L'erreur 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 :

$$\begin{aligned} \text{Erreur} &< 50\% / 10 \\ \text{Erreur} &< 5\% \end{aligned}$$

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% me 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 :

$$\text{Erreur théorique} = | \text{Débit réel} - \text{débit idéal} | / \text{débit idéal.}$$

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 :

$$| 9615 - 9600 | / 9600 = 0,0015, \text{ soit } 0,15\%$$

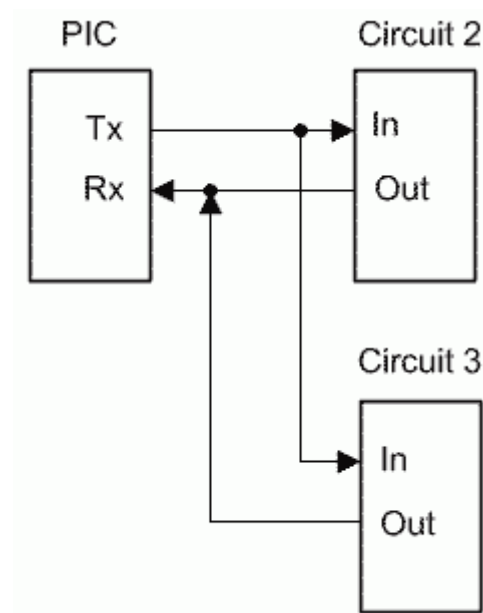
A cette erreur, pour être précis, nous devrions ajouter la dérive de votre horloge et l'erreur de votre interlocuteur.

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, presque à coup sûr, de ne pas avoir de problème.

25.2 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 :

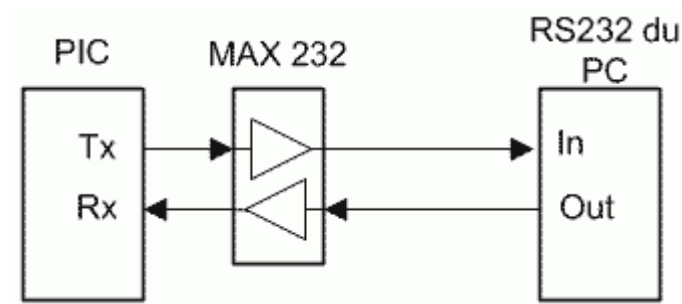


Dans ce cas, notre PIC est privilégié, puisqu'il peut parler avec le circuit 2 et 3, alors que ces circuits ne peuvent dialoguer entre eux.

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.

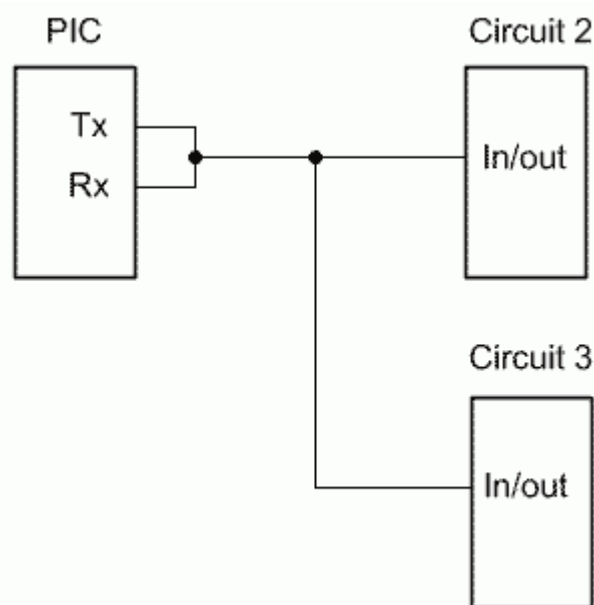
Notez que je représente toujours des liaisons directes, connectées directement sur votre PIC. On peut aussi, pour diverses raisons, dont la principale est d'allonger les distances de communication, utiliser des pilotes de lignes spécifiques.

Prenons par exemple le cas de la communication half-duplex entre votre PIC et le port série d'un PC :



Vous notez la présence d'un driver de ligne, de type MAX232 qui convertit les niveaux 0V/5V de votre PIC en niveaux +12V/-12V à destination de la RS232 de votre PC, et réciproquement. La norme RS232 est également une norme qui précise un mode de fonctionnement full-duplex.

Voyons maintenant le cas de la liaison half-duplex :

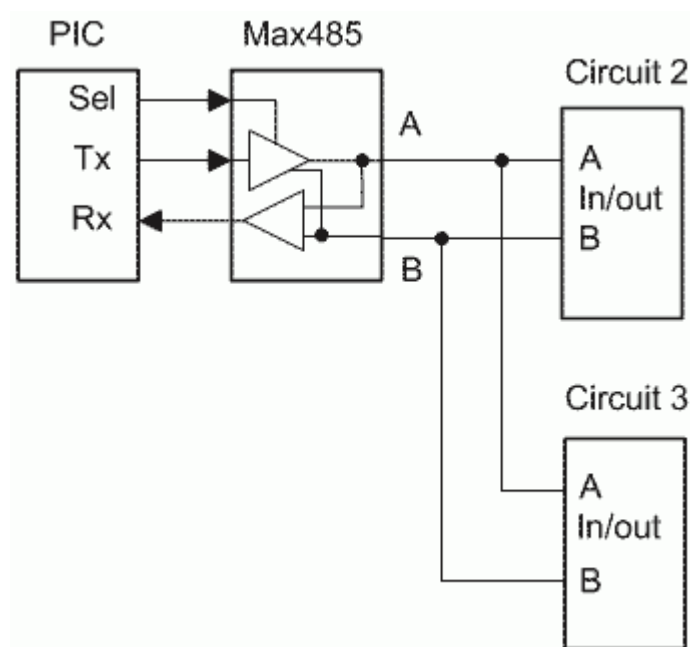


Dans ce cas, tous les composants peuvent parler entre eux. Bien entendu, il vous faudra alors définir des protocoles pour qu'un seul interlocuteur ne parle à la fois. J'ai déjà parlé de ces protocoles dans les chapitres précédents. Le revers de la médaille, c'est que votre PIC peut émettre et recevoir, 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.

25.3 le registre TXSTA

Venons-en à nos PICs, et en particulier aux registres et mécanismes mis en œuvre dans le cadre de nos liaisons série asynchrones. Commençons par le registre TXSTA.

Je vous ai déjà parlé de ce registre, c'est logique, puisque les mêmes sont utilisés pour le module USART, que ce soit en mode synchrone ou asynchrone. Je vais surtout ici vous montrer ses spécificités dans le mode asynchrone

TXSTA en mode asynchrone

b7 : non	: non utilisé en mode asynchrone
b6 : TX9	: TRANSMITT 9 bits enable bit (1 = 9 bits, 0 = 8 bits)
b5 : TXEN	: TRANSMITT ENable bit
b4 : SYNC	: SYNChronous mode select bit (1 = synchrone, 0 = asynchrone)
b3 : N.U.	: Non Utilisé : lu comme « 0 »

b2 : BRGH : Baud Rate Generator High mode select bit
 b1 : TRMT : TRANSMIT shift register status bit (1 = TSR vide)
 b0 : TX9D : TRANSMIT 9th bit Data

Voyons ou rappelons ce que signifie tout ceci :

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.

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.

SYNC indique si vous travaillez en mode synchrone (1) ou asynchrone (0). Dans ce chapitre, nous traitons le mode asynchrone, donc ce bit devra être positionné à « 0 ».

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.

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 9^{ème} bit à envoyer, quand vous avez décidé, via le positionnement de TX9, de réaliser des émissions de mots de 9 bits.

25.4 Le registre RCSTA

De nouveau, j'ai déjà parlé de ce registre, je serai donc bref.

RCSTA en mode asynchrone

b7 : SPEN : Serial Port ENable bit
 b6 : RX9 : RECEIVE 9 bits enable bit (1 = 9 bits, 0 = 8 bits)
 b5 : non : non utilisé en mode asynchrone
 b4 : CREN : Continuous Receive ENable bit
 b3 : ADDEN : ADDRESS detect ENable bit
 b2 : FERR : Frame ERROR
 b1 : OERR : Overflow ERROR bit
 b0 : RX9D : RECEIVE 9th Data bit

Quant au rôle de chacun de ces bits :

SPEN met le module USART en service.

RX9 permet de choisir une réception sur 8 ou sur 9 bits, exactement comme pour l'émission.

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.

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 suivez, on a donc un bit FERR vu comme s'il était le 10^{ème} bit du mot reçu.

OERR indique une erreur de type overflow. J'ai déjà expliqué ce bit dans le chapitre sur l'USART en mode synchrone. 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.

RX9D contient le 9^{ème} bit de votre donnée reçue, pour autant, bien entendu, que vous ayez activé le bit RX9.

25.5 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 inversement.

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)

$$\text{Débit} = \text{Fosc} / (64 * (\text{SPBRG} + 1))$$

Ou encore

$$\text{SPBRG} = (\text{Fosc} / (\text{Débit} * 64)) - 1$$

Les limites d'utilisation sont (avec notre PIC à 20MHz) :

Débit min = $20 \cdot 10^6 / (64 * (255+1)) = 1221$ bauds

Débit max = $20 \cdot 10^6 / (64 * (0+1)) = 312.500$ bauds

Si BRGH = 1 (haute vitesse)

Débit = $F_{osc} / (16 * (SPBRG + 1))$

Ou encore

$SPBRG = (F_{osc} / (\text{Débit} * 16)) - 1$

Les limites d'utilisation sont (avec notre PIC à 20MHz) :

Débit min = $20 \cdot 10^6 / (16 * (255+1)) = 4883$ bauds

Débit max = $20 \cdot 10^6 / (16 * (0+1)) = 1.250.000$ 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 par les 2 modes. Dans ce cas, vous choisirez BRGH = 1 si cela vous permet d'obtenir le taux d'erreur le moins élevé.

25.6 L'initialisation

Je vais maintenant commencer à être un peu plus concret, et, tout d'abord, voyons comment initialiser notre PIC dans ce mode par un exemple:

```
Init
    bsf    STATUS,RP0        ; passer en banque 1
    movlw  B'01000100'       ; mode asynchrone, émission 9 bits haute vitesse
    movwf  TXSTA              ; dans registre TXSTA
    movlw  BRGVAL             ; valeur calculée de SPBRG
    movwf  SPBRG              ; dans baud rate generator
    bcf    STATUS,RP0        ; repasser banque 0
    movlw  B'11000000'       ; module en service, réception 9 bits
    movwf  RCSTA              ; dans registre RCSTA
```

Il ne restera plus qu'à lancer au moment opportun émission et/ou réception en positionnant les bits TXEN et/ou CREN.

25.7 Emission, réception, et erreur d'overflow

Je ne vais pas tout recommencer ici. Les mécanismes sont exactement les mêmes que pour le mode synchrone. Je vous renvoie donc au chapitre précédent, pour vous remémorer émission et réception en mode synchrone maître. Les flags utilisés sont les mêmes. La seule différence est que maintenant vous pouvez émettre et recevoir en même temps.

Je vous livre donc simplement un petit résumé :

Pour l'émission

- Vous validez 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.
- Si vous utilisez le format de donnée sur 9 bits, vous devez commencer par placer la valeur du 9^{ème} 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. TRMT passe à « 0 » (TSR plein).
- Vous recommencez 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. Vous pouvez 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 l'ordre de cette séquence, je vous renvoie au chapitre précédent.

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 9^{ème} 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 9^{ème} 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 9^{ème} 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 9^{ème} 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 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.

25.8 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 positionnée à « 0 ». Dans ce cas, quelque chose s'est mal déroulé.

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.

25.9 Utilisation du 9^{ème} bit comme bit de parité

Nous avons vu dans le chapitre précédent, comment utiliser notre 9^{ème} bit comme un bit d'usage spécifique. Je vais vous montrer un petit exemple d'utilisation de ce bit comme bit de parité.

Comme la parité n'est pas gérée automatiquement, il nous suffit de créer une petite routine pour la générer. En voici un exemple :

```
parite
    movf   octet,w      ; charger octet à envoyer
    movwf  temp         ; sauver dans variable temporaire en RAM commune
    bsf    STATUS,RP0   ; passer en banque 1
    bsf    TXSTA,TX9D   ; bsf pour parité impaire, bcf pour parité paire
loop
    andlw  0x01         ; garder bit 0 de l'octet
    xorwf  TXSTA         ; si « 1 », inverser TX9D
    bcf    STATUS,C     ; effacer carry pour décalage
    rrf    temp,f       ; amener bit suivant en b0
    movf   temp,w       ; charger ce qui reste de temp
    btfss  STATUS,Z     ; il reste des bits à « 1 » ?
    goto   loop         ; oui, poursuivre
    bcf    STATUS,RP0   ; non, repasser banque 0
    ... ..              ; reste à charger TXREG et lancer l'émission.
```

Il ne vous reste plus qu'à réaliser la routine qui effectue le contrôle de la parité pour la réception. En fait, c'est pratiquement la même, comme vous le prouvera l'exercice de fin de chapitre.

Vous pouvez vous amuser à essayer d'optimiser cette petite routine donnée à titre d'exemple. Bien sûr il existe plein d'autres méthodes.

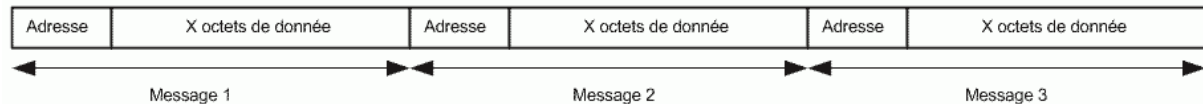
25.10 La gestion automatique des adresses

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. J'en ai déjà parlé dans les transmissions synchrones.

Imaginons, par exemple, que nous décidions d'envoyer l'adresse du destinataire, suivie par les octets de données.

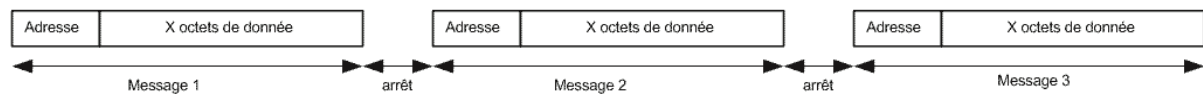
Dans ces octets de données pourra se trouver également la longueur du message, l'adresse de l'expéditeur, etc., mais tel n'est pas mon propos ici. Je m'intéresse à l'adresse du destinataire.

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 de les séparer :



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) :

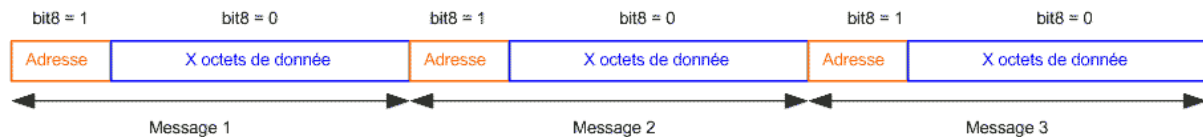
- On resette le compteur de caractères
- On signale que le message est complet et peut être traité
- Fin de l'interruption.

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 9^{ème} 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 9^{ème} 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 dédié 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 longueur 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.

25.11 Le mode sleep

Comme vous devez commencer à devenir des experts, vous avez remarqué que les horloges mises en œuvre dans les registres à décalage (RSR et TSR) sont dérivées de Tosc, donc de l'horloge principale du PIC.

Comme celle-ci est stoppée durant le mode sleep, **votre USART ne pourra fonctionner en mode asynchrone dans cet état.**

25.12 Exemple de communication série asynchrone

Nous allons maintenant mettre en pratique une liaison asynchrone. Je vais prendre comme exemple un échange d'information qui tient à cœur à beaucoup d'entre-vous, à savoir la communication entre votre PIC et votre PC.

La première chose à faire est d'examiner les ports de communication disponibles sur votre PC. Nous disposons du port série RS232 classique, du port USB, et du port parallèle. Sur ces 3 ports, seul le dernier est facilement interfaçable, car il travaille avec des niveaux compatibles avec notre PIC. Cependant, je vais utiliser le port série pour cette application.

D'une part, parce que ce port est spécifiquement conçu pour mettre en œuvre des liaisons série, et d'autre part, parce que c'est au sujet de ce port que vous me posez le plus de questions dans le courrier que je reçois. Le port USB, quant à lui, nécessite des drivers spécifiques du côté du PC, et un hardware non moins spécifique du côté du PIC, qui sortent du cadre de cet ouvrage.

Il importe en premier lieu d'analyser comment connecter physiquement notre PIC à notre port RS232. Le PIC utilise les niveaux 0V et 5V pour définir respectivement des signaux « 0 » et « 1 ». La norme RS232 définit des niveaux de +12V et -12V pour établir ces mêmes niveaux.

Nous aurons donc besoin d'un circuit (driver de bus) chargé de convertir les niveaux des signaux entre PIC et PC. La pin TX du PIC émettra en 0V/5V et sera convertie en +12V/-12V vers notre PC. La ligne RX du PIC recevra les signaux en provenance du PC, signaux qui seront converti du +12V/-12V en 0V/5V par notre circuit de pilotage du bus.

Notez que la liaison étant full-duplex, émission et réception sont croisées, chaque fil ne transitant l'information que dans un seul sens.

Nous utiliserons le célèbre circuit MAX232 pour effectuer cette adaptation de niveaux. Ce circuit contient un double convertisseur à double direction. Autrement dit, il dispose de :

- 2 blocs, dénommés T1 et T2, qui convertissent les niveaux entrés en 0V/5V en signaux sortis sous +12V/-12V. En réalité, on n'a pas tout à fait +12V et -12V, mais plutôt de l'ordre de +8,5V/-8,5V (en théorie +10V/-10V), ce qui reste dans la norme RS232. Les entrées de ces blocs sont donc dirigés vers le PIC, les sorties sont connectées sur le port RS232.
- 2 blocs, dénommés R1 et R2, qui convertissent les niveaux entrés en +12V/-12V en signaux sortis sous 0V/5V. Les entrées de ces blocs sont donc connectées sur le port RS232, les sorties sur le PIC.

Le port RS232 de votre PC contient d'autres lignes, comme CTS, DSR, etc. Il n'est pas dans mon intention de vous décrire en détail la norme RS232, ces informations sont

disponibles partout. Sachez seulement que les signaux supplémentaires sont destinés à indiquer quand on est prêt à parler, et quand on doit écouter la ligne pour recevoir des octets.

Il s'agit donc de lignes de sélection, que vous pourrez gérer, si le besoin s'en fait sentir, via des lignes I/O classiques de votre PIC. Le double convertisseur du MAX232 s'explique d'ailleurs par la nécessité de convertir, outre RX et TX, ces signaux RS232 supplémentaires.

Ce qui m'intéresse ici est l'émission et la réception proprement dites, le reste est histoire de détails.

Maintenant, nous allons avoir besoin de 2 logiciels, un du côté du PIC, et qui nous intéresse, et un du côté du PC, afin de recevoir et d'envoyer des informations.

Mon ami Caméléon a eu la grande gentillesse de réaliser spécialement pour moi (et donc pour vous) un logiciel complètement paramétrable pour vous permettre de dialoguer avec votre PIC. Ce programme se trouve dans le sous-répertoire « **BSCP** (Bigonoff Support de Cours sur les PICs) » du répertoire « fichiers ». Nous verrons en temps utile comment l'utiliser.

S'il ne vous est pas possible d'utiliser ce programme (par exemple en cas d'utilisation sous MAC ou LINUX), il vous restera à utiliser un programme de communication RS232 standard.

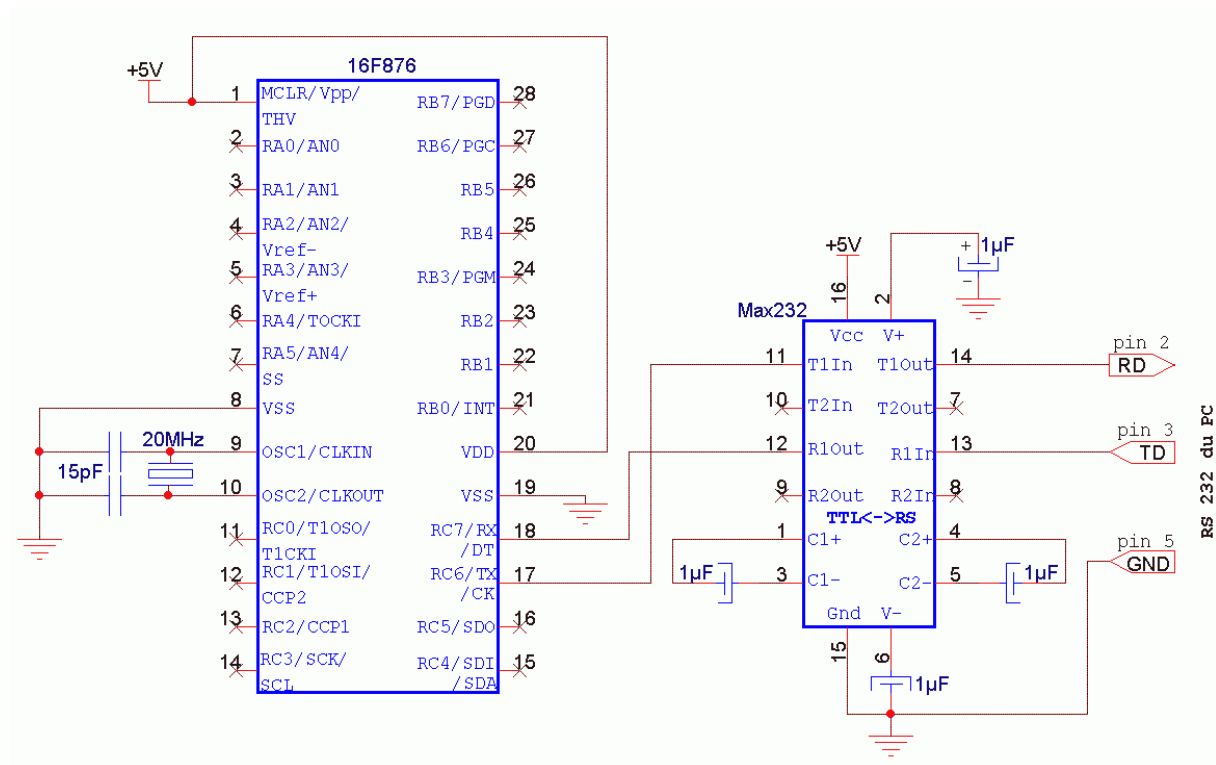
Dans ce dernier cas, si vous avez la possibilité de couper la gestion de flux matérielle dans le logiciel utilisé (CTS etc.), vous devrez le faire. Dans le cas contraire, il vous faudra interconnecter les pins concernées en suivant les spécificités des « null-modem ». Autrement dit, dans la fiche DB9 de votre câble, du côté du PC, vous connecterez ensemble les pins 4,6, et 1. Un second pontage entre les pins 7 et 8 complètera l'intervention.

La pin 3 de la fiche DB9 de votre PC est l'émission (vu par le PC), donc la réception vu du côté du PIC. Ce signal est dénommé TD (Transmitt Data).

La pin 2 est la pin de réception de votre PC. Ce signal est dénommé RD (Receive Data).

La pin 5 est la masse de votre RS232, qu'il ne faudra pas oublier de connecter à la masse de votre PIC.

Voyons le schéma obtenu :



Il n'y a pas de grandes difficultés. Vous aurez simplement besoin d'un MAX232 ou équivalent (STN232...), d'une fiche DB9 femelle, d'un bout de câble à 3 fils et de 4 condensateurs électrolytiques (chimiques).

Concernant ces derniers, leur valeur n'est pas critique (de 1 à 22 μF), mais ils sont polarisés. Vous devez donc veiller à les connecter dans le bon sens. Le repère « - » sur le condensateur correspond au grand côté du symbole sur le schéma (voir condensateur connecté sur la pin 2 du MAX232 que j'ai repéré). Certains datasheets renseignent un condensateur entre la pin 2 et la pin 16 au lieu de la masse. Pas d'inquiétude, c'est strictement similaire. Vous pouvez également ajouter un condensateur de quelques μF entre la pin 16 et la pin 15 (découplage), mais en général cela n'est pas nécessaire, quoi que de bonne pratique.

Les numéros des pins sont ceux correspondants à la fiche DB9 du PC. Si vous utilisez un câble entre votre PC et votre platine d'expérimentation, vérifiez s'il s'agit bien d'un câble d'allonge. En effet, si vous utilisez un câble null-modem, les pins 2 et 3 seront croisées, tenez-en compte au moment de raccorder. Dans le doute, contrôlez au multimètre (« Ohm-mètre »).

Passons à notre programme. Créez une copie de votre fichier « **m16F876.asm** » et renommez cette copie « **rs232.asm** ».

Je vous propose un exercice qui réalise les opérations suivantes :

- Liaison full-duplex entre votre PC et le PIC, à 9600 bauds, 8 bits de données, parité paire, 1 stop-bit
- Le PIC attend de recevoir une chaîne de caractères terminée par une pression sur la touche « return ». Cette touche génère l'envoi de 2 caractères, à savoir le « carriage-return », ou

retour de chariot, de code ASCII 0x0D, suivi « du line-feed », ou passage à la ligne, de code ASCII 0x0A.

- Le PIC répond en renvoyant la chaîne reçue, mais inversée
- Toutes les gestions d'erreur sont prises en compte.

Je vous ai créé un programme dont les fonctions sont clairement séparées, ce qui vous permettra de vous en servir comme base pour vos propres applications. Je n'ai donc pas cherché à optimiser, j'ai préféré qu'il soit le plus clair possible.

Le début de notre programme ne pose pas de difficultés. J'ai mis le watchdog en service :

```

;*****
;   Exemple de communication série asynchrone.
;   Communication avec le port RS232 d'un PC.
;   Le programme reçoit une série de caractères (maximum 94)
;   Il répond par la chaîne de caractères inversée.
;   La fin de la chaîne est détectée par le caractère "line-feed" (0x0A)
;
;*****
;
;   NOM:      RS232
;   Date:     23/07/2002
;   Version:  1.0
;   Circuit:  Platine d'expérimentation
;   Auteur:   Bigonoff
;
;*****
;
;   Fichier requis: P16F876.inc
;
;*****
;
;   La liaison s'effectue au travers d'un MAX232
;   1 start-bit, 8 bits de donnée, 1 bit de parité paire, 1 stop-bit
;   Liaison en 9600 bauds (9615 bauds réels)
;   Utilisation du module USART
;   Le PIC travaille à 20MHz
;*****

LIST      p=16F876           ; Définition de processeur
#include <p16F876.inc>        ; fichier include

__CONFIG  _CP_OFF & _DEBUG_OFF & _WRT_ENABLE_OFF & _CPD_OFF & _LVP_OFF &
_BODEN_OFF & _PWRTE_ON & _WDT_ON & _HS_OSC

;*****
;
;               ASSIGNATIONS SYSTEME
;*****

; REGISTRE OPTION_REG (configuration)
; -----
OPTIONVAL EQUB'10001111'
; RBPU      b7 : 1= Résistance rappel +5V hors service
; PSA       b3 : 1= Assignment prédiviseur sur Watchdog
; PS2/PS0   b2/b0 valeur du prédiviseur = 128

```

Nous pouvons penser être amenés à recevoir ou à émettre une centaine de caractères simultanément. Donc, nous risquons d’être bloqués dans notre routine d’interruption de l’ordre de $100 \text{ octets} * 11 \text{ bits/octet} / (9600 \text{ bits/seconde}) = 114 \text{ ms}$. Or, notre watchdog risque bien de déborder après 7ms. Je mettrai donc le prédiviseur sur le watchdog, avec une valeur confortable de minimum 896 ms.

En réalité, cela ne surviendra pas, car la réception d’un octet prendra $11/9600 = 1,15 \text{ ms}$. Nous avons donc le temps d’exécuter 5700 instructions entre la réception de 2 octets consécutifs. Inutile de dire que nous avons le temps de sortir de notre routine d’interruption et de resetter notre watchdog.

Il nous faut maintenant calculer la valeur que nous devons placer dans notre registre SPBRG, en fonction de la fréquence de notre quartz et du débit choisi ? Nous travaillerons en mode haute-vitesse si c’est possible, ce que nous allons voir de suite :

$$\text{SPBRG} = (\text{Fosc} / (\text{Débit} * 16)) - 1$$

$$\text{SPBRG} = (20 * 10^6 / (9600 * 16)) - 1 = 129,2$$

Nous prendrons la valeur D’129’, ce qui nous donne une fréquence réelle de :

$$\text{Débit} = \text{Fosc} / (16 * (\text{SPBRG} + 1))$$

$$\text{Débit} = 20 * 10^6 / (16 * 130) = 9615,4 \text{ bauds}$$

L’erreur sera de :

$$\text{Err} = (9615,4 - 9600) / 9600 = 0,16\%$$

Ceci ne nous posera donc pas le moindre problème. Autrement dit :

```
; *****
;                                     ASSIGNATIONS PROGRAMME                      *
; *****

BRGVAL    EQU    D'129'; pour un débit de 9615 bauds en mode high-speed
```

Nous aurons besoin d’écrire quelques messages courts (4 caractères) à destination de notre PC. Nous créons donc une petite macro qui place 4 caractères dans notre buffer de sortie (bufout), et qui complète le message par les 2 caractères définis comme fin de message, à savoir « carriage-return » + « line-feed ». Le message sera entouré par des « [] », afin de bien le distinguer

```
; *****
;                                     MACRO                                          *
; *****

MESS      macro a1,a2,a3,a4 ; inscrit le message dans bufout
    BANKSEL    bufout      ; passer en banque 3
    movlw     "["          ; mettre crochet
    movwf     bufout        ; dans bufout
    movlw     a1            ; charger argument 1
    movwf     bufout+1      ; dans bufout
```

```

movlw a2          ; charger argument 2
movwf bufout+2    ; dans bufout
movlw a3          ; charger argument 3
movwf bufout+3    ; dans bufout
movlw a4          ; charger argument 4
movwf bufout+4    ; dans bufout
movlw "]"         ; mettre crochet
movwf bufout+5    ; dans bufout
movlw 0x0D        ; charger carriage return
movwf bufout+6    ; dans bufout
movlw 0x0A        ; charger line-feed
movwf bufout+7    ; dans bufout
BANKSEL 0         ; repasser banque 0
Endm

```

J'ai, comme tout le monde, créé mes variables au fur et à mesure de la réalisation du programme. Je vous les livre en une fois, nous verrons leur rôle au fur et à mesure. J'ai fait bon usage de la banque commune, ce qui m'a évité pas mal de changements de banques, c'est une démarche logique.

```

;*****
;                               VARIABLES BANQUE 0                               *
;*****

; Zone de 80 bytes
; -----

CBLOCK 0x20        ; Début de la zone (0x20 à 0x6F)
ptr1 : 1           ; pointeur temporaire 1
ptr2 : 1           ; pointeur temporaire 2
octemp : 1         ; sauvegarde temporaire
ENDC              ; Fin de la zone

;*****
;                               VARIABLES ZONE COMMUNE                               *
;*****

; Zone de 16 bytes
; -----

CBLOCK 0x70        ; Début de la zone (0x70 à 0x7F)
w_temp : 1         ; Sauvegarde registre W
status_temp : 1    ; sauvegarde registre STATUS
FSR_temp : 1       ; sauvegarde FSR (si indirect en interrupt)
PCLATH_temp : 1    ; sauvegarde PCLATH (si prog>2K)
bufinptr : 1       ; pointeur sur caractère courant buffer entrée
bufoutptr : 1      ; pointeur sur caractère courant buffer sortie
flags : 1          ; 8 flags divers
                  ; b0 : parité calculée
                  ; b1 : erreur de parité
                  ; b2 : erreur de frame
                  ; b3 : erreur overflow

local1 : 1         ; variable locale pour interruptions
ENDC

#define PARITE      flags,0 ; parité calculée
#define ER_PAR      flags,1 ; erreur de parité
#define ER_FR       flags,2 ; erreur de frame
#define ER_OV       flags,3 ; erreur d'overflow

```

J'ai posé que nous pouvons recevoir 94 caractères maximum avant de recevoir les « carriage-return » + « line-feed » de fin de message. Donc, nous allons devoir stocker les données reçues en vue de leur traitement.

De même, pour respecter le mode full-duplex, j'ai admis qu'on pouvait continuer à recevoir des caractères durant l'émission de la réponse précédente. Donc, nous allons avoir besoin d'une autre zone de stockage des caractères à envoyer.

J'ai donc placé mon buffer d'entrée (bufin) dans la RAM en banque 3, et le buffer de sortie (bufout) dans la RAM en banque 4.

Ceci est impératif, puisque j'ai choisi de mémoriser 94 + 2 caractères. Mais, si j'avais choisi, par exemple, de mémoriser 80 octets, j'aurais choisi les mêmes banques. J'explique pourquoi :

Puisque nous avons un buffer d'entrée et un buffer de sortie, il est logique que notre programme aura besoin d'une routine de traitement, qui va écrire les données dans le buffer de sortie en fonction de ce qu'il a reçu dans le buffer d'entrée.

Autrement dit, il va devoir lire les caractères dans bufin et en écrire dans bufout. Il va donc devoir passer sans arrêt, au sein de cette routine, de bufin à bufout, ces 2 buffers étant dans des banques différentes (pour cause de place disponible).

Il y a 2 méthodes pour accéder à une variable en RAM :

- Par adressage direct
- Par adressage indirect.

On suppose que dans votre programme réel, la banque 0 sera fortement utilisée, donc ne pourra pas contenir un des buffers. Imaginons alors bufin en banque 1 et bufout en banque 2, on aura un traitement du style :

boucle

```
bsf    STATUS,RP0    ; passer banque 1
bcf    STATUS,RP1
..      ; lire octet dans bufin et traiter
bcf    STATUS,RP0    ; passer banque 2
bsf    STATUS,RP1
..      ; écrire octet dans bufout
goto   boucle        ; traiter octet suivant.
```

Vous constatez que vous avez **2 bits à positionner** pour chaque changement de banques. Si, par contre, vous utilisez l'adressage indirect :

boucle

```
bcf    STATUS,IRP    ; pointer banques 0 et 1
..      ; lire octet dans bufin et traiter
bsf    STATUS,IRP    ; pointeur banques 2 et 3
..      ; écrire octet dans bufout
goto   boucle        ; traiter octet suivant.
```

Vous devrez changer IRP, utilisé pour l'adressage indirect, à chaque changement de banques.

Vous voyez alors tout de suite, que si vous choisissez les banques 2 et 3 pour vos buffers, le passage en adressage direct ne nécessitera que la modification de RP0 (RP1 restant positionné à 1).

Le passage en adressage direct ne nécessitera plus de modifier IRP, vous pourrez le laisser à «1 » en permanence dans cette routine.

Quand vous avez de grosses zones de données à manipuler, posez-vous la question de savoir s'il n'y a pas moyen de limiter le nombre d'instructions de changement de banques dans vos boucles de traitement. Bien entendu, cela n'a pas grande importance ici, mais c'est pour la beauté du geste, et pour les bonnes habitudes.

Voici donc nos buffers :

```
; *****
;
;          VARIABLES BANQUE 2
; *****

; Zone de 96 bytes
; -----

      CBLOCK    0x110      ; Début de la zone (0x110 à 0x16F)
      bufin : D'96'      ; zone de stockage des données entrées

      ENDC              ; Fin de la zone

; *****
;
;          VARIABLES BANQUE 3
; *****

; Zone de 96 bytes
; -----

      CBLOCK    0x190      ; Début de la zone (0x190 à 0x1EF)
      bufout : D'96'      ; message à envoyer

      ENDC              ; Fin de la zone
```

J'ai choisi de traiter l'émission et la réception USART en utilisant les interruptions. C'est une démarche logique, puisque réception et émission pourront être simultanées. Je conserve donc ces 2 interruptions.

Notez que si vous conservez la structure classique de la détection des interruptions, vous avez un algorithme de ce type :

- Si RCIF est positionné,
- On traite l'interruption réception USART **et on termine l'interruption**
- Sinon, si TXIF est positionné,
- On traite l'interruption émission USART **et on termine l'interruption**

Vous voyez alors que si on reçoit une grande série de caractères, la réception a priorité sur l'émission, et donc, tant qu'on reçoit on n'émet plus. Nous allons donc modifier ceci en :

- Si RCIF est positionné,
- On traite l'interruption réception USART
- Ensuite, si TXIF est positionné,
- On traite l'interruption émission USART et on termine l'interruption

Lors du même passage dans la routine d'interruption, nous pourrions traiter à la fois l'émission et la réception d'un caractère. La réception ne bloque plus l'émission. Pour réaliser ceci, il nous suffit de supprimer les lignes « goto restoreg » présentes dans le fichier maquette.

Par contre, comme durant le déroulement du programme, nous aurons besoin de couper par moment les interruptions réception et/ou émission, nous devons conserver les tests sur RCIF et TXIF. En effet, nous pourrions avoir une interruption provoquée, par, par exemple, RCIF, alors que TXIF est coupé et TXIF positionné. Il ne faudrait pas alors que notre programme « pense » qu'une interruption sur TXIF a eu lieu.

Tout ceci nous donne :

```
;*****
;
;                               DEMARRAGE SUR RESET                               *
;*****

    org 0x000          ; Adresse de départ après reset
    goto    init       ; Initialiser

; //////////////////////////////////////
;                               I N T E R R U P T I O N S                               *
; //////////////////////////////////////

;*****
;
;                               ROUTINE INTERRUPTION                               *
;*****
;-----
; La suppression des lignes "goto restoreg" permet dans ce cas de traiter
; l'interruption sur réception et sur émission en une seule fois
;-----

;sauvegarder registres
;-----
    org 0x004          ; adresse d'interruption
    movwf    w_temp     ; sauver registre W
    swapf    STATUS,w    ; swap status avec résultat dans w
    movwf    status_temp ; sauver status swappé
    movf     FSR , w     ; charger FSR
    movwf    FSR_temp    ; sauvegarder FSR

; switch vers différentes interrupts
;-----
;                               ; Interruption réception USART
;                               ; -----

BANKSEL    PIE1          ; sélectionner banque 1
btfss     PIE1,RCIF      ; tester si interrupt autorisée
goto      intsw1         ; non sauter
bcf       STATUS,RP0     ; oui, sélectionner banque0
btfss     PIR1,RCIF      ; oui, tester si interrupt en cours
goto      intsw1         ; non sauter
call      intrc          ; oui, traiter interrupt
```

```

; Interruption transmission USART
; -----
intsw1
    bsf     STATUS,RP0      ; sélectionner banque1
    btfss   PIE1,TXIE       ; tester si interrupt autorisée
    goto    restorereg      ; non sauter
    bcf     STATUS,RP0      ; oui, sélectionner banque0
    btfss   PIR1,TXIF       ; oui, tester si interrupt en cours
    goto    restorereg      ; non sauter
    call    inttx           ; oui, traiter interrupt

; restaurer registres
; -----
restorereg
    movf    FSR_temp,w      ; charger FSR sauvé
    movwf   FSR             ; restaurer FSR
    swapf   status_temp,w   ; swap ancien status, résultat dans w
    movwf   STATUS         ; restaurer status
    swapf   w_temp,f        ; Inversion L et H de l'ancien W
    swapf   w_temp,w        ; Réinversion de L et H dans W
    retfie                 ; return from interrupt

```

Bon, tout ceci est de l'enfantillage, vu votre niveau actuel. Passons à plus corsé, la routine d'interruption de réception d'un caractère :

Nous allons construire un pseudo-code pour nous y retrouver :

- On teste si on a une erreur de frame (souvenez-vous qu'on doit lire FERR avant l'octet reçu). Si oui, on positionne un flag
- On lit la parité, et on la mémorise dans un flag (même remarque avec RX9D)
- On lit l'octet reçu, on le mémorise dans le buffer d'entrée.
- On vérifie si on est en fin de buffer. Si oui :
 - On remplace les 2 derniers caractères par « carriage-return » et « line-feed »
 - On stoppe l'interruption de réception, le temps que le buffer soit traité
- On vérifie la parité du caractère reçu. Si mauvaise, on positionne un flag
- On teste si le caractère reçu est le line-feed, dans ce cas, on stoppe l'interruption de réception le temps que le buffer soit traité.
- On regarde si RCIF est toujours positionné. Dans ce cas, il y a un autre caractère dans la FIFO, on sort de l'interruption, on y rentrera alors de suite pour traiter ce caractère
- Si RCIF n'est plus positionné, on vérifie si on n'a pas une erreur d'overflow (n'oublions pas que nous coupons par moment notre interruption réception). Dans ce cas,
 - On stoppe et on remet en service la réception
 - On positionne un flag d'erreur

Vous allez peut-être vous demander pourquoi, en fin de message, je stoppe l'interruption de réception. En fait, nous recevons un message, celui-ci est terminé. Les caractères qui suivent éventuellement font partie d'un autre message, donc doivent être mémorisés de nouveau en début de notre buffer d'entrée.

Il nous faut cependant avoir fini de traiter le message reçu (dans notre programme principal), avant d'autoriser le message suivant à « écraser » celui en cours. Ceci explique cet arrêt des interruptions.

Vous pourriez penser qu'on aurait pu stopper la réception via CREN, mais, dans ce cas, les caractères reçus durant le temps de traitement seraient perdus, alors qu'en supprimant les interruptions, nous pouvons mémoriser 2 octets dans la file FIFO avant de perdre un octet par erreur d'overflow. La réception continue donc, les octets ne seront simplement pas traités immédiatement. A nous de construire notre programme pour que nous ne perdions pas d'octets. Ceci dépend énormément du fonctionnement de l'application dans son ensemble.

Voyons ce que tout ceci nous donne :

```

;*****
;                               INTERRUPTION RECEPTION USART                               *
;*****
;-----
; Reçoit le caractère de l'USART et le place dans le buffer d'entrée.
; Si la longueur atteint D'96', on n'encode plus, et on place 0x0D en avant-
; dernière position et 0x0A en dernière position
; Si la réception est terminée (longueur atteinte ou 0x0A reçu), on stoppe les
; interruptions de réception et on repositionne le pointeur au début du buffer
; Les erreurs sont détectées et signalées
;-----
intrc
        ; tester si erreur de frame
        ; -----
        btfsc RCSTA,FERR      ; tester si erreur de frame
        bsf   ER_FR          ; oui, signaler erreur de frame

        ; lire la parité
        ; -----
        bcf   PARITE          ; par défaut, parité = 0
        btfsc RCSTA,RX9D      ; parité lue = 1?
        bsf   PARITE          ; oui, le signaler

        ; lire octet reçu
        ; -----
        bsf   STATUS,IRP      ; pointer banques 2 et 3 en indirect
        movf  bufinptr,w      ; charger pointeur destination
        movwf FSR             ; dans pointeur d'adresse
        movf  RCREG,w         ; charger octet reçu
        movwf INDF            ; sauver dans buffer

        ; vérifier la parité
        ; -----
intrcl
        movf  INDF,w          ; charger caractère reçu
        call  calcpair        ; calculer la parité
        movf  RCSTA,w         ; charger registre commande
        xorwf flags,w         ; comparer parité reçue et calculée
        andlw 0x01            ; ne garder que le résultat
        btfss STATUS,Z        ; b0 = b1? (parité calculée = parité reçue?)
        bsf   ER_PAR          ; non, signaler erreur de parité

        ; tester si erreur d'overflow
        ; -----
        btfsc PIR1,RCIF       ; encore d'autres octets dans RCREG?
        goto  intrc2          ; oui, vérifier caractère
        btfss RCSTA,OERR       ; non, erreur d'overflow?
        goto  intrc2          ; non, vérifier caractère
        bcf   RCSTA,CREN       ; oui, arrêt de la réception (reset de OERR)

```

```

    bsf    RSTA,CREN        ; remise en service de la réception
    bsf    ER_OV           ; signaler erreur overflow
    goto   intrcend         ; et fin d'interruption

    ; tester si caractère reçu = 0x0A
    ; -----
intrc2
    movf   INDF,w           ; charger caractère reçu
    xorlw  0x0A             ; comparer avec line-feed
    btfsc  STATUS,Z         ; identique?
    goto   intrcend         ; oui, fin de message

    ; vérifier si buffer plein
    ; -----
    incf   bufinptr,f       ; incrémenter pointeur de caractères
    movf   FSR,w            ; charger pointeur
    xorlw  0x6D             ; comparer avec dernier emplacement possible
    btfss  STATUS,Z         ; identique?
    return ; non, fin de réception
    incf   FSR,f            ; pointeur sur emplacement suivant
    movlw  0x0D             ; mettre carriage-return
    movwf  INDF             ; en avant-dernière position
    incf   FSR,f            ; pointer sur dernière position
    movlw  0x0A             ; charger line-feed
    movwf  INDF             ; en dernière position

    ; fin de message
    ; -----
intrcend
    movlw  LOW bufin        ; oui, adresse de départ du buffer d'entrée
    movwf  bufinptr         ; prochain caractère sera le premier
    bsf    STATUS,RP0       ; passer banque 1
    bcf    PIE1,RCIE        ; fin des interruptions de réception
    bcf    STATUS,RP0       ; repasser banque 0
    return ; et retour

```

Vous constatez que cette sous-routine d'interruption est assez touffue. J'ai géré toutes les erreurs possibles, leur traitement sera assuré par le programme principal (dans cet exemple, je me contenterai d'afficher un message d'erreur).

Si vous examinez cette routine, vous vérifierez que l'octet présent dans RCREG est toujours lu, comme je vous l'ai expliqué. La détection de l'erreur d'overflow s'effectue lorsque tous les octets valides ont été lus, ainsi on perd le minimum d'informations.

La détection de la fin d'un message stoppe l'interruption de la réception. Celle-ci sera remise en service dès que le message reçu aura été pris en compte.

Voyons maintenant notre routine d'interruption concernant l'émission. Celle-ci se contente de placer les octets présents dans bufout vers le registre TXREG. Dès que TXREG est vide, l'interruption est générée pour permettre le placement de l'octet suivant.

Quand le dernier octet a été placé, on interdit toute nouvelle interruption de l'émission. Cette interruption sera remise en service dès qu'on aura un nouveau message à envoyer.

```

; *****
;                                     INTERRUPTION EMISSION USART *
; *****

```

```

;-----
; envoie le caractère pointé par bufoutptr, puis incrémente le pointeur
; Si ce caractère est le "line-feed", alors on arrête l'émission et on pointe de
; nouveau au début du buffer
;-----
inttx
                                ; Charger octet à envoyer
                                ; -----
    bsf     STATUS,IRP          ; pointer banques 3 et 4
    movf    bufoutptr,w         ; charger pointeur d'octets
    movwf   FSR                 ; dans pointeur
    movf    INDF,w              ; charger octet à envoyer

                                ; calculer parité
                                ; -----
    call    calcpar             ; calculer parité
    bsf     STATUS,RP0          ; passer banque1
    bcf     TXSTA,TX9D          ; par défaut, parité = 0
    btfsc   PARITE              ; parité = 1?
    bsf     TXSTA,TX9D          ; oui, positionner parité

                                ; envoyer caractère
                                ; -----
    movf    INDF,w              ; charger octet à envoyer
    bcf     STATUS,RP0          ; passer banque 0
    movwf   TXREG               ; envoyer octet + parité
    incf    bufoutptr,f         ; pointer sur octet suivant

                                ; tester si fin de message
                                ; -----
    xorlw   0x0A                ; comparer octet envoyé avec Line-feed
    btfss   STATUS,Z            ; égalité?
    return  ; non, retour d'interruption

                                ; traiter fin d'émission
                                ; -----
    bsf     STATUS,RP0          ; passer en banque 1
    bcf     PIE1,TXIE           ; fin des interruptions émission USART
    bcf     STATUS,RP0          ; repasser banque 0
    movlw   LOW bufout          ; adresse du buffer de sortie
    movwf   bufoutptr           ; prochain caractère = premier
    return  ; fin d'interruption

```

Cette routine ne comporte aucune difficulté. De nouveau, la fin de l'émission est réalisée en stoppant l'interruption d'émission. Autrement dit, on « coupe » l'approvisionnement de l'émetteur, mais on le laisse expédier les octets déjà chargés. Si on avait coupé TXEN, on aurait perdu l'octet en cours de transmission, et celui préchargé dans RCREG.

Nous arrivons maintenant dans notre routine de calcul de la parité, routine appelée par nos 2 précédentes routines d'interruption.

Elle est directement tirée des explications de la partie théorique, je l'ai simplement modifiée pour travailler sur un flag, flag qui sera utilisé par nos routines appelantes :

```

;*****
;                                CALCULER LA PARITE                                *
;*****
;-----
; L'octet dont on calcule la parité est dans W

```

```

; La parité est paire
; le flag PARITE est positionné suivant la valeur calculée
;-----
calcpair
    movwf local1        ; sauver dans variable locale (temporaire)
    bcf    PARITE        ; effacer bit de parité

calcpair1
    andlw  0x01          ; garder bit 0 de l'octet
    xorwf  flags,f       ; si " 1 ", inverser parité
    bcf    STATUS,C      ; effacer carry pour décalage
    rrf    local1,f      ; amener bit suivant en b0
    movf   local1,w      ; charger ce qui reste de l'octet
    btfss  STATUS,Z      ; il reste des bits à " 1 " ?
    goto   calcpair      ; oui, poursuivre
    return              ; non, retour

```

La routine d'initialisation ne pose aucun problème particulier. Nous nous contentons d'initialiser l'USART comme vu précédemment. Nous préparons le buffer de sortie avec le message « [PRET] » pour indiquer que tout se passe comme prévu.

```

; //////////////////////////////////////
;                               P R O G R A M M E
; //////////////////////////////////////

;*****
;                               INITIALISATIONS
;*****
init
    ; Registre d'options (banque 1)
    ; -----
    BANKSEL  OPTION_REG    ; sélectionner banque 1
    movlw   OPTIONVAL      ; charger masque
    movwf   OPTION_REG     ; initialiser registre option

    ; registres interruptions (banque 1)
    ; -----
    bsf     INTCON,PEIE     ; autoriser interruptions périphériques

    ; initialiser USART
    ; -----
    movlw   B'01000100'    ; émission sur 9 bits, mode haute vitesse
    movwf   TXSTA          ; dans registre de contrôle
    movlw   BRGVAL         ; valeur pour baud rate generator
    movwf   SPBRG          ; dans SPBRG
    bcf     STATUS,RP0     ; passer banque 0
    movlw   B'11000000'    ; module USART en service, réception 9 bits
    movwf   RCSTA          ; dans registre de contrôle

    ; Initialiser message de bienvenue
    ; -----
    MESS    "P","R","E","T" ; inscrire PRET dans le buffer de sortie

    ; initialiser variables
    ; -----
    clrf    flags          ; effacer flags
    movlw   LOW bufin      ; adresse du buffer de réception
    movwf   bufinptr       ; dans pointeur
    movlw   LOW bufout     ; adresse basse du buffer d'émission
    movwf   bufoutptr      ; dans pointeur

```

```

; autoriser interruptions (banque 0)
; -----

bsf    INTCON,GIE      ; valider interruptions
goto   start           ; programme principal

```

Vous remarquerez sans doute que lorsque je charge l'adresse du buffer, je fais précéder l'adresse du buffer par la directive « LOW ». Ceci est logique, souvenez-vous que bufin est en banque 2 et bufout en banque 3. Leur adresse est donc codée sur 9 bits.

Autrement dit, la ligne :

```
movlw bufin      ; adresse du buffer de réception
```

serait traduite par l'assembleur en :

```
movlw 0x110      ; adresse du buffer de réception
```

On ne peut stocker 9 bits dans un registre. Le 9^{ème} bit sera placé, puisque nous utilisons l'adressage indirect, dans le bit IRP. Nous devons donc écrire 0x10 dans notre pointeur, et ignorer le «1 » du 9^{ème} bit. Ceci s'effectue en utilisant la directive « LOW ».

Notez que si vous ne le faites pas, MPASM le fera automatiquement pour vous, et vous gratifiera d'un « warning » du style :

Warning[202] _chemin\RS232.ASM numéro_de_ligne : Argument out of range. Least significant bits used.

L'assembleur vous indique que la donnée est trop grande pour être stockée, et que seuls les 8 bits de poids faibles seront conservés. Ca tombe bien, c'est justement ce dont on a besoin. Mais, comme je n'aime pas les warnings inutiles, j'utilise la directive « LOW », comme expliqué.

Nous allons maintenant écrire la routine qui permet de lire le buffer d'entrée, et de le recopier à l'envers dans notre buffer de sortie. Il ne faut pas oublier que tout message (d'après nos conventions) doit se terminer par 0x0D,0x0A. Nous devons donc copier l'intégralité du buffer d'entrée en commençant par la fin, **mais sans copier les 2 derniers caractères (0x0D,0x0A)**. Nous devons par contre ajouter ces 2 caractères en fin de buffer de sortie.

Si l'utilisateur a entré une chaîne vide (uniquement la touche « return »), le message ne contiendra que les 2 derniers caractères de message. Afin de visualiser ceci, nous enverrons dans ce cas le message « [VIDE] ».

Pour pouvoir copier à l'envers, on commence par rechercher la fin du message, et on recopie dans l'ordre inverse dans le buffer de sortie :

```

; *****
;                                     PREPARER LA REPONSE                                     *
; *****
; -----
; copie le buffer d'entrée inversé dans le buffer de sortie
; en modifiant cette sous-routine, vous pouvez changer le fonctionnement

```

```

; de votre programme. Vous pouvez inventer ici de nouvelles règles de
; réponse
;-----
preprep
    ; rechercher la fin du message
    ; -----
    movlw LOW bufin-1    ; pointer sur début du message-1
    movwf FSR            ; dans pointeur
    bsf    STATUS,IRP    ; pointer banques 3 et 4 en indirect
prepl
    incf   FSR,f          ; pointer sur suivant
    movf   INDF,w         ; charger caractère
    xorlw  0x0A           ; comparer avec line-feed
    btfss  STATUS,Z       ; tester si car = 0x0A
    goto   prepl          ; non, suivant
    decf   FSR,w          ; oui, prendre pointeur trouvé-1
    movwf  ptr1           ; sauver dans pointeur temporaire1

    ; traiter ligne vide
    ; -----
    xorlw  0x10           ; tester position pointeur
    btfss  STATUS,Z       ; ligne vide?
    goto   prep3          ; non, sauter
    MESS   "V","I","D","E" ; inscrire "[vide]" dans le buffer de sortie
    return                ; fin du traitement

    ; copier message inversé
    ; -----
prep3
    movlw  LOW bufout-1   ; adresse du buffer d'émission -1
    movwf  ptr2           ; dans pointeur temporaire2
prep2
    decf   ptr1,f         ; pointer sur source suivante
    incf   ptr2,f         ; pointer sur destination suivante
    movf   ptr1,w         ; charger pointeur source
    movwf  FSR            ; dans pointeur d'adresse
    movf   INDF,w         ; charger octet
    movwf  octemp         ; le sauver
    movf   ptr2,w         ; charger pointeur destination
    movwf  FSR            ; dans pointeur d'adresse
    movf   octemp,w       ; prendre octet source
    movwf  INDF           ; dans destination
    movf   ptr1,w         ; charger pointeur source
    xorlw  0x10           ; comparer avec première position
    btfss  STATUS,Z       ; on a tout copié?
    goto   prep2          ; non, suivant

    incf   ptr2,w         ; charger position octet suivant bufout
    movwf  FSR            ; dans pointeur d'adresses
    movlw  0x0D           ; charger carriage-return
    movwf  INDF           ; dans buffer de sortie
    incf   FSR,f         ; pointer sur suivant
    movlw  0x0A           ; charger line-feed
    movwf  INDF           ; dans buffer de sortie
    return                ; et fin

```

Il ne reste plus que notre programme principal. Ce dernier va séquencer les différentes étapes de la transmission. Son pseudo-code sera du type :

- On lance l'émission pour permettre l'affichage du message « [PRET] »

- On attend que le message soit complètement expédié
- On met la réception en service
- **Début de la boucle**
- On attend d'avoir reçu la fin de réception du message (0x0A)
- On attend la fin éventuelle de l'émission du message précédent
- Si on a reçu une erreur,
- On écrit le message d'erreur dans le buffer de sortie
- On autorise les interruptions émission pour envoyer le message d'erreur
- **On retourne en début de boucle**
- Sinon,
- On recopie le message à l'envers
- On autorise les interruptions réception pour recevoir les données suivantes
- On autorise les interruptions émission pour envoyer le message inversé
- **On retourne en début de boucle**
- **Fin de la boucle**

On pouvait simplifier (par exemple pour l'émission du message « PRET », mais j'ai préféré laisser le tout sous une forme qui permette d'identifier clairement les différentes étapes. Ceci nous donne :

```
;*****
;
;          PROGRAMME PRINCIPAL
;*****

start
    ; lancer l'émission, message de bienvenue
    ; -----

    bsf    STATUS,RP0      ; passer en banque 1
    bsf    TXSTA,TXEN      ; émission en service.
    bsf    PIE1,TXIE      ; envoyer message "pret"

    ; attendre fin de l'émission
    ; -----
    clrwdt                ; effacer watch dog
    btfsc  PIE1,TXIE      ; reste des caractères à envoyer?
    goto   $-2             ; oui, attendre
    btfss  TXSTA,TRMT      ; buffer de sortie vide?
    goto   $-4             ; non, attendre

    ; lancer la réception
    ; -----
    bcf    STATUS,RP0      ; passer banque 0
    bsf    RCSTA,CREN      ; lancer la réception

    ; recevoir le message
    ; -----
    bsf    STATUS,RP0      ; passer banque 1
loop
    bsf    PIE1,RCIE      ; autoriser interruption réception
    clrwdt                ; effacer watchdog
    btfsc  PIE1,RCIE      ; tester si message complet reçu
    goto   $-2             ; non, attendre

    ; Attendre fin émission précédente
```

```

; -----
clrwdt          ; effacer watchdog
btfsc PIE1, TXIE ; message précédent envoyé?
goto $-2        ; non, attendre
btfss TXSTA, TRMT ; buffer de sortie vide?
goto $-4        ; non, attendre

; traiter erreurs
; -----
movf flags, w   ; charger flags
andlw B'00001110' ; conserver flags d'erreur
btfsc STATUS, Z ; tester si au moins une erreur
goto messprep   ; non, traitement normal

btfss ER_PAR    ; tester si erreur de parité
goto err2       ; non, sauter
MESS "P", "A", "R", " " ; écrire "PAR" dans le buffer de sortie
bcf ER_PAR      ; acquitter l'erreur
err2
btfss ER_FR     ; tester si erreur de frame
goto err3       ; non, sauter
MESS "F", "E", "R", "R" ; écrire "FERR" dans le buffer de sortie
bcf ER_FR       ; acquitter l'erreur
err3
btfss ER_OV     ; tester si erreur d'overflow
goto msuite     ; envoyer le message
MESS "O", "E", "R", "R" ; écrire "OERR" dans le buffer de sortie
Bcf ER_OV       ; acquitter l'erreur
goto msuite     ; envoyer message d'erreur

; traitement normal du message
; -----
messprep
bcf STATUS, RP0 ; passer banque 0
call preprep    ; préparer réponse
msuite
bsf STATUS, RP0 ; passer banque 1
bsf PIE1, RCIE  ; réautoriser interruption réception

; envoyer réponse
; -----
bsf PIE1, TXIE ; lancer le message
goto loop      ; traiter message suivant
END            ; directive fin de programme

```

En somme, on peut dire que toute la subtilité du programme réside dans la façon de gérer les autorisations d'interruption. Remarquez qu'on ne coupe jamais ni l'émission, ni la réception, on se contente de prendre en compte immédiatement ou de postposer (arrêt de l'interruption) la réception ou l'émission d'un caractère. Je pense que c'est la bonne façon de procéder.

Bien entendu, j'en ai déjà parlé, à partir du moment où l'interruption de réception n'est pas constamment en service, vous devez gérer une erreur d'overflow éventuelle (ce qui a été fait ici). Sans cette précaution, votre réception pourrait s'arrêter de fonctionner.

Vous pourriez me dire que dans le cas d'un overflow, vous perdrez un caractère, mais, si vous arrêtez la réception via CREN et que l'émetteur envoie un caractère à ce moment, vous êtes certains de le perdre. Dans la présente gestion, vous avez la file FIFO de 2 caractères

pour vous laisser le temps de traiter votre buffer d'entrée avant de réautoriser les interruptions. Ceci vous permet tout de même l'exécution de plus de 10.000 instructions pour traiter votre message précédent (eh oui, toujours se mettre à l'échelle de temps du PIC).

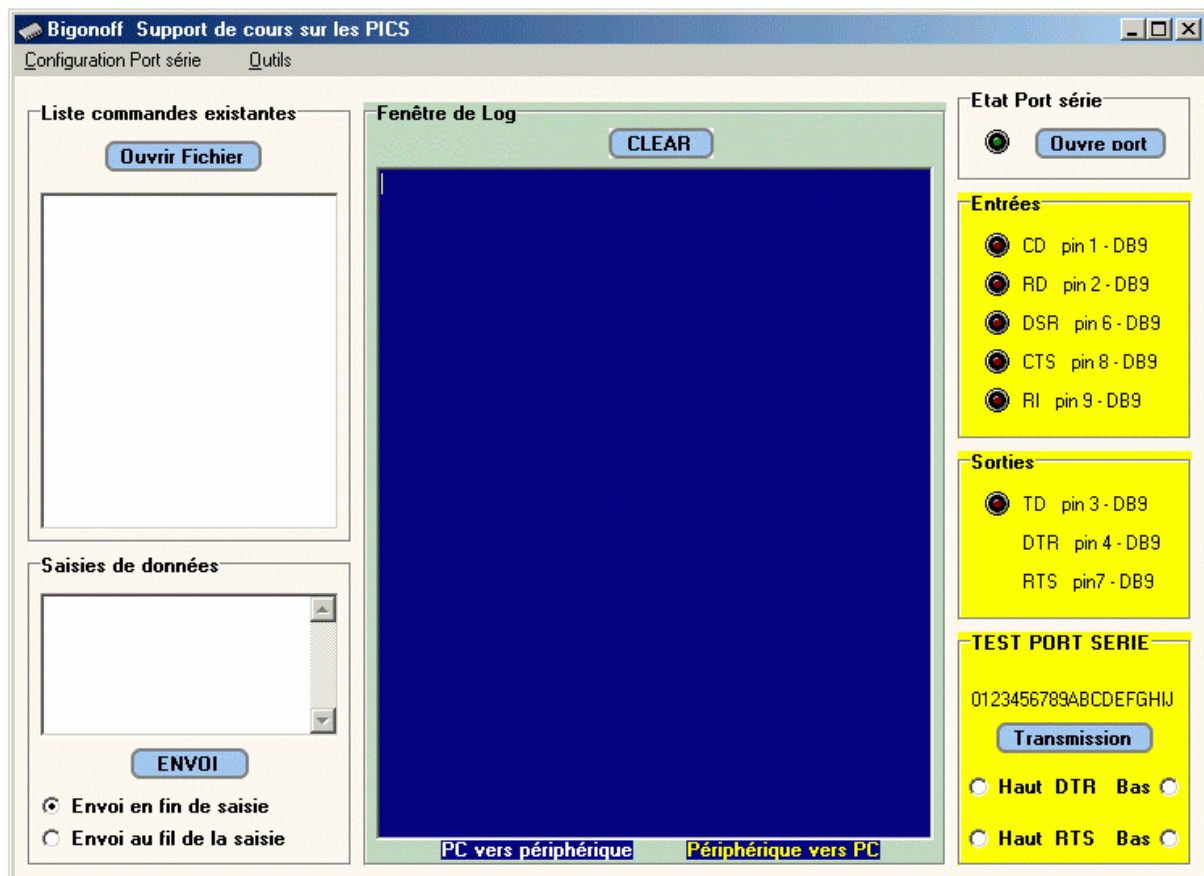
25.12.1 Utilisation de BSCP.

Lancez l'assemblage, et programmez votre PIC avec le fichier obtenu. Placez votre PIC sur votre circuit, mais ne lancez pas encore l'alimentation. Connectez le montage à la RS232 de votre PC. Faites attention de ne pas vous tromper. Notez que le port série de votre PC est prévu pour supporter les pires outrages sans sourciller. C'est pourquoi c'est le port favori des « bidouilleurs ».

Allez maintenant dans votre répertoire « fichiers », vous y voyez un sous-répertoire « BSCP » qui contient l'utilitaire **fourni par mon ami Caméléon**. A l'heure où j'écris ces lignes, il subsiste un bug dans le programme, qui empêche de l'utiliser dans le mode « envoi au fil de la saisie ».

Notez que ce programme constitue un très bon debugger, que vous pourrez utiliser dans vos différentes applications. Il vous permettra, non seulement de surveiller les transferts série asynchrone, mais également de l'utiliser comme debugger, en ajoutant dans vos programmes des routines qui émettront des messages à son intention.

Double-cliquez sur le fichier « bscp.exe », la fenêtre suivante s'ouvre :



La partie « ouvrir fichier » vous permet de préparer à l'avance des commandes dans le fichier « liste.ini ». Celle-ci contient d'ailleurs quelques exemples. Nous ne nous en servons pas dans cette application.

Commencez par aller dans le menu « configuration port série », et réglez les paramètres en fonction de notre liaison, c'est-à-dire :

Numéro du port : en fonction du port sur lequel vous avez connecté votre montage

Vitesse : 9600

Bits de données : 8

Stop-bit : 1

Parité : paire

Contrôle de flux : sans

Une fois fait, cliquez sur « valider »

Le contrôle de flux est utilisé lorsque vous connectez les signaux DTR, RTS, etc. Nous ne les utiliserons pas dans cette application. Le but n'était pas en effet de donner un cours sur le RS232. Si vous désirez créer une application qui gère ces bits, ces pins seront connectées à des pins quelconques de votre PIC, au travers du MAX232. La gestion se fera par manipulation logicielle des lignes.

Si le port série est fermé (témoin « Etat port série » éteint), cliquez sur « ouvre port ». Le témoin s'allume.

Vous taperez vos commandes dans la fenêtre « saisies des données », le résultat s'affichera dans la « fenêtre de log ». Le bouton « clear » permet d'effacer le contenu de cette fenêtre. Les caractères envoyés vers le PC s'affichent en blanc, ceux en provenance du PIC en jaune.

Cliquez sur la case à cocher « Envoi en fin de saisie ».

Maintenant, vous pouvez lancer l'alimentation du PIC. Si tout s'est bien passé, vous devez avoir, en jaune (donc en provenance du PIC), le message :

[PRET]

Allez dans la fenêtre de saisie, pressez une fois sur « return », puis cliquez sur <Envoi>. Vous avez envoyé alors les caractères 0x0D et 0x0A. Votre pic interprète ceci comme une chaîne vide, et répond :

[VIDE]

Tout se passe bien. Retournez dans votre fenêtre de saisie, effacez le retour de ligne, puis tapez « 123456789 » suivis de la touche <Return>. Cliquez sur <Envoi>. En blanc, s'inscrit le message envoyé, en jaune, la réponse du PIC, qui est le même message, mais inversé.

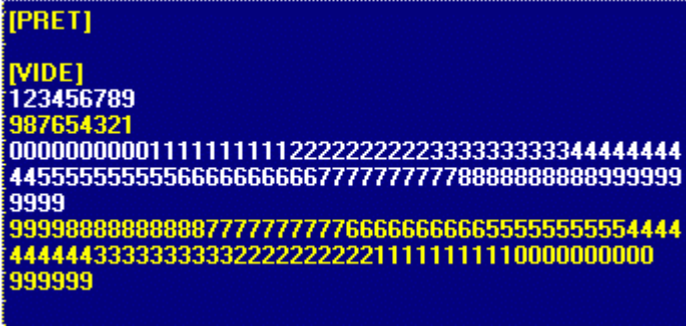
123456789

987654321

Tout se passe bien, jusqu'à présent. Nous allons maintenant taper un long message (100 caractères) pour voir si la limite de buffer est bien gérée. Effacez la fenêtre de saisie des données, et tapez « 0000000000111111111222222222 » etc., jusque 10 fois « 999... ». Ne vous préoccupez pas du retour de ligne automatique, il ne génère pas de « return ». Vous avez donc tapé 100 chiffres. Pressez <Return>, puis cliquez sur <Envoi>.

Vous constatez alors que le PIC répond par les 94 premiers caractères reçus inversés. Il passe alors à la ligne (fin de message), puis répond par les 6 derniers « 9 » inversés. Le PIC a bien limité la taille du buffer à 94 caractères plus les délimiteurs de fin de message (qui provoquent un passage à la ligne). Le message a été scindé en 2 messages, sans perte d'informations.

Voici ce que vous devriez obtenir dans votre fenêtre de log :



```
[PRET]
[VIDE]
123456789
987654321
0000000000111111111222222222333333333344444444
4455555555566666666677777777778888888888999999
9999
9999888888888877777777766666666655555555554444
444444333333333222222221111111110000000000
999999
```

Cliquez sur le bouton <Clear> pour effacer le contenu de cette fenêtre.

Nous allons maintenant essayer de générer des erreurs. Allez dans le menu de configuration du port série, et placez la parité sur « impair ». Passez dans votre fenêtre de saisie, effacez tout, et tapez « 123456 » puis <Return>. Cliquez sur <Envoi>. Le message est envoyé au PIC, mais, comme la parité est fausse, le PIC répond :

[PAR]

en lieu et place du message inversé. Notez que si Bscp arrive à déchiffrer notre message, qui, lui, arrive en parité paire, c'est tout simplement parce qu'il ne détecte pas les erreurs de parité. C'est pratique pour notre application.

Passons encore dans les configurations, et plaçons la parité sur « sans ». Bscp va donc envoyer des données sur 8 bits, le PIC les recevra sur 9. Les conséquences seront triples :

- Les données reçues par le PIC seront incorrectes
- Les données renvoyées par le PIC ne seront pas comprises par Bscp
- Le PIC va recevoir le start-bit de l'octet suivant en place du stop-bit de l'octet en cours. On aura donc une erreur de frame détectée par le PIC.

Le fait que suite à l'erreur, on continue à pouvoir recevoir, prouve que cette erreur est correctement gérée. Dans le cas contraire, la réception serait définitivement bloquée.

Notez le message [OERR] après la réponse à la première salve.

Remarquez qu'établir des routines de gestion d'erreurs n'est pas compliqué, cependant, vérifier qu'elles fonctionnent effectivement équivaut à provoquer volontairement les erreurs, ce qui n'est pas toujours simple.

Nous en avons terminé avec les manipulations, et, par conséquent, avec cet exercice.

25.13 Conclusions

Nous en avons terminé avec les liaisons série asynchrone et avec notre USART . C'est un module qui vous ouvrira bien des portes en ce qui concerne les communications.

Il vous faut encore savoir que MPLAB est dans l'impossibilité de simuler les liaisons série de ce type. Autrement dit, vous devez être très vigilant lorsque vous écrivez ce type de programme, car les erreurs sont en général assez difficiles à détecter.

Vous verrez cependant qu'à ce sujet, je vous réserve une surprise dans le prochain livre.

Notes :

26. Norme ISO7816 et PIC16F876

26.1 Avant-propos

Il n'est pas dans mon propos, ici, de reprendre toute la théorie de la norme ISO7816. Je vous renvoie pour ceci à la première partie du cours. Je vais donc aborder ici les différences entre les cartes sur base de 16F84, et celles sur base de 16F876.

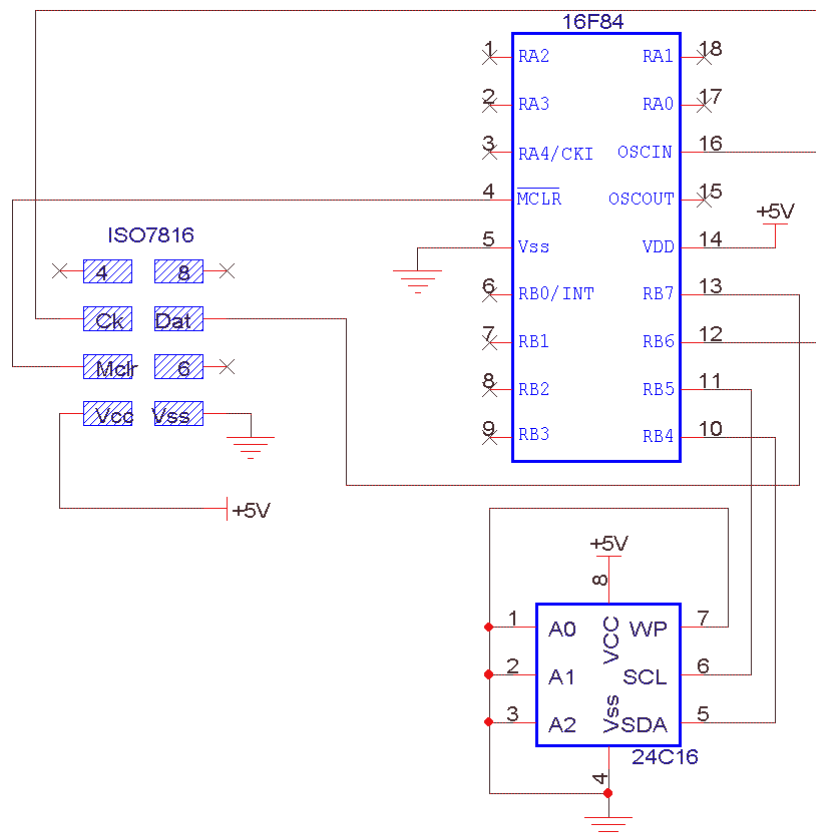
Les « cartes pour serrures codées » que l'on trouve dans le commerce sont toutes issues d'un même modèle. Je vais vous démontrer que ce n'est pas parce que tout le monde procède d'une façon, que cette façon est la meilleure. Il faut conserver son ouverture d'esprit.

Je ne construirai pas de programme ici, les exemples que je vous ai donné tout au long de cet ouvrage suffisent largement à mettre en œuvre ce que je vais vous présenter.

Mais tout d'abord, voyons ce qui existe :

26.2 L'ancêtre «Wafer »

La première carte populaire qu'on a vu apparaître est une carte, dénommée « wafer » ou « piccard1 », qui contient un PIC 16F84 accompagné d'une eeprom de type 24C16. Cette carte présente le schéma suivant :



Que constatons-nous en examinant ce schéma ?

Tout d'abord, l'horloge du PIC est fournie par le maître de l'ISO7816. Notre carte sert de clé, donc subit l'horloge du maître. Attention, il ne s'agit pas ici d'une horloge de transmission synchrone, mais carrément de l'horloge qui va cadencer notre PIC. Pour rappel, la liaison s'effectue en mode asynchrone half-duplex.

L'horloge(CK) est donc appliquée, en tout logique à l'entrée de l'oscillateur de notre PIC (OSCIN). Ce dernier n'aura donc pas besoin d'un quartz pour fonctionner, il suffira de lui attribuer le paramètre de configuration « `_XT_OSC` ».

Ensuite, nous constatons que cette horloge est également appliquée à la pin RB6. Cette pin ne sert à rien en mode de fonctionnement normal, mais elle permet de programmer le PIC dans un programmeur de carte à puces. En effet, la programmation des PICs s'effectue via une liaison synchrone, l'horloge étant appliquée sur RB6, et les données de programmation sur RB7. MCLR servant à transmettre la tension de programmation.

Les pins d'alimentation (alimentation en provenance du maître), ne posent aucun problème.

La ligne MCLR est appliquée à la pin MCLR du PIC. Ceci permet tout d'abord de provoquer un reset du PIC au moment où le maître le désire, et ensuite d'appliquer la tension de programmation lorsqu'on connecte la carte dans un programmeur de carte à puce.

La ligne bidirectionnelle de donnée (DAT) permet de véhiculer les données entre la carte et son hôte. RB7 n'a pas été choisie au hasard, puisqu'elle permet également de véhiculer les données en mode programmation du PIC.

Reste l'eprom, sélectionnée via A0/A2 en adresse B '000', avec autorisation d'écriture (WP à la masse). **L'horloge SCL** est connectée à la pin RB5 du PIC, **la ligne de donnée SDA** transitent via la pin RB4. Souvenez-vous que nous travaillons en I²C.

L'alimentation est fournie à la carte par la pin Vcc du connecteur.

Pour résumer, nous avons là une carte qui permet d'utiliser la norme ISO7816 de façon totalement logicielle, et qui permet la programmation du PIC sans démonter celui-ci de la carte.

L'eprom peut être programmée par chargement préalable dans le PIC d'un logiciel qui reçoit les informations du programmeur de carte à puce, et qui les envoie vers l'eprom. Ce logiciel est appelé « **loader** ». En somme, il transforme provisoirement le PIC de la carte en programmeur d'eprom.

Certaines cartes utilisent les contacts libres du connecteur ISO pour connecter directement l'eprom sur ces contacts, et ainsi, permettre la programmation directe de l'eprom sans charger de loader, grâce à un programmeur de cartes à puce modifié.

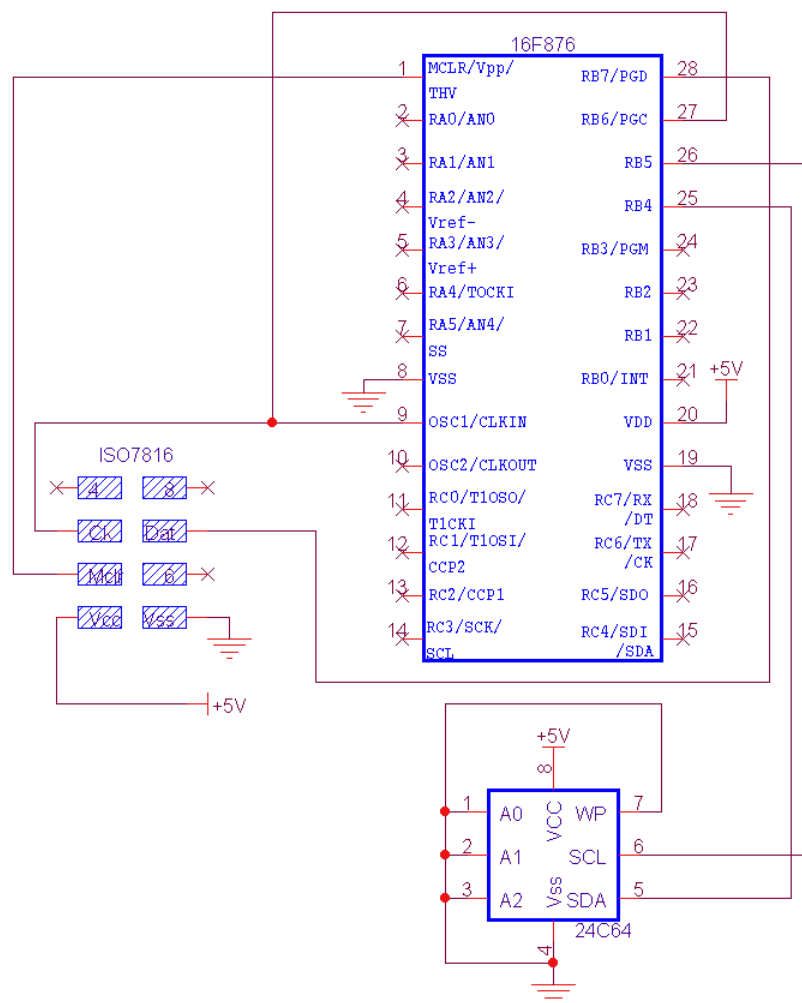
Nous pouvons donc en conclure qu'il est difficile de faire plus efficace pour ces cartes de type piccard1.

26.3 La succession « piccard2 ».

La taille des programmes nécessaires à diverses applications, qui sortent du cadre de cet ouvrage, ont incité les utilisateurs à rechercher de la mémoire supplémentaire (programme et RAM).

Ils ont donc recherché dans la gamme des PICs après un composant plus puissant. Ils l'ont trouvé sous forme du modèle 16F876. Ils en ont profité également pour augmenter la taille de l'eeprom, et sont passés au modèle 24C32 ou 24C64.

Voici donc le schéma qu'ils ont décidé d'utiliser :



Que pouvons-nous dire sur ce schéma ? Et bien, tout simplement qu'il est strictement identique à celui de la wafer. Autrement dit :

- La liaison série asynchrone half-duplex avec le maître s'effectue comme sur un 16F84, en gérant la transmission bit par bit via RB7.
- La liaison série synchrone I²C avec l'eeprom s'effectue en gérant bit par bit horloge et data, comme sur un 16F84, via RB4 et RB5.

On en déduit que cette carte n'utilise aucune des fonctions intégrées qui fait la puissance de ce PIC.

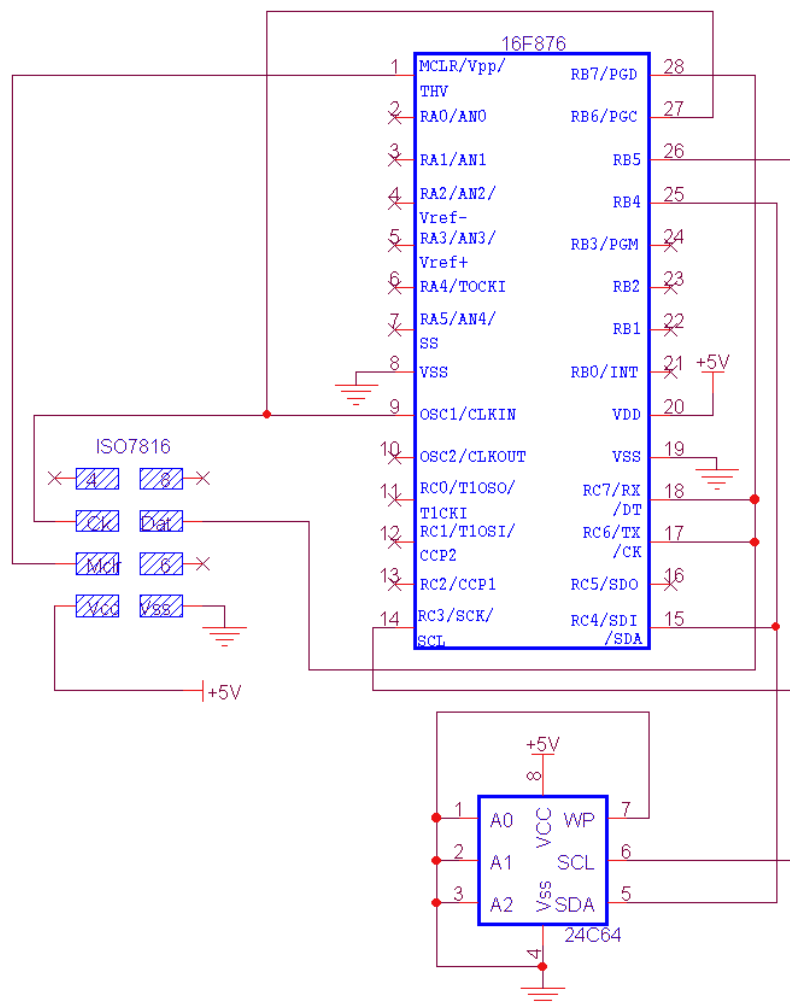
Il ne permet donc pas le moindre gain en terme de traitement de données, il permet uniquement de disposer de la mémoire supplémentaire qu'offre le 16F876.

J'en déduis que (aie, je vais me faire des ennemis) ceux qui ont inventé cette carte ne savaient pas exploiter les nouvelles fonctions de ce composant. Ils s'en sont servis comme d'un « super 16F84 ».

Maintenant que vous savez tout ou presque sur ce composant, vous allez trouver le schéma que je vous propose tout à fait logique.

26.4 La Bigcard 1

Et ben oui, c'est moi qui l'invente (je pense), donc je lui donne le nom que je veux, non ? C'est surtout mon petit clin d'œil. Regardez ce schéma :



Vous constatez que les liaisons d'origine ont été conservées, autrement dit, un programme écrit pour piccard2 fonctionnera sans problème sur bigcard1. Cette carte est de plus programmable exactement comme les autres piccard2.

Par contre, vous constatez en premier lieu les liaisons supplémentaires entre les lignes SCL et SDA de l'EEPROM, et les pins SCL et SDA du PIC. Ceci vous donne l'énorme avantage de pouvoir traiter les liaisons I²C en vous servant de l'interface MSSP intégré au PIC.

Vous n'avez donc plus qu'à gérer les échanges avec l'EEPROM comme nous l'avons vu dans le chapitre concerné. Ceci vous laisse beaucoup plus de temps pour vous occuper du reste, et donc, augmente la puissance disponible du PIC, pour le cas où celle-ci viendrait à faire défaut.

De plus, la norme I²C impose une résistance de rappel au +5V pour les lignes SCL et SDA. Nous mettons en service les résistances de rappel au +5V du PORTB. Ainsi, nous faisons d'une pierre deux coups : en conservant la liaison SCL et SDA vers RB4 et RB5, non seulement nous restons compatibles avec tous les programmes, mais, en plus, nous économisons nos deux résistances de rappel.

Reste le problème de notre liaison asynchrone, qui est parfaitement gérée par cette carte. J'ai donc connecté TX et RX à la ligne « DAT », ce qui permet d'utiliser notre USART pour gérer la transmission. De nouveau, ceci libère pas mal de temps pour traiter les informations. La pin RB7 sert de nouveau à la fois comme compatibilité piccard2, également pour permettre la programmation sur circuit, et enfin comme résistance de rappel au +5V des pins Tx et Rx.

Comme nous avons affaire à une liaison full-duplex, lorsque émettons, nous mettons en service TXEN. Nous coupons TXEN sitôt l'émission achevée, ce qui nous replace automatiquement en réception. Nous n'avons donc pas de conflit de ligne (le problème était d'ailleurs le même pour l'utilisation de RB7 sur les piccard2, qui devait passer tantôt en lecture, tantôt en écriture).

Il vous faudra simplement veiller à attendre 372 coups d'horloge (donc 93 cycles d'instruction) entre l'émission de 2 octets, ceci afin de tenir compte du second stop-bit imposé par la norme. Autrement dit, la fin de l'émission sera déterminée par le passage à « 1 » de TMRT, suivi par une boucle comportant 93 cycles d'instruction. C'est enfantin à réaliser.

Vous voici donc en possession d'une carte plus puissante et plus simple à utiliser qu'une simple piccard2. Nul doute que vous en ferez bon usage.

26.5 La bigcard 2

Le principal défaut de toutes ces cartes est que le PIC tourne à la vitesse imposée par le maître.

Or, on peut tenir le raisonnement suivant :

- L'horloge du maître est indispensable pour déterminer la vitesse des échanges sur la liaison série ($1/372^{\text{ème}}$ de l'horloge fournie).
- Par contre, cette horloge utilisée comme horloge principale, bride les performances du PIC, puisqu'à l'heure où j'écris ces lignes, on peut faire tourner un 16F876 à 20 MHz, l'horloge ISO7816 étant souvent centrée aux alentours de 4Mhz.

Ces conditions sont les suivantes :

-
- The diagram illustrates the hardware setup for an ATmega16F876 microcontroller. The microcontroller is connected to a +5V power supply and ground. A 20MHz crystal oscillator is connected to pins 9 (OSC1/CLKIN) and 10 (OSC2/CLKOUT), with a 15pF capacitor connected to pin 9. The microcontroller's pins are connected to an ISO7816 transceiver as follows: pin 14 (SCL) to pin 1 (A0), pin 15 (SDA) to pin 2 (A1), pin 16 (SDO) to pin 3 (A2), pin 17 (RC6/TX/CK) to pin 4 (WP), and pin 18 (RC7/RX/DT) to pin 5 (SCL). The transceiver is powered by +5V (pin 8) and ground (pin 4). The 24C64 EEPROM is connected to the transceiver's output pins: pin 5 (SCL) to pin 1 (A0), pin 6 (SCL) to pin 2 (A1), pin 7 (WP) to pin 3 (A2), pin 8 (VCC) to +5V, and pin 9 (VSS) to ground.

516

- Soit vous connaissez la vitesse de l'horloge fournie par le maître, et vous n'avez plus alors qu'à calculer le débit correspondant une fois pour toutes ($1/372^{\text{ème}}$ de la fréquence). Exemple, si l'horloge vaut 4Mhz, le débit sera de $4.000.000 / 372 = 10572$ bauds.
- Soit vous désirez que votre PIC calcule cette vitesse lui-même, ce qui est facilement réalisable, puisque l'entrée d'horloge est appliquée à la pin CCP1 (c'est bien entendu voulu). En utilisant le module CCP, vous pouvez mesurer le temps d'une période de l'horloge, donc sa fréquence. Il vous suffit alors de calculer la valeur correspondante de SPBRG pour déterminer la fréquence utilisée par l'USART.

Vous voici donc en présence d'une **super-carte, capable de travailler à très haute vitesse**, et qui devrait vous permettre de reculer les limites que vous rencontrerez peut-être dans vos applications.

Vous pouvez ajouter un jumper entre, par exemple, RB1 et la masse, pour n'autoriser le PIC à calculer une nouvelle fréquence de travail que lorsque vous positionnez le jumper. Ceci permet de ne pas devoir recalculer à chaque démarrage, la valeur calculée étant sauvegardée en eeprom interne.

26.6 Conclusions

Vous avez constaté que ce n'est pas parce que tout le monde fait la même chose qu'il n'y a pas moyen de faire mieux.

J'utilise une Bigcard1 depuis longtemps pour toutes sortes d'applications personnelles, et avec la plus grande satisfaction. La Bigcard2 est à réserver aux cas désespérés, mais vous permet d'envisager le recul des limites de traitement des autres cartes, sans remplacer votre microcontrôleur.

Inutile de m'écrire pour me demander des exemples d'utilisation illicite de ces différentes cartes, je ne répondrai pas. Il y a assez d'utilisations pratiques parfaitement légales de ces cartes pour pouvoir vous amuser. D'autant que les législations sont différentes selon les pays, et que je ne peux les connaître toutes.

A vous donc d'utiliser au mieux ces cartes, vous disposez de tous les renseignements et exemples d'utilisation des différentes techniques (USART, PC, CCP) pour créer vos propres programmes.

Notes : ...

27. Le mode sleep

27.1 Généralités

Il ne reste plus grand-chose à dire concernant ce mode. Non seulement j'en ai déjà parlé dans le premier ouvrage, mais, de plus, j'en ai parlé dans les modules concernés de cette seconde partie.

Néanmoins, je pense que rassembler les informations éparses en un seul endroit n'est pas faire preuve de redondance inutile.

27.2 Les mécanismes

Nous savons déjà que tous les modules ne peuvent fonctionner en mode sleep. Il importe de savoir quels sont les événements susceptibles de réveiller votre PIC placé en mode sommeil.

Voici tout d'abord les 3 types de réveil possible de votre PIC.

- Reset sur la pin MCLR
- Débordement du watchdog, si celui-ci est mis en fonction au moment de la configuration
- Positionnement d'un flag d'interruption (par exemple INTF), alors que le bit d'autorisation correspondant est positionné (dans cet exemple, INTE).

Notez que l'action sur la pin MCLR provoque le reset du PIC. Les 2 autres cas se contentent de réveiller le PIC, qui continuera l'exécution du programme à partir du point de mise en sommeil.

Le débordement du watchdog dans le mode « sleep » réveille donc le PIC sans provoquer de reset. La lecture du bit « TO » du registre STATUS positionné à « 0 », vous indiquera que c'est le watchdog qui aura causé le réveil du PIC. Si ce n'est pas cet événement que vous attendez, il vous suffira donc, si TO = 0, de replonger le PIC en mode sommeil en attendant l'événement concerné.

Il faut également se souvenir que l'instruction « sleep », si elle est effectivement exécutée, resette le watchdog (donc, exécute un clrwdt), et positionne le bit TO.

Voici un exemple :

```
Wait
sleep                ; placer le PIC en mode sleep
btfss STATUS,TO      ; c'est le watchdog qui a réveillé ?
goto wait            ; oui, alors on se rendort
..                   ; non, alors c'est le bon réveil
```

Pour réveiller le PIC, il faut que le bit d'autorisation correspondant soit positionné. Mais ceci s'effectue sans tenir compte de l'état de GIE. Nous aurons alors 2 réveils possibles sur l'utilisation du bit d'autorisation.

- Si GIE est positionné, le PIC se réveille, l'instruction suivant « sleep » est exécutée, puis on se connecte en 0x04 pour traiter l'interruption.
- Si GIE n'est pas positionné, le PIC se réveille et poursuit l'exécution du programme où il était arrivé (à la ligne qui suit « sleep »).

Que se passe-t-il si le flag d'interruption est positionné avant l'exécution de l'instruction « sleep » ?

Et bien, l'instruction « sleep » sera considérée comme un « nop ». Ceci aura les conséquences suivantes :

- Le PIC ne sera pas mis en mode sommeil
- Le bit TO ne sera pas positionné à « 1 »
- Le watchdog ne sera pas positionné.

Faites donc attention à ces différents éléments. La boucle donnée dans l'exemple précédent risquerait de boucler sans fin, le pic ne se mettant pas en sommeil. Après le temps de débordement du watchdog, le PIC serait reseté, puisque débordement sans être en mode « sleep ».

Dans le même ordre d'idées, ne vous fiez pas uniquement à l'instruction « sleep » pour resetter votre watchdog. Si vous avez besoin de le faire impérativement, faites précéder l'instruction « sleep » d'une instruction « clrwdt ».

Un dernier point important est à prendre en compte, c'est le temps de réveil du PIC. En effet, il faut savoir qu'au moment où le PIC se réveille, l'oscillateur, qui était arrêté, se remet en service.

Or, il faut un certain temps pour que l'oscillateur soit suffisamment stable pour pouvoir commencer à exécuter les instructions. Le PIC attend donc 1024 Tosc avant de recommencer à travailler. Tenez donc compte que le réveil d'un PIC qui était en mode sommeil est loin d'être instantané.

27.3 Les interruptions concernées

Toutes les interruptions ne peuvent avoir lieu en mode sommeil, donc toutes ne peuvent réveiller le PIC. Voici la liste des événements de ce type susceptibles de réveiller votre PIC :

- Lecture ou écriture avec le module PSP (PSPIF / PSPIE)
- Timer 1 configuré en mode compteur asynchrone (TMR1IF / TMR1IE)

- Module CCP1 en mode capture (CCP1IF / CCP1IE)
- Module CCP2 en mode capture (CCP2IF / CCP2IE)
- Événement « start » ou « stop » détecté pour MSSP en mode I²C (SSPIF / SSPIE)
- Fin de transmission pour le module MSSP en mode esclave, SPI ou I²C (SSPIF / SSPIE)
- Fin d'émission pour le module MSSP en mode esclave, SPI ou I²C (SSPIF / SSPIE)
- Réception pour le module USART en mode esclave synchrone (RCIF / RCIE)
- Emission pour le module USART en mode esclave synchrone (TXIF / TXIE)
- Fin de conversion A/N, si l'horloge de conversion est en mode RC (ADIF / ADIE)
- Fin d'écriture eeprom (EEIF / EEIE)

Ce sont là les seuls événements susceptibles de réveiller votre PIC.

27.4 Conclusion

Le mode sleep est principalement utilisé lorsque le PIC doit attendre des temps très longs entre 2 actions.

Ce mode permet donc d'économiser de l'énergie, et de limiter le vieillissement du PIC, en le plaçant au repos le plus souvent possible.

N'oubliez cependant pas que les pins conservent leur niveau, et donc, que si vous placez le pic en mode sommeil pour économiser l'énergie (fonctionnement sur piles par exemple), vous devrez, avant de le mettre dans ce mode, placer les niveaux adéquats, fonctions de votre électronique associée, sur les pins afin de limiter au maximum la consommation.

N'oubliez cependant pas que ce mode nécessite quelques précautions d'utilisation.

Notes : ...

28. Le reste du datasheet

28.1 Introduction

Nous arrivons tout doucement à la fin de l'étude de nos 16F87x. Je vais maintenant passer en revue les différents détails du datasheet que je n'ai pu faire entrer dans les différents chapitres vus jusqu'à présent.

28.2 Le mode LVP

Le mode **LVP**, pour **Low Voltage Programming** permet de programmer le PIC sans utiliser la haute tension de programmation (13V). La programmation s'effectue alors via la pin RB3/PGM sous une tension d'alimentation classique du PIC.

Ceci vous autorise à programmer votre PIC SANS avoir besoin d'un programmeur. Il vous suffit d'avoir le logiciel de programmation qui gère ce mode, la liaison pouvant s'effectuer directement depuis le port // de votre PC, par exemple. Attention au port série, qui utilise des tensions non compatibles. Dans ce cas, l'utilisation d'un convertisseur de niveau de type MAX232 ou STN232 ou équivalent sera nécessaire.

C'est également un mode très pratique pour les utilisateurs de PC portables, qui se trouvent confrontés, du fait de l'utilisation d'une tension insuffisante sur le port série, à l'obligation d'utiliser une alimentation externe : le comble pour un portable, d'être contraint de se connecter sur une source d'alimentation.

Ce mode est obtenu en plaçant « **LVP_ON** » au niveau de votre directive « **_CONFIG** ». Ce bit est validé par défaut lorsque vous recevez votre PIC vierge, ce qui vous autorise à la programmer d'origine dans ce mode.

Une fois le PIC dans ce mode, la pin RB3 n'est plus disponible en tant que pin d'entrée/sortie normale. Vous la laisserez donc libre, et reliée uniquement à votre PC.

De plus, si vous avez mis en service les résistances de rappel au +5V du PORTB, vous devrez effacer le bit 3 du registre TRISB de votre programme, afin de libérer cette pin de la résistance de rappel, et permettre ainsi son utilisation en tant que pin de programmation.

Le PIC en mode « **LVP_ON** » peut être programmé indifféremment en mode haute-tension ou en mode LVP.

Le PIC en mode **LVP_OFF** ne peut être programmé qu'en mode haute-tension.

Cependant, le bit LVP ne peut être mis hors-service (égal à « 1 ») qu'uniquement en mode de programmation haute-tension. Ceci vous évite, en cas d'erreur, de vous retrouver avec un PIC placé en mode haute-tension, alors que vous n'avez pas de programmeur.

De même, une fois le PIC en mode haute-tension, l'effacement de ce bit ne pourra également se faire qu'en mode haute-tension. Si, par hasard, vous tentez de programmer en mode LVP un PIC qui a été programmé avec le bit LVP hors-service par un programmeur

classique, vous serez dans l'impossibilité d'y arriver, et vous devrez reprogrammer ce PIC à l'aide d'un programmeur classique. Quitte alors à en profiter pour remettre le bit LVP en service (`_LVP_ON` ou bit LVP = « 0 »).

Une opération d'effacement (bulk erase) nécessite une tension d'alimentation, en mode LVP, comprise entre 4,5 et 5,5V. Toute autre tension empêchera l'effacement du PIC (et la modification des bits de configuration).

Par contre, une programmation « ordinaire » pourra se faire dans toute la plage de tension admise par le PIC.

28.3 PWRTE et BODEN

Ces bits sont positionnés au moment de la programmation, vous choisissez de les mettre ou non en service grâce à la directive « `CONFIG` ».

Le bit `PWRTE` concerne le délai entre le démarrage de l'oscillateur du PIC et l'exécution de la première instruction.

Le PIC attend un temps typique de 72ms entre la mise sous tension et l'exécution de l'instruction inscrite à l'emplacement 0x00.

Le délai de 72ms est réalisé de façon indépendante de l'oscillateur principal du PIC, grâce à un circuit RC intégré.

Si `PWRTE` est positionné (`_PWRTE_ON`) et que l'oscillateur principal n'est pas en mode RC (donc est en mode XT, HS, ou LP), alors ce temps est augmenté d'une durée égale à 1024 `Tosc` de l'horloge principale. Ceci assure un délai supplémentaire qui assure le démarrage stabilisé certain du quartz.

Je vous conseille de toujours positionner ce bit pour vos applications réelles.

`BODEN` permet de mettre en service le circuit de détection de la chute d'alimentation du PIC (brown-out circuit). S'il est mis en service (`_BODEN_ON`), alors, si la tension d'alimentation chute sous les 4V (pour les PICs actuels standards) pendant une durée supérieure à 100µs, alors le PIC sera automatiquement resetté (voir chapitre sur les différents types de reset).

Le redémarrage interviendra quand la tension repassera au-dessus de la barre des 4V (typiques). Le temps entre le redémarrage de l'oscillateur et l'exécution de la première instruction sera de 72ms pour un oscillateur de type RC, auquel on ajoutera 1024 cycles `Tosc`, comme pour le bit `PWRTE`.

Notez que le positionnement du bit `BODEN` valide automatiquement le bit `PWRTE`. Autrement dit, si vous utilisez la détection de chute de tension, le délai de 1024 cycles au démarrage sera automatiquement en service, aussi bien à la mise sous tension, qu'après un reset sur chute de tension.

Je vous conseille également de toujours positionner ce bit, excepté s'il est préférable pour votre application que le PIC fonctionne le plus longtemps possible, quitte à avoir un fonctionnement erratique en cas de diminution trop forte de la tension d'alimentation.

Bien évidemment, si votre PIC est alimenté avec une tension inférieure à la tension de détection du Brown-out (entre 3,7V et 4,35V), vous ne pourrez mettre BODEN en service.

Remarques importantes :

si vous maintenez MCLR à l'état bas durant la mise sous tension, le temps de démarrage est tout de même pris en compte. Autrement dit, si vous remplacez MCLR à l'état haut après un temps supérieur à 72ms (+ éventuellement 1024 T_{osc}), le PIC démarrera alors instantanément.

Ceci peut être utilisé pour synchroniser le démarrage de plusieurs PICs utilisant le même oscillateur. Vous êtes alors certain que tous les programmes commencent à exécuter leur instruction à l'adresse 0x00 strictement en même temps.

Un autre point à savoir, c'est que si vous validez le bit BODEN, le comparateur utilisé en interne consomme un certain courant (quelques dizaines de μ A). Aussi, si vous placez votre pic en mode sleep, ce bit augmentera la consommation du pic en conséquence.

Et bien, je crois que nous avons ici terminé l'étude des 16F87x en eux-mêmes. Il ne nous reste plus qu'à étudier quelques techniques utiles.

28.4 Les adresses « réservées » 0x01 à 0x03

J'ai lu beaucoup de « n'importe quoi » à propos des adresses 0x01 à 0x03. D'aucun prétendent que ce sont des adresses réservées, je vais vous démontrer le contraire.

L'adresse 0x00 est le vecteur de reset. A ce propos, j'ai lu également pas mal de propos qui prêtent à confusion. Lors d'un reset, le PIC utilise le vecteur 0x00 comme adresse de reset. C'est donc l'adresse en elle-même qui est le vecteur d'interruption, non son contenu.

Donc, prétendre qu'à l'adresse 0x00, on place le vecteur de reset, tout comme prétendre qu'à l'adresse 0x04 on place le vecteur d'interruption est complètement faux. En fait, à ces adresses, on y place ce qu'on veut, il ne s'agit nullement d'un vecteur.

Par opposition, certains processeurs utilisent des emplacements mémoire comme « vecteurs d'interruptions ». Dans ces emplacements mémoire, on placera alors une adresse (pas une instruction de saut). Le processeur ira dans ces emplacements mémoire, chargera le vecteur qui y est placé (adresse), et sautera à l'adresse désignée par ce vecteur. Le 680x0 fonctionne sur ce principe, et bien d'autres d'ailleurs.

Rien de tel sur notre PIC. Les vecteurs sont câblés en interne, et ne sont pas éditables. On aura donc 3 vecteurs internes, avec un contenu figé.

- Le vecteur de reset vaut 0x0000, donc le PIC se connectera sur l'adresse 0x0000 dans laquelle on mettra ce qu'on désire à chaque reset ou mise sous tension.

- Le vecteur d'interruption vaut 0x0004, donc le PIC se connectera sur l'adresse 0x0004 dans laquelle on mettra ce qu'on désire lors de chaque interruption validée et détectée.
- Le vecteur de debuggage vaut 0x2004, donc le PIC se connectera sur l'adresse 0x2004 à chaque « halt » du programme en mode debugger. A cette adresse, étant donné qu'il n'y a qu'un emplacement, vous serez bien obligé de mettre une instruction « goto ».

Revenons-en à nos adresses 0x0001 à 0x0003. Qu'est-ce qui a fait dire à certains qu'il s'agissait d'adresses réservées ?

En fait, c'est simple à comprendre. Lorsque l'utilisateur utilise les routines d'interruptions, lesquelles se connectent à l'adresse 0x0004, il ne reste qu'à l'utilisateur que 4 emplacements mémoire (0x0000 à 0x0003) pour placer son programme. Comme c'est très limité (c'est le moins que l'on puisse dire), il placera donc en général à l'adresse 0x0000 un saut vers son programme principal. On retrouvera donc une structure du type :

```
ORG 0x0000      ; vecteur de reset
goto start     ; on saute au programme principal

ORG 0x0004      ; vecteur d'interruption
..             ; routines d'interruptions.
```

Certains se croient même « obligés » d'insérer un « goto interrupt » à l'adresse 0x0004, ce qui ne sert à rien, qu'à ralentir la réaction effective à l'interruption.

La même logique a donc amené ces utilisateurs à se dire : « Etant donné qu'à l'adresse 0x00 j'ai un saut, et que je n'ai rien d'autre entre 0x00 et 0x04, c'est donc que ces adresses sont réservées, dans le cas contraire, Microchip aurait placé son vecteur d'interruption en 0x01 ».

C'est aller un peu vite en besogne. Imaginons que votre programme principal ne se trouve pas en page 0, mais en page 3. Vous allez donc être contraint de commencer votre programme par :

```
ORG 0x0000      ; vecteur de reset
bsf PCLATH,3    ; pointer sur page 3
bsf PCLATH,4
goto start     ; on saute au programme principal
```

Ici, il ne nous reste plus que l'adresse 0x03 de libre. Et si maintenant, on désire debugger un tel programme, Microchip nous recommande de placer un « nop » comme première instruction. Ceci nous donne :

```
ORG 0x0000      ; vecteur de reset
nop             ; pour le debugger
bsf PCLATH,3    ; pointer sur page 3
bsf PCLATH,4
goto start     ; on saute au programme principal
```

Cette fois, nous avons utilisé toute la zone entre 0x00 et 0x04. Voilà pourquoi il existe cet écart, il n'y a rien de magique ni de réservé dans ces emplacements. C'est la taille minimale nécessaire pour couvrir tous les cas d'applications pratiques.

De plus, si vous n'utilisez pas les interruptions, vous n'avez nul besoin d'insérer un saut, vous pouvez simplement écrire l'intégralité de votre programme à partir de l'adresse 0x00.

D'ailleurs, même si vous utilisez les interruptions, rien ne vous empêche de placer 3 instructions à partir de l'adresse 0x00 avant de réaliser votre saut. Par exemple :

```
ORG 0x0000      ; vecteur de reset
bsf    STATUS,RP0 ; passer en banque 1
bcf    TRISB,0    ; RB0 en entrée
bcf    STATUS,RP0 ; repointer en banque 0
goto   start      ; sauter pour poursuivre le reste du programme.
```

Vous constatez que ces adresses sont donc parfaitement utilisables, et aucunement réservées.

28.5 L'overclockage

Après une discussion avec kudelsko, administrateur du site <http://kudelsko.free.fr/>, il est apparu que ce dernier, ne pouvant se procurer des PICs d'une fréquence supérieure à 4Mhz, s'est mis à tenter de les utiliser à la fréquence maximale autorisée pour la gamme, c'est-à-dire, à les overclocker.

Il en est résulté que tous les PICs ainsi « boostés » acceptaient sans problème de travailler avec les nouvelles fréquences permises, c'est-à-dire :

10MHz pour tous les 16F84
20Mhz pour tous les 16F84A
20MHz pour tous les 16F87x

Il a alors tenté d'interroger Microchip, pour obtenir des informations sur cette facilité déconcertante à overclocker les PICs. La réponse de Microchip a été que cette information était une information stratégique commerciale, et ne pouvait être divulguée.

On peut alors raisonnablement penser, vu le grand nombre de tests effectués par ce sympathique webmaster, que les versions -04, -10, et -20 sont strictement identiques, et ne diffèrent que du point de vue de l'étiquetage (et du prix).

On peut également raisonnablement penser que, vu le faible prix des PICs, un test de la vitesse sur la chaîne de fabrication n'était probablement pas effectué.

Donc, pour vos applications non critiques (étant donné qu'il subsiste toutefois un doute), je vous suggère de tenter l'expérience.

Ne manquez pas d'aller voir sur le site de Kudelsko, et de lui rapporter votre expérience en la matière. Plus il y aura d'utilisateurs qui auront tenté l'expérience, plus on pourra passer éventuellement de l'interrogation à la certitude.

Notez que les essais de Kudelsko ont été effectués en remplaçant les condensateurs connectés au quartz par des condensateurs de 4,7pf. Vous pouvez tenter de conserver les valeurs originales et lui renvoyer vos résultats.

28.6 Ce dont je n'ai pas parlé

Il reste des points très pointus que je n'ai pas abordé dans cette seconde partie, afin d'accélérer sa sortie et sa diffusion. Je pense principalement aux techniques de bootloader et de debugger sur circuit.

J'aborderai ces points avec énormément de détails, avec des exemples pratiques, et des programmes opérationnels, dans les troisième et quatrième parties du cours : « Les secrets des 16F87x ».

Je termine avec un mot sur les programmes objets. J'ai voulu inclure un chapitre à ce sujet, seulement, je me suis aperçu qu'expliquer leur fonctionnement et leur utilisation allait nécessiter plus de 100 pages.

Les techniques mises en œuvre (création d'objets, linkage, multi-langages) sortent du cadre de ce cours, et relèvent plus de l'utilisation de MPLAB, MPASM, et MPLINK que du fonctionnement des 16F87x.

En se limitant à l'utilisation d'un seul langage, il est aussi simple d'inclure directement dans le fichier source les routines dont on a besoin, plutôt que de créer des fichiers objets, écrits sous forme relogeable, et donc plus complexes.

J'ai donc abandonné cette idée, partant du fait que les programmes objets sont principalement utiles dans les applications multi-langages, et que les créateurs de tels programmes pourront sans peine s'en sortir, les techniques utilisées étant de grands classiques de la programmation.

Annexe1 : Questions fréquemment posées (F.A.Q.)

Je vais tenter de répondre ici à un maximum de questions que se posent les utilisateurs en général. Ce chapitre s'étoffera au fur et à mesure des questions que je recevrai.

A1.1 Je n'arrive pas à utiliser mon PORTA

Vous n'avez probablement pas utilisé la maquette fournie, ou alors vous avez effacé la ligne concernée. En général, le problème provient de l'oubli de configurer le registre ADCON1. Votre port fonctionne alors en mode analogique.

A1.2 Le choix du programmeur

Je vous conseille de vous rendre sur le site de Ic-Prog : www.ic-prog.com, afin de télécharger la dernière version de IC-Prog. Sur ce site, vous trouverez également des schémas de programmeurs facilement réalisables. Pensez à choisir une version disposant d'une alimentation.

Notes :...

Contribution sur base volontaire

La réalisation de ces cours m'a demandé beaucoup de temps et d'investissements (documentations, matériel, abonnements, etc.).

Aussi, pour me permettre de poursuivre, je vous demande, si cela est dans vos possibilités, et si vous appréciez ce que je fais, de contribuer un peu, chacun selon ses possibilités et ses désirs.

J'ai donc besoin de votre aide pour continuer l'aventure. En effet, je ne dispose plus vraiment de la capacité de consacrer l'intégralité de mon temps libre à écrire des cours et des programmes sans recevoir un petit "coup de pouce".

Cependant, je ne voulais pas tomber dans le travers en verrouillant l'accès aux fichiers, et en imposant un paiement pour les obtenir. En effet, je tiens à ce qu'ils restent disponibles pour tous.

J'ai donc décidé d'instaurer un système de contribution sur base volontaire en permettant à celui qui le désire, et en fonction de ses propres critères, de m'aider financièrement. Le but n'étant pas de me faire riche, mais plutôt de m'aider à "joindre les 2 bouts".

Il ne s'agit donc pas d'un paiement, ni d'une obligation. Il s'agit simplement d'une assistance sans promesse d'aucun sorte, et sans contrainte. Je continuerai à répondre au courrier de tout le monde, sans distinction, et sans interrogation à ce sujet.

Une bonne méthode consiste donc, pour celui qui le désire, à télécharger le document choisi, le lire ou l'utiliser, puis décider si cela vaut ou non la peine de m'aider sur base de l'usage que vous en faites.

Si oui, vous vous rendez sur mon site : www.abcelectronique.com/bigonoff ou www.bigonoff.org, et vous suivez la page « cours-part1 ». Vous y trouverez, dans la page « contributions », la procédure à suivre. Pensez que ces contributions me sont très utiles, et contribuent à me permettre de continuer à travailler pour vous.

Pour remercier ceux qui ont contribué, soit par l'envoi d'une lettre, soit par la création d'un site d'utilité publique, j'offrirai le logiciel BigoPic V2.0.

N'oubliez pas de mettre votre email en caractère d'imprimerie, pour que je puisse vous répondre.

Je réponds toujours au courrier reçu. Aussi, si vous n'obtenez pas de réponse, n'hésitez surtout pas à me contacter pour vérifier s'il n'y a pas de nouveau un problème.

Merci d'avance à tous ceux qui m'ont aidé ou m'aideront à poursuivre ce travail de longue haleine.

B. Utilisation du présent document

Le présent ouvrage est destiné à faciliter la compréhension de la programmation des PICs en général par l'étude de toutes les possibilités de la gamme mid-range. La suite sera bientôt disponible et est en cours de réalisation.

Communiquez à l'auteur (avec politesse) toute erreur constatée afin que la mise à jour puisse être effectuée dans l'intérêt de tous, si possible en utilisant le livre de report d'information présent sur la page de téléchargement du cours.

J'avais autorisé de proposer la première partie en téléchargement sur des sites de webmasters qui en faisaient la demande. Je remercie tous ceux qui ont joué correctement le jeu.

Cependant, certains ne l'ont pas fait, et n'ont pas mis leur site régulièrement à jour en fonction des nouvelles versions. Ceci implique que je reçois régulièrement du courrier de personnes qui me signalent des erreurs corrigées depuis longtemps, et des utilisateurs qui s'aperçoivent avec dépit qu'ils viennent d'imprimer une version obsolète du cours. Vu le nombre de pages, ça ne fait pas plaisir à tout le monde.

Aussi, pour ces raisons, et par facilité de maintenance pour moi, j'ai décidé que ce cours serait disponible uniquement sur mon site : www.abcelectronique.com/bigonoff ou www.bigonoff.org

Aussi, si vous trouvez mon cours ailleurs, merci de m'en avertir.

Bien entendu, j'autorise (et j'encourage) les webmasters à placer un lien sur le site, inutile d'en faire la demande. Bien entendu, je ferai de même en retour si la requête m'en est faite. Ainsi, j'espère toucher le maximum d'utilisateurs.

Le présent ouvrage peut être utilisé par tous, la modification et la distribution sont interdites sans le consentement écrit de l'auteur.

Tous les droits sur le contenu de ce cours, et sur les programmes qui l'accompagnent demeurent propriété de l'auteur.

L'auteur ne pourra être tenu pour responsable d'aucune conséquence directe ou indirecte résultant de la lecture et de l'application du cours ou des programmes.

Toute utilisation commerciale est interdite sans le consentement écrit de l'auteur. Tout extrait ou citation dans un but d'exemple doit être accompagné de la référence de l'ouvrage.

J'espère n'avoir enfreint aucun droit d'auteur en réalisant cet ouvrage, je n'ai utilisé que les programmes mis gracieusement à la disposition du public par la société Microchip. Les datasheets sont également disponibles gracieusement sur le site de cette société, à savoir : <http://www.microchip.com>

Si vous avez aimé cet ouvrage, si vous l'utilisez, ou si vous avez des critiques, merci de m'envoyer un petit mail, ou mieux, de poster un message sur mon site. Ceci me permettra de savoir si je dois ou non continuer cette aventure avec les parties suivantes.

Certains continuent à envoyer des messages sur mon ancienne adresse. Prenez connaissance de la bonne adresse, pour ne pas encombrer des adresses non concernées. Vous risquez de plus d'attendre longtemps votre réponse.

Sachez que je réponds toujours au courrier reçu, mais notez que :

- Je ne réalise pas les programmes de fin d'étude pour les étudiants (même en payant), c'est une demande qui revient toutes les semaines dans mon courrier. Tout d'abord je n'ai pas le temps, et ensuite je ne pense pas que ce soit un bon service. Enfin, pour faire un peu d'humour, si je donnais mes tarifs, ces étudiants risqueraient un infarctus.
- Je n'ai malheureusement pas le temps de debugger des programmes complets. Inutile donc de m'envoyer vos programmes avec un message du style « Ca ne fonctionne pas, vous pouvez me dire pourquoi ? ». En effet, je passe plus de 2 heures par jour pour répondre au courrier, si, en plus, je devais debugger, j'y passerais la journée. Vous comprenez bien que c'est impossible, pensez que vous n'êtes pas seul à poser des questions. Posez plutôt une question précise sur la partie qui vous semble inexacte.
- Si vous avez des applications personnelles, n'hésitez pas à les faire partager par tous. Pour ce faire, vous pouvez me les envoyer. Attention cependant, faites précéder votre envoi d'une demande, je vous redirigerai alors sur une autre boîte, celle-ci étant limitée à 512K.
- Avec cette version, j'essaie de répondre aux demandes légitimes des personnes qui travaillent sur différentes plates-formes (Mac, Linux, Windows, etc.). Si, cependant, la version fournie est inexploitable sur votre machine, merci de me le faire savoir. Notez cependant que ce cours utilise MPLAB pour les exercices, il faudra donc éventuellement adapter ces exercices en fonction du logiciel qu'il vous sera possible d'utiliser.

Je remercie tous ceux qui m'ont soutenu tout au long de cette aventure, et qui se reconnaîtront. Nul doute que sans les nombreux encouragements reçus, ce livre n'aurait jamais vu le jour.

Merci au webmaster de www.abcelectronique.com, pour son hébergement gratuit.

Merci à Byte, pour sa proposition d'hébergement gratuit.

Merci à Grosvince pour sa proposition d'hébergement gratuit.

Merci à Bonny Gijzen pour la modification de IC-Prog qui servira pour la suite de l'aventure : www.ic-prog.com

Merci à Kudelsko (<http://kudelsko.free.fr/>) pour les informations concernant l'overclockage.

Dernière remarque : il est impossible que vous trouviez trace d'un plagiat ici, étant donné que je n'ai lu **aucun** ouvrage sur le sujet, autre que le datasheet de Microchip. Tout est donc issu de mes propres expériences. Donc, en cas de copie manifeste (j'en ai vu), ce sont les autres qui ont copié (ceci vaut également pour les ouvrages édités de façon classique).

- Edition terminée en révision beta0 le 22/09/2002.
- Révision 1 : 30/12/2002 : première version dégrossie
- Révision 2 : 30/01/2003 : Corrections mineures page 411, 413, 185, 86, 87, 89, 108, 251, 314, 340, 412.
- Révision 3 : 12/03/2003 : Ajout de l'explication du bug sur l'I²C esclave page 434.
- Révision 4 : 15/04/2003 : Correctifs pages 12, 30, 34, 65, 108, 120, 130, 155, 190, 193, 211, 219, 220, 251, 230, 262, 330, 331, 349, 379, 390, 411, 417, 439, 443, 445, 475, 491, 494, 507.
- Révision 5 : 27/04/2003 : Correctifs pages 88, 91, 123, 160, 164, 209, 226, 231, 234, 235, 239, 243, 271, 281, 284, 293, 305
- Révision 6 : 13/07/2003 : Correctifs pages 252, 367, 417, 423, 448, 472, 477, 483
- Révision 7 : 19/04/2003 : Correctifs importants pages 441, 445, 485, 507.
- Révision 8 : 09/03/2004 : Corrections mineures page 169, 232, 393, 396, correctif programme page 454 et fichier.
- Révision 9 : 09/04/2004 : Corrections mineures page 38, correctif légendes chronogrammes pages 323 et 324.
- Révision 10 : 09/06/2004 : Correctifs pages 365, 375, 380, 401, chronogrammes pages 365, 366, 367, renumérotation de certaines pages, table des matières, modification de mon adresse courrier.
- Révision 11 : 25/08/2004 : Correctif pages 444, 460, 524, ajouts pages 380 et 381, correction du programme cli_opti.asm et du cours pages 83 à 92, remarque importante page 47.
- Révision 12 : 27/03/2006 : Correctifs pages 100, 101, 102, 124, 270, 317, 325, 326, 365, 367, 414, remarques page 489, 412 et 415. Modifs mineures dans certains fichiers .asm, dont les maquettes.

Réalisation : Bigonoff

Email : bigocours@hotmail.com (Attention BIGOCOURS PAR BIGONOFF)