

**MINF**

**Programmation des PIC32MX**

# **Programmation des microcontrôleurs PIC32MX**



**Théorie PIC32MX**

**Christian HUBER (CHR)  
Serge CASTOLDI (SCA)**  
**Version 1.9 novembre 2017**



## CONTENU DU COURS

### **2. Architecture et périphériques du PIC32MX**

Ce chapitre décrit l'architecture des PIC32MX ainsi que les périphériques à disposition.

### **3. Jeu d'instructions des PIC32MX**

Ce chapitre introduit l'organisation des instructions et fournit en annexe une référence sur le jeu d'instructions MIPS32.

### **4. Pile et sous-programme**

Ce chapitre décrit le mécanisme de la pile et son usage lors des appels de sous programmes, ainsi que les mécanismes utilisés par le compilateur C pour passer les paramètres aux sous-programmes.

### **5. Les interruptions**

Ce chapitre décrit tout d'abord le principe des interruptions avant de décrire le mécanisme des interruptions des PIC32MX en particulier. Suit le comment de la mise en œuvre en utilisant les fonctions de la PLIB\_INT de Harmony.

### **6. Timers, PWM et capture**

Ce chapitre traite de l'utilisation des Timers des PIC32MX, ainsi que l'utilisation des modules associés au Timers pour la capture, la comparaison et la génération de signaux PWM.

### **7. Gestion de la liaison RS232**

Ce chapitre décrit l'architecture des UART des PIC32MX, ainsi que le principe du traitement de réception et d'émission en utilisant les interruptions et des FIFO.

### **8. Gestion du bus SPI**

Ce chapitre traite du bus SPI et de sa gestion avec le microcontrôleur PIC32MX du kit ainsi que les fonctions de la PLIB\_SPI.

### **9. Gestion du bus I2C**

Ce chapitre traite du bus I2C et de sa gestion avec le microcontrôleur PIC32MX du kit ainsi que les fonctions de la PLIB\_I2C.

### **10. Programmation concurrente**

Ce chapitre traite des mécanismes pour obtenir une programmation concurrente. Introduction à RTOS.

## HISTORIQUE DES VERSIONS

### **Version 1.0**

La version 1.0 correspond à la création du cours théorique pour le PIC32MX.

### **Version 1.5**

La version 1.5 correspond à la reprise complète du cours théorique pour le PIC32MX en introduisant dans les exemples les fonctions de la **PLIB de Hamony** 1.0.

### **Version 1.7**

La version 1.7 correspond à l'adaptation du cours théorique pour le PIC32MX en introduisant dans les exemples les fonctions de la **PLIB de Hamony** 1\_06. La version 1.6 de certains chapitres correspond à des corrections en cours de semestre, d'où le saut à la version 1.7.

### **Version 1.8**

La version 1.8 correspond à l'adaptation du cours théorique pour le PIC32MX en introduisant dans les exemples les fonctions de la **PLIB de Hamony** 1\_08.

Pour éviter la redondance avec le cours pour le labo, les aspects pratiques seront réduits au minimum; par contre le cours labo utilisera le MHC sous forme de recette de cuisine avec un minimum d'explications.

### **Version 1.9**

Reprise et relecture par SCA.

Passage à Harmony 1.11.

## REMARQUE PRÉLIMINAIRE

Le présent cours est illustré à travers l'utilisation des outils de Microchip, dont notamment le framework Harmony. Au fil de l'évolution des versions, il est possible que quelques différences existent entre la version utilisée lors de l'élaboration et la version actuelle lors de la lecture. Les principes présentés restent applicables.

Dans la mesure du possible, la version utilisée pour les exemples sera indiquée.

Le lecteur est invité à se reporter à la documentation de sa version disponible sous :

<Répertoire Harmony>\v<n>\doc

**MINF**  
**Programmation des PIC32MX**

# **Chapitre 2**

## **Architecture et périphériques des PIC32MX**



### **Théorie PIC32MX**

**Christian HUBER (CHR)**  
**Serge CASTOLDI (SCA)**  
**Version 1.91 décembre 2018**



## CONTENU DU CHAPITRE 2

<b>2. Architecture et périphériques des PIC32MX</b>	<b>2-1</b>
2.1.1. Groupe documentation Core	2-1
2.1.2. Groupe documentation Implémentation	2-1
2.1.3. Groupe documentation Peripheral	2-2
<b>2.2. Schéma bloc du PIC32MX</b>	<b>2-3</b>
2.2.1. Schéma bloc du CPU	2-4
2.2.1.1. MDU	2-4
2.2.1.2. MMU	2-4
2.2.2. Concept de Pipeline	2-4
2.2.2.1. Principe exécution des instructions	2-4
2.2.3. Prefetch cache	2-6
2.2.4. Les registres du CPU	2-7
<b>2.3. Organisation de la mémoire</b>	<b>2-8</b>
2.3.1. Kseg et Useg	2-9
<b>2.4. Les périphériques du PIC32MX795F512L</b>	<b>2-10</b>
<b>2.5. Oscillateur et circuit de reset</b>	<b>2-12</b>
2.5.1. L'oscillateur	2-12
2.5.1.1. Les modes de l'oscillateur	2-12
2.5.1.2. Configuration avec oscillateur à quartz	2-12
2.5.1.3. Situation avec oscillateur externe	2-13
2.5.1.4. Détail du système de génération des horloges	2-13
2.5.2. Circuit de Reset	2-14
<b>2.6. Annexe A – Liste E/S du PIC32MX795F512L</b>	<b>2-15</b>
2.6.1. Boitier 100 pin TQFP	2-15
2.6.2. Liste des E/S	2-15
2.6.2.1. Port A	2-15
2.6.2.1. Port B (entrées analogiques)	2-16
2.6.2.1. Port C	2-16
2.6.2.1. Port D	2-17
2.6.2.1. Port E	2-17
2.6.2.1. Port F (Uart SPI I2C)	2-18
2.6.2.1. Port G	2-18
2.6.3. Liste des CN (Change Notification), ordre des ports	2-19
2.6.1. Liste des CN (Change Notification)	2-20
<b>2.8. Historique des versions</b>	<b>2-21</b>
2.8.1. Version 1.0 mars 2014	2-21
2.8.2. Version 1.5 novembre 2014	2-21
2.8.3. Version 1.7 novembre 2015	2-21
2.8.4. Version 1.8 novembre 2016	2-21
2.8.5. Version 1.8.1 décembre 2016	2-21
2.8.6. Version 1.9 octobre 2017	2-21
2.8.7. Version 1.91 décembre 2018	2-21



## 2. ARCHITECTURE ET PÉRIPHÉRIQUES DES PIC32MX

Les PICs existent en de nombreuses variantes et familles. Dans ce chapitre, nous allons focaliser notre étude sur la famille PIC32MX et plus particulièrement sur le modèle MX795F512L utilisé sur le kit PIC32.

Les documents de références sont l'ensemble des datasheets PIC32 que l'on trouve sur le réseau sous :

...\\PROJETS\\SLO\\1102x\_SK32MX775F512L\\Datasheets\\PIC32 Family Reference Manual.

La documentation est découpée en de multiples sections, dont notamment :

- Un datasheet regroupant l'essentiel de la famille du microcontrôleur MX795 et de ses périphériques.
- 35 sections distinctes détaillant le fonctionnement des différentes parties internes et périphériques.

Les sections se répartissent en 3 groupes :

### 2.1.1. GROUPE DOCUMENTATION CORE

 PIC32 Family Reference Manual, Sect. 02 CPU.pdf

### 2.1.2. GROUPE DOCUMENTATION IMPLEMENTATION

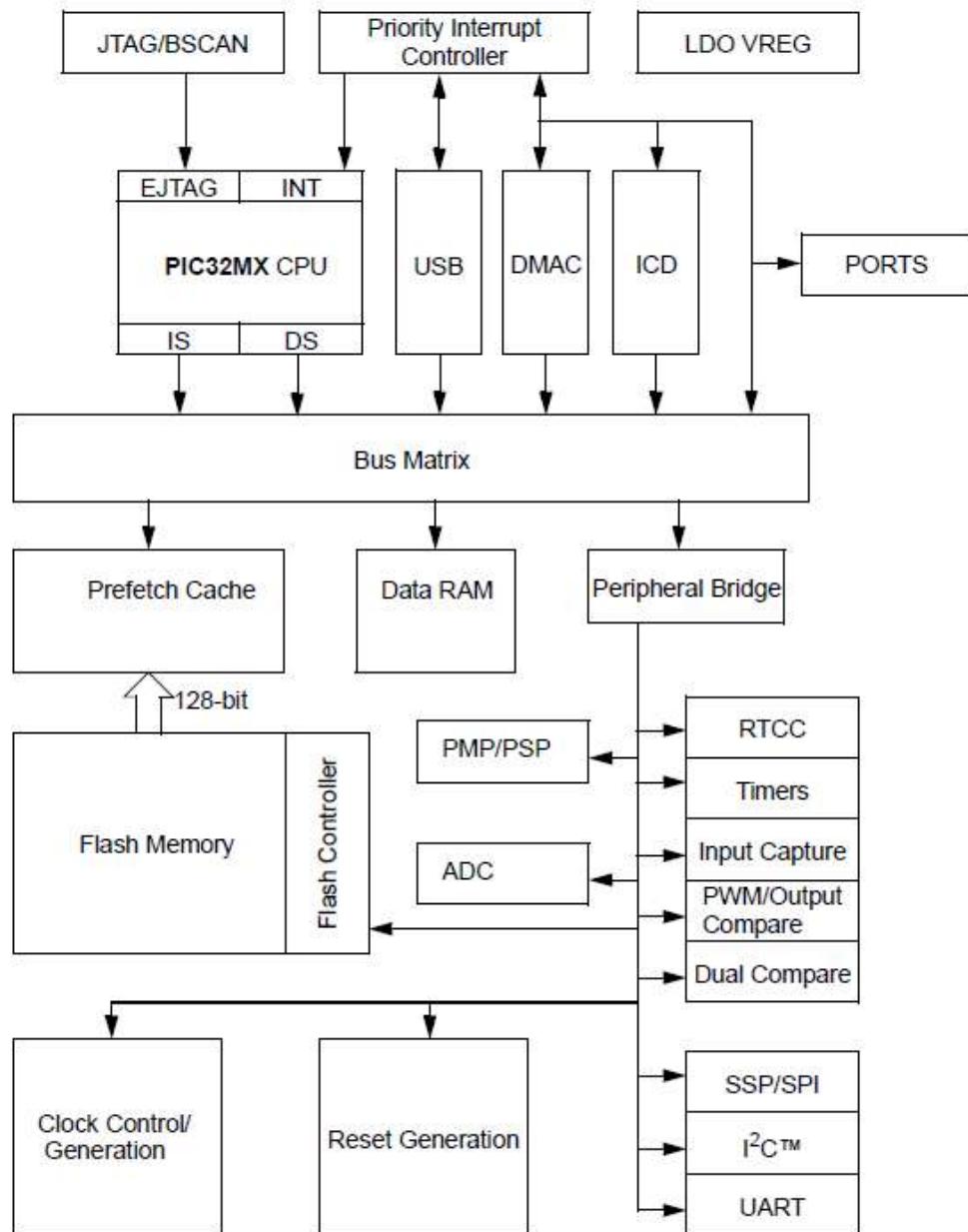
- Section 3. Memory Organization
- Section 4. Prefetch Module
- Section 5. Flash Programming
- Section 6. Oscillator
- Section 7. Resets
- Section 8. Interrupts
- Section 9. Watchdog Timer and Power-up Timer
- Section 10. Power-Saving Modes
- Section 31. Direct Memory Access (DMA) Controller with programmable Cyclic Redundancy Check (CRC)
- Section 32. High-Level Integration (Configuration, Code Protection and Voltage Regulation)
- Section 33. Device Programming, Debugging, In-Circuit and In-Circuit Testing

### 2.1.3. GROUPE DOCUMENTATION PERIPHERAL

The PIC32MX devices have many peripherals that allow it to interface with the external world. The following sections of this manual discuss the PIC32MX peripherals:

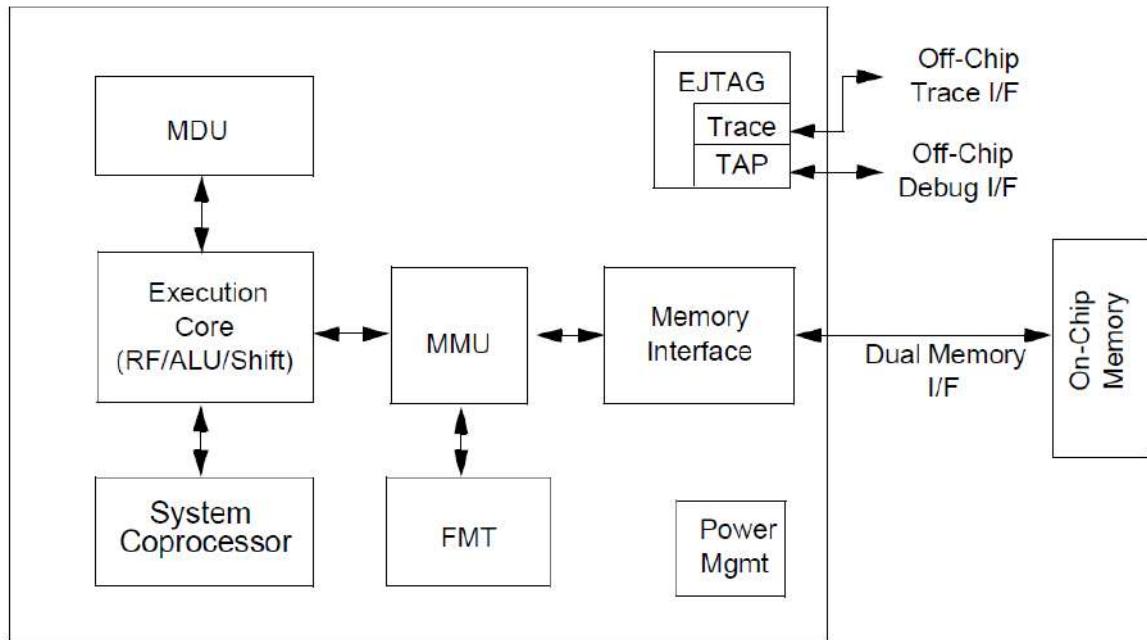
- Section 12. I/O Ports
- Section 13. Parallel Master Port
- Section 14. Timers
- Section 15. Input Capture Module
- Section 16. Output Compare/Pulse Width Modulation (PWM) Module
- Section 17. 10-bit A/D Converter
- Section 19. Comparator Module
- Section 20. Comparator Voltage Reference Module
- Section 21. UART Module
- Section 23. SPI Module
- Section 24. I<sup>2</sup>C™ Module
- Section 27. USB OTG
- Section 29. Real-Time Clock/Calendar (RTCC) Module

## 2.2. SCHÉMA BLOC DU PIC32MX



Le schéma bloc ci-dessus présente de manière synthétique les différents éléments composant le PIC32MX.

## 2.2.1. SCHÉMA BLOC DU CPU



### 2.2.1.1. MDU

Le MDU (Multiply/Divide Unit) permet des multiplications 32 x 16.

### 2.2.1.2. MMU

Le MMU (Memory Management Unit) s'occupe de la gestion de la mémoire en collaboration avec le FMT (Fixed Mapping Translation).

## 2.2.2. CONCEPT DE PIPELINE

Le principe du pipeline consiste à subdiviser le traitement d'une instruction en plusieurs sous-opérations. Le traitement traverse les différents étages, d'où le nom de pipeline. Avec un pipeline, plutôt que de terminer complètement une instruction avant de passer à la suivante, le processeur peut débuter le traitement d'une nouvelle instruction sans attendre que la précédente soit terminée. Cela permet d'améliorer la vitesse d'exécution, car les différentes étapes sont effectuées simultanément.

### 2.2.2.1. PRINCIPE EXÉCUTION DES INSTRUCTIONS

Les différents étages du pipeline d'exécution du PIC32 sont :

- **I** Inst. Fetch L'instruction est chargée depuis la mémoire
- **E** Execution Décodage et exécution
- **M** Mem. Fetch Les opérandes sont chargés depuis la mémoire (SRAM ou flash)
- **A** Align Le résultat est aligné
- **W** Writeback Le résultat est écrit dans sa destination

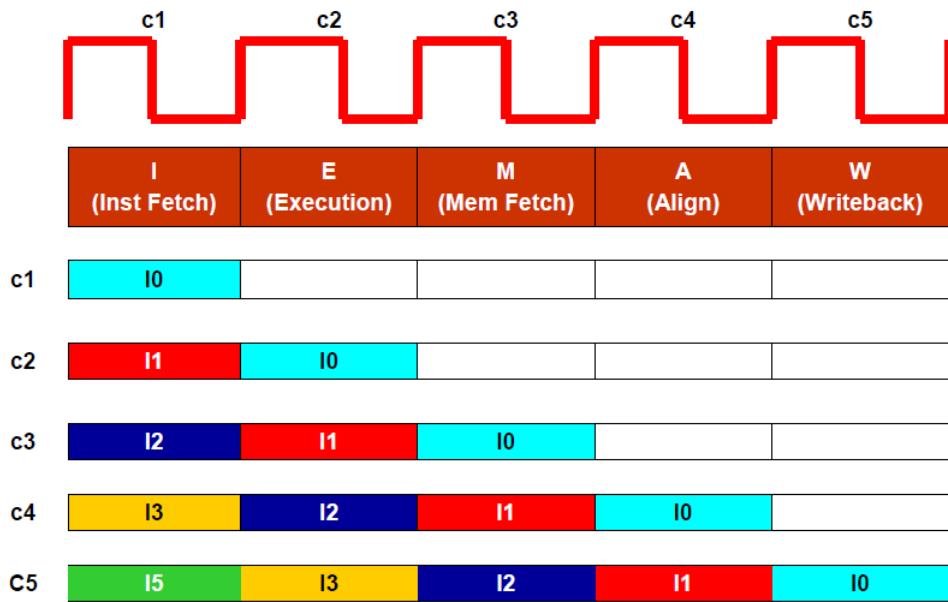
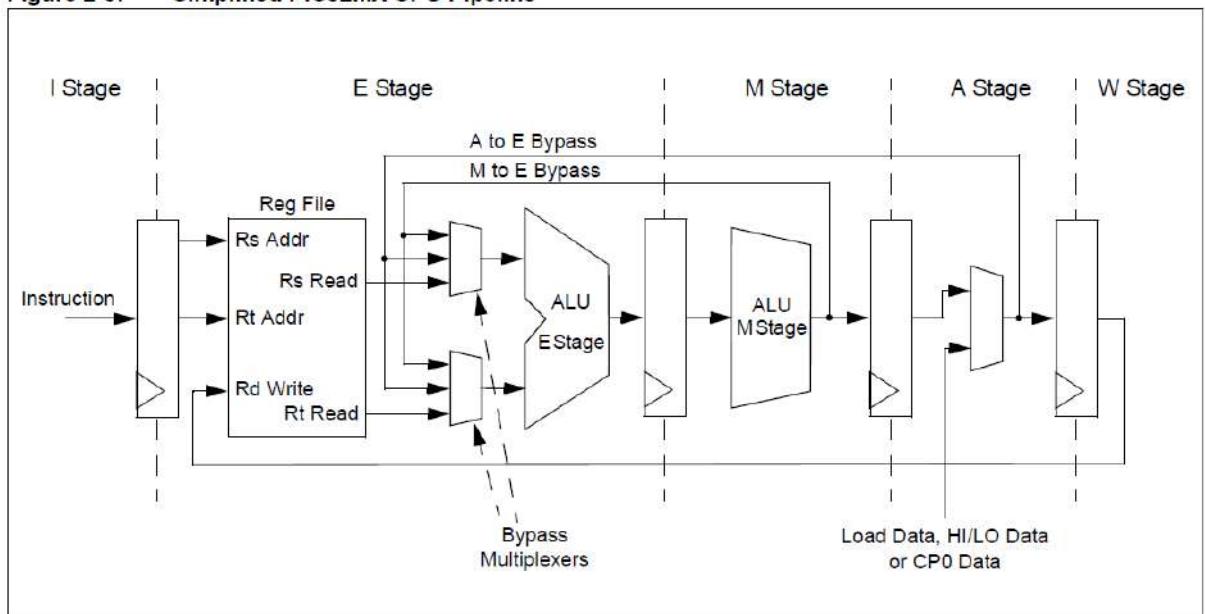


Figure 2-3: Simplified PIC32MX CPU Pipeline



Sources :

- "PIC32 Execution Pipeline" webinar, disponible sous :  
<https://www.microchip.com/webinars.microchip.com/WebinarDetails.aspx?dDocName=en542876>
- "Section 2. CPU for Devices with M4K® Core" (DS61113), tiré du PIC32 Family Reference Manual"

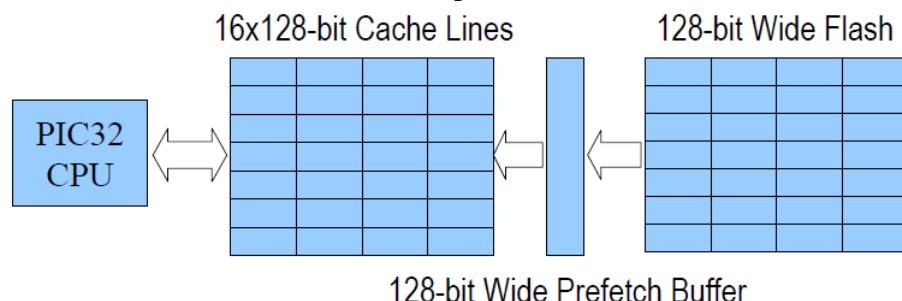
### 2.2.3. PREFETCH CACHE

Lorsque le CPU est plus rapide que la flash, il n'est alors pas possible de faire un accès flash (lecture instruction suivante) à chaque instruction sans temps morts. On peut alors avoir recours à un stockage RAM intermédiaire intelligent (le cache).

Pour les PIC32, la limite où les accès flash ralentiraient le CPU se situe environ à une fréquence  $f_{CPU}$  de 30 MHz (fréquence de lecture maximale de la flash). Plus on augmente la fréquence du CPU, plus la flash va devenir limitante.

Le prefetch cache existe sur les PIC de la série de celui du kit (PIC32MX795F512L,  $f_{CPU}$  maximale 80 MHz). Mais ce n'est pas le cas pour tous les PIC32. Il n'existe par exemple pas sur ceux de la série du PIC32MX130 ( $f_{CPU}$  maximale 50 MHz).

Le prefetch cache est une mémoire RAM rapide intermédiaire entre la flash et le CPU :



Caractéristiques :

- Le prefetch cache peut être désactivé pour un fonctionnement déterministe
- 16 lignes de 128 bits (= 4 instructions 32 bits)
- Accès en lecture/écriture depuis le code
- Possibilité de verrouiller n'importe quelle ligne (par exemple pour des portions de codes fréquentes ou à optimiser : petite boucle, prologue d'interruption)
- Possibilité d'utiliser jusqu'à 4 lignes pour des constantes stockées en flash et fréquemment lues

Principes :

- Lors de l'exécution d'une instruction, le prefetch cache charge automatiquement depuis la flash les 128 bits suivant l'instruction exécutée, "au cas où".
- Lors d'un accès flash de la part du CPU :
  - Soit la donnée est déjà présente dans le cache : l'accès peut alors être fait immédiatement.
  - Soit cette adresse de flash n'est pas dans le cache. On a alors une condition de "**cache miss**".

Les opérations sont alors :

1. Le prefetch buffer charge alors les 128 bits contenant la donnée voulue.
2. Les 128 bits sont stockés dans la ligne la moins récemment utilisée (Least Recently Used).
3. La donnée peut être retournée au CPU.

Sources :

- "PIC32 Prefetch Cache Module" webinar, disponible sous <https://www.microchip.com/webinars.microchip.com/WebinarDetails.aspx?dDocName=en542873>
- "Section 4. Prefetch Datasheet Cache" (DS60001119), tiré du PIC32 Family Reference Manual"

## 2.2.4. LES REGISTRES DU CPU

Le processeur dispose de 32 registres généraux appelés GPR (General Purpose Registers).

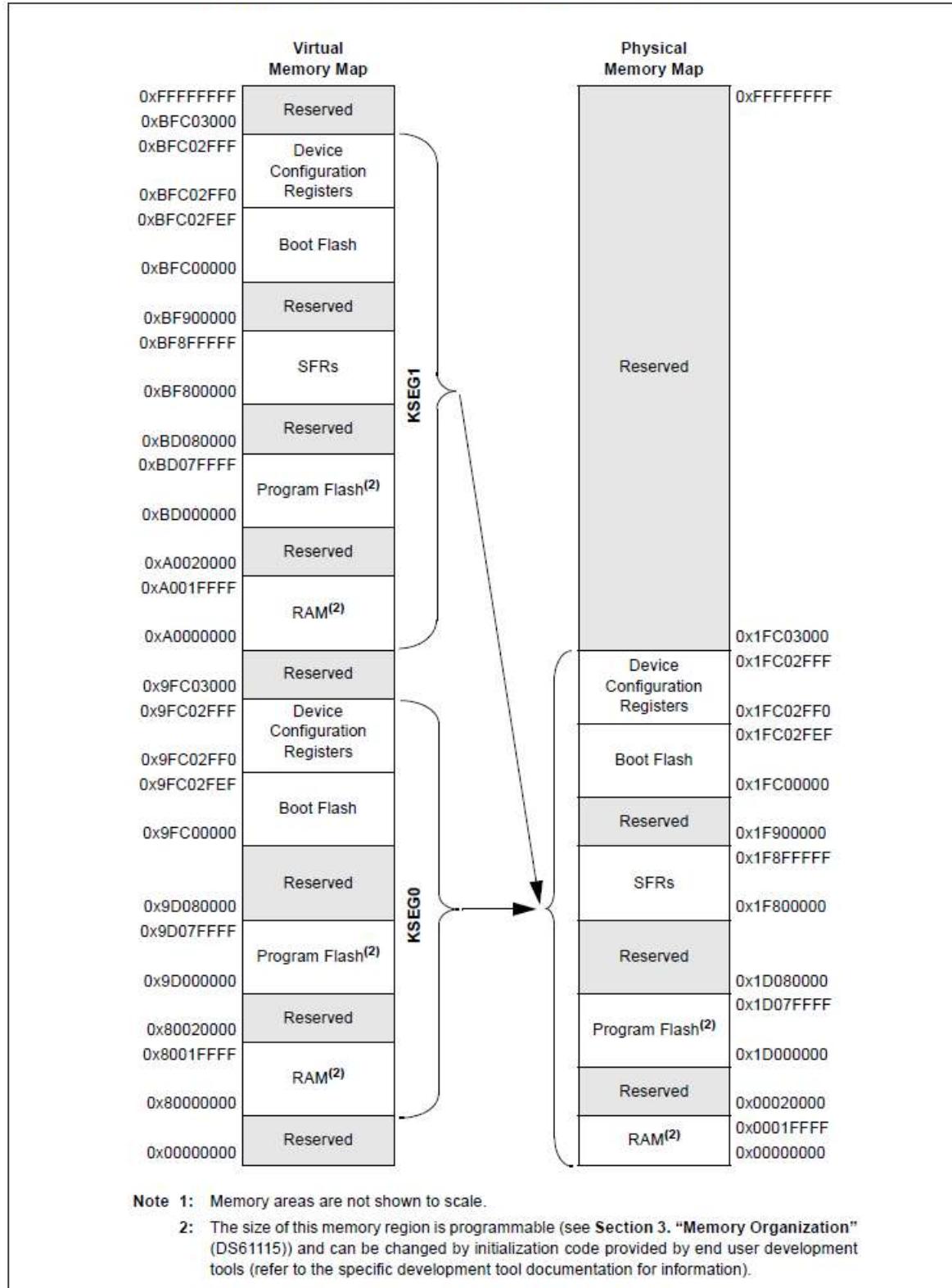
31	0	31	0
r0 (zero)		HI	
r1 (at)		LO	
r2 (v0)			
r3 (v1)			
r4 (a0)			
r5 (a1)			
r6 (a2)			
r7 (a3)			
r8 (t0)			
r9 (t1)			
r10 (t2)			
r11 (t3)			
r12 (t4)			
r13 (t5)			
r14 (t6)			
r15 (t7)			
r16 (s0)			
r17 (s1)			
r18 (s2)			
r19 (s3)			
r20 (s4)			
r21 (s5)			
r22 (s6)			
r23 (s7)			
r24 (t8)			
r25 (t9)			
r26 (k0)			
r27 (k1)			
r28 (gp)			
r29 (sp)			
r30 (s8 or fp)			
r31 (ra)		31	0
		PC	

General Purpose Registers

Special Purpose Registers

## 2.3. ORGANISATION DE LA MÉMOIRE

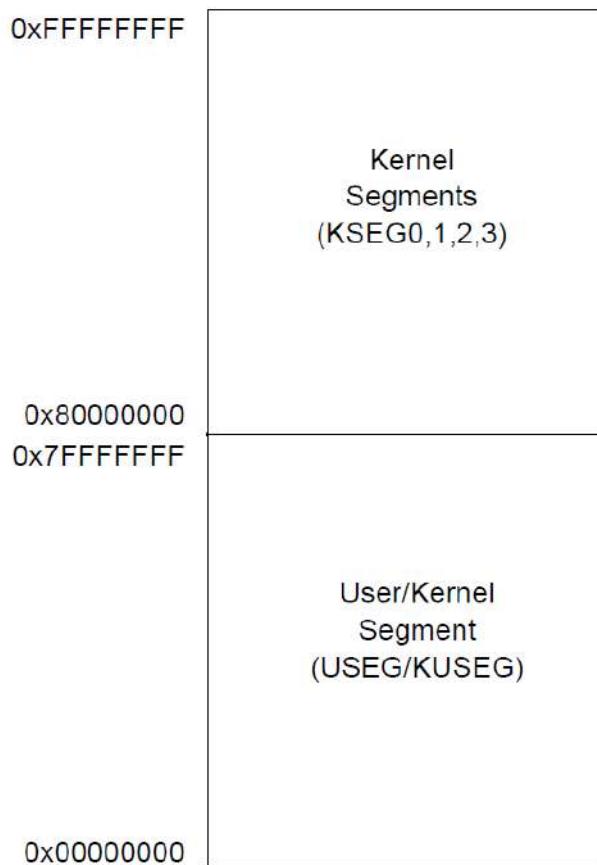
**FIGURE 4-6: MEMORY MAP ON RESET FOR PIC32MX695F512H, PIC32MX695F512L,  
PIC32MX795F512H AND PIC32MX795F512L DEVICES**



La particularité de l'organisation est l'introduction d'un mapping physique et d'un mapping virtuel.

### 2.3.1. KSEG ET USEG

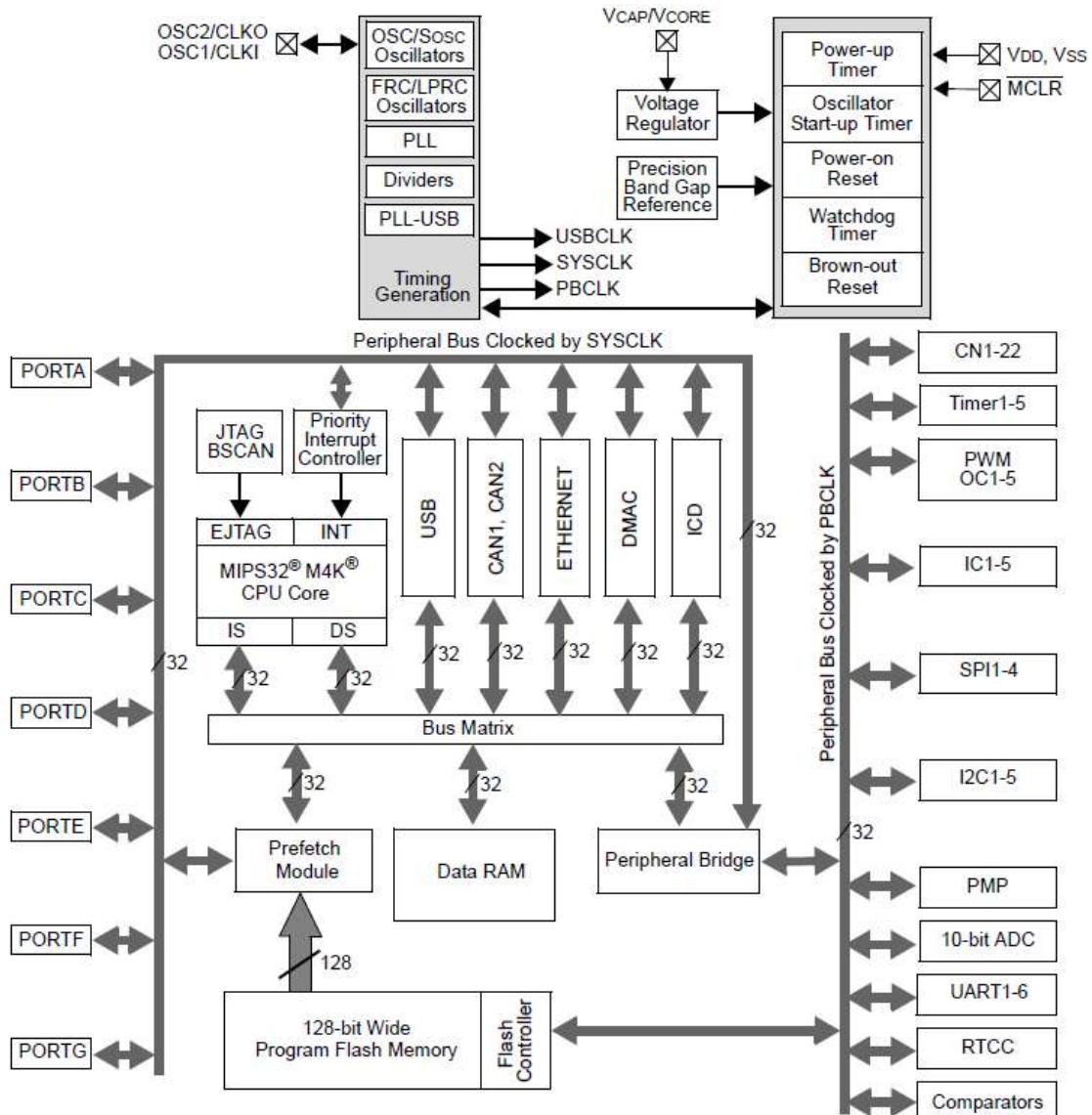
Les 4 GB de la mémoire virtuelle sont découpés en 2 parties de 2 GB.



La zone du kernel est divisée en 4 segments de 512 MB chacun, appelés KSEG0, KSEG1, KSEG2 et KSEG3. Seules les applications en mode kernel peuvent accéder à cet espace. La zone du kernel comprend tous les registres des périphériques, ce qui a pour conséquence que seules les applications en mode kernel peuvent accéder aux périphériques.

## 2.4. LES PERIPHERIQUES DU PIC32MX795F512L

Le diagramme ci-dessous permet de découvrir les différents éléments périphériques d'un PIC32MX795F512L.



Nous trouvons les éléments suivants :

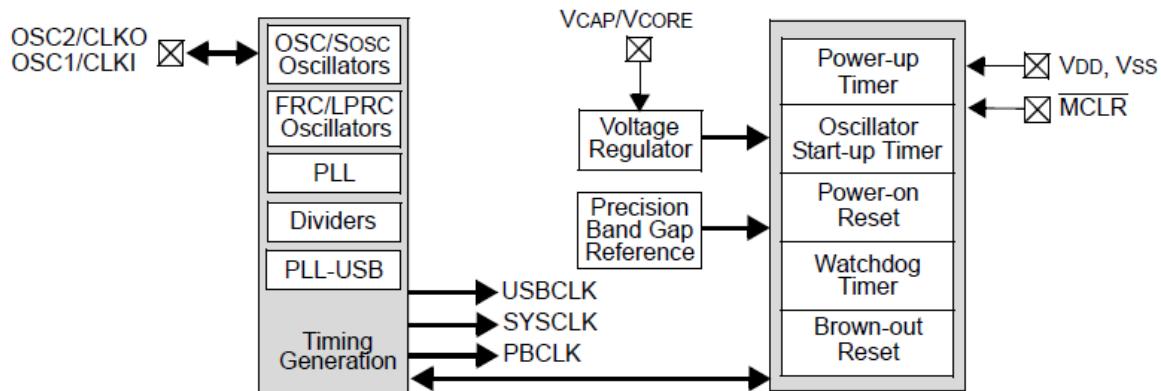
- 7 ports E/S. Les ports sont 16 bits interfacés sur un bus périphérique 32 bits.
- CN1-22 représente les CN (Change Notification Inputs).
- 5 timers. Il s'agit de compteurs 16 bits qui peuvent être utilisés par paire.
- PWM et OC1-5. 5 Output Compare permettant de générer des signaux PWM.
- IC1-5, 5 Input Capture permettant de capturer la valeur d'un timer sur un signal de déclenchement.
- SPI1-4. Possibilité de supporter 4 bus SPI.
- I2C1-5. Possibilité de supporter 5 bus I2C.
- PMP, (Parallel Master Output Port), il s'agit d'un port parallèle.

- ADC 10 bits, il s'agit d'un convertisseur analogique/digital avec une résolution de 10 bits. A disposition **16** entrées sélectionnables.
- UART1-6 **6** Universal Asynchronous Receiver Transmitter, permettant de réaliser des transmissions RS232 par exemple.
- RTCC Real Time Clock and Calendar.
- Comparators, il s'agit d'un module comparateur et tension de référence.

Les périphériques seront décrits en détail par la suite dans des chapitres séparés, en traitant leur structure et comment les programmer en langage C en utilisant les fonctions de la Peripheral Library correspondante.

## 2.5. OSCILLATEUR ET CIRCUIT DE RESET

Le bloc ci-dessous, extrait du schéma global, nous donne les indications sur la connexion de l'oscillateur, ainsi que les éléments internes pour la gestion du démarrage (Power-up) et du reset.



### 2.5.1. L'OSCILLATEUR

Source : IC32 Family Reference Manual, Sect. 06 Oscillators.pdf

#### 2.5.1.1. LES MODES DE L'OSCILLATEUR

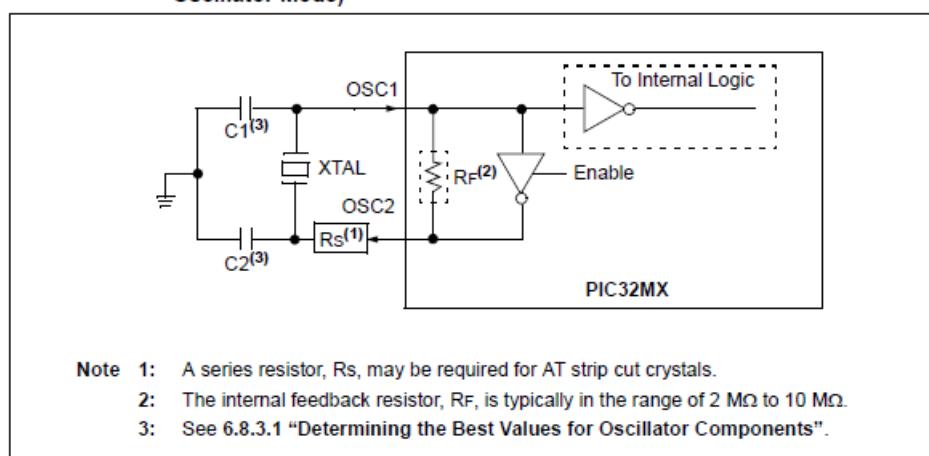
Table 6-4: Primary Oscillator Operating Modes

Oscillator Mode	Description
HS	High-speed crystal
XT	Resonator, crystal or resonator
EC	External clock input
HSPLL	Crystal, PLL enabled
XTPLL	Crystal resonator, PLL enabled
ECPLL	External clock input, PLL enabled

Note: The clock applied to the CPU, after applicable prescalers, postscalers, and PLL multipliers, must not exceed the maximum allowable processor frequency.

#### 2.5.1.2. CONFIGURATION AVEC OSCILLATEUR À QUARTZ

Figure 6-2: Crystal or Ceramic Resonator Operation (XT, XTPLL, HS, or HSPLL Oscillator Mode)



### 2.5.1.3. SITUATION AVEC OSCILLATEUR EXTERNE

Figure 6-3: External Clock Input Operation with Clock-Out (EC, ECPLL Mode)

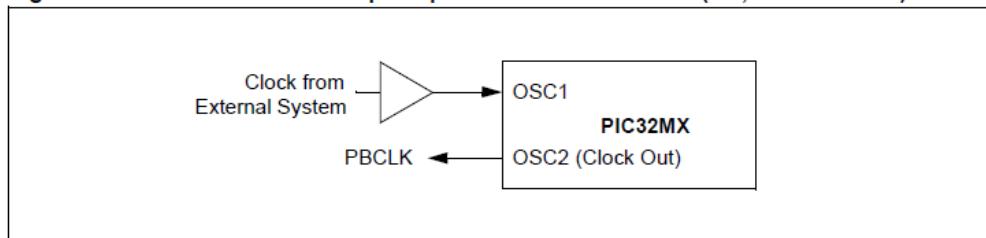
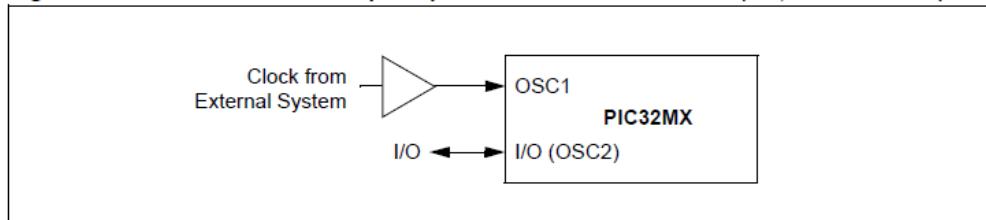
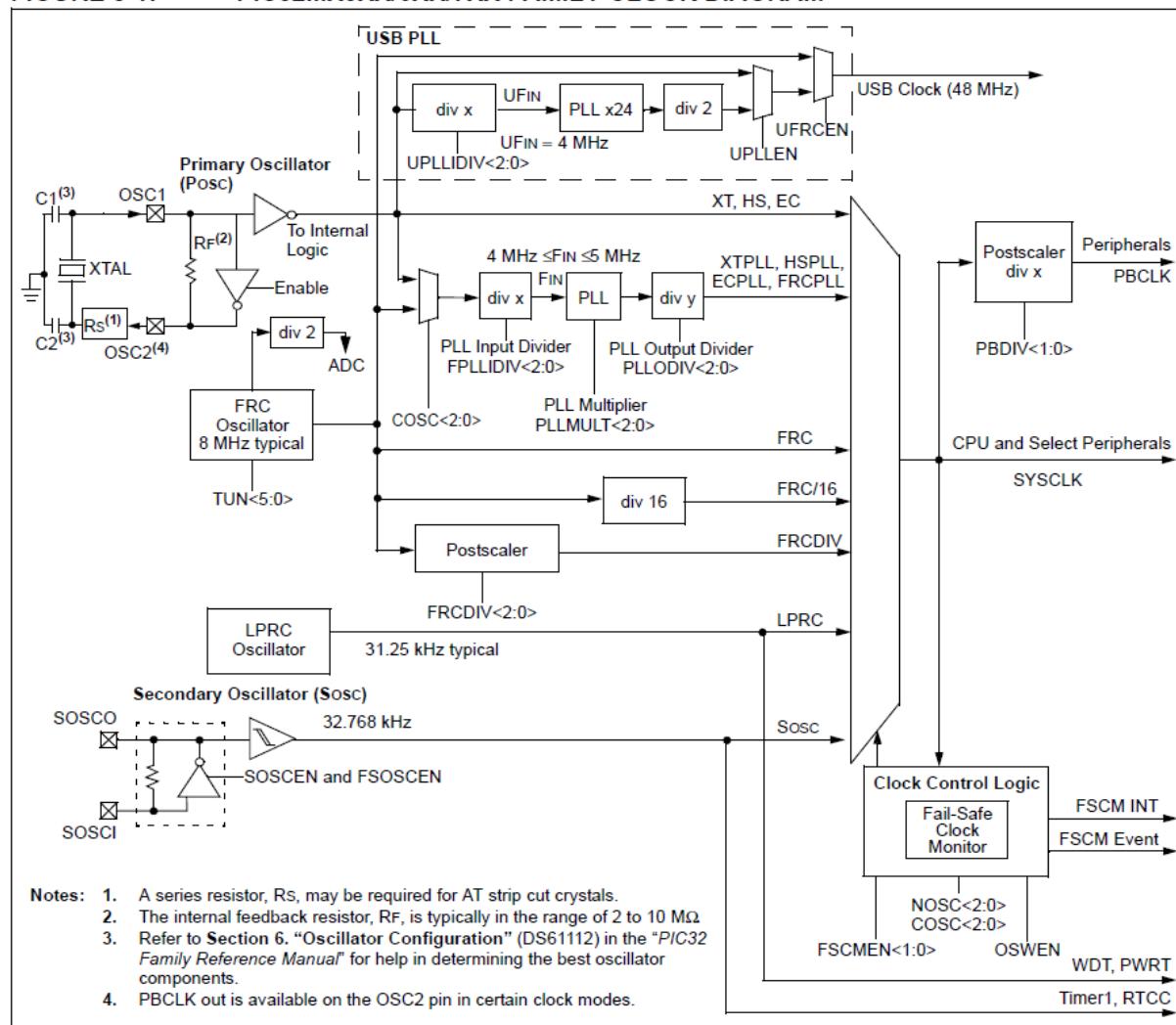


Figure 6-4: External Clock Input Operation with No Clock-Out (EC, ECPLL Mode)



### 2.5.1.4. DÉTAIL DU SYSTÈME DE GÉNÉRATION DES HORLOGES

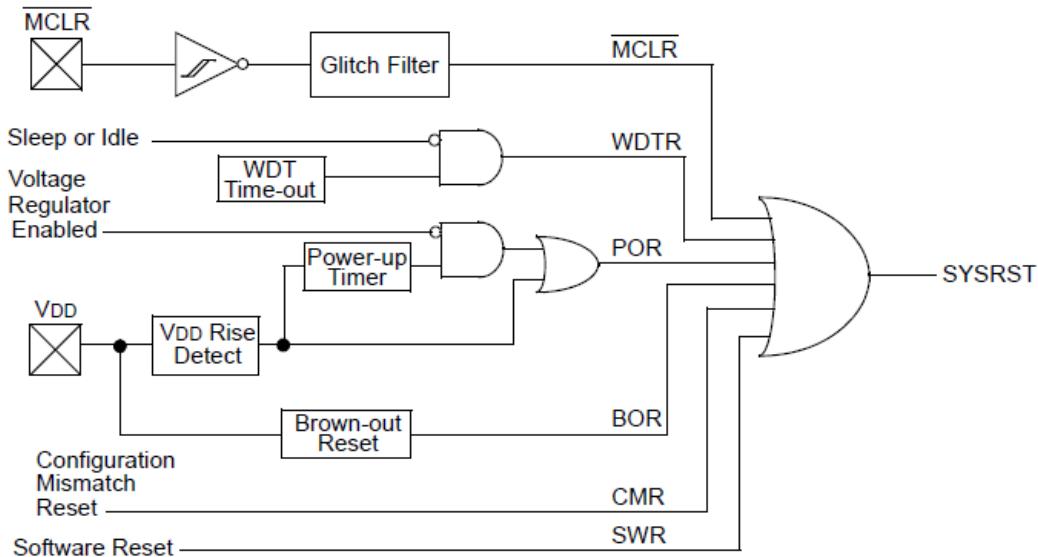
FIGURE 8-1: PIC32MX5XX/6XX/7XX FAMILY CLOCK DIAGRAM



### 2.5.2. CIRCUIT DE RESET

Les PIC32MX possèdent une circuiterie qui permet de générer un Reset sur un certain nombre de conditions.

Le schéma suivant montre le principe de cette circuiterie :



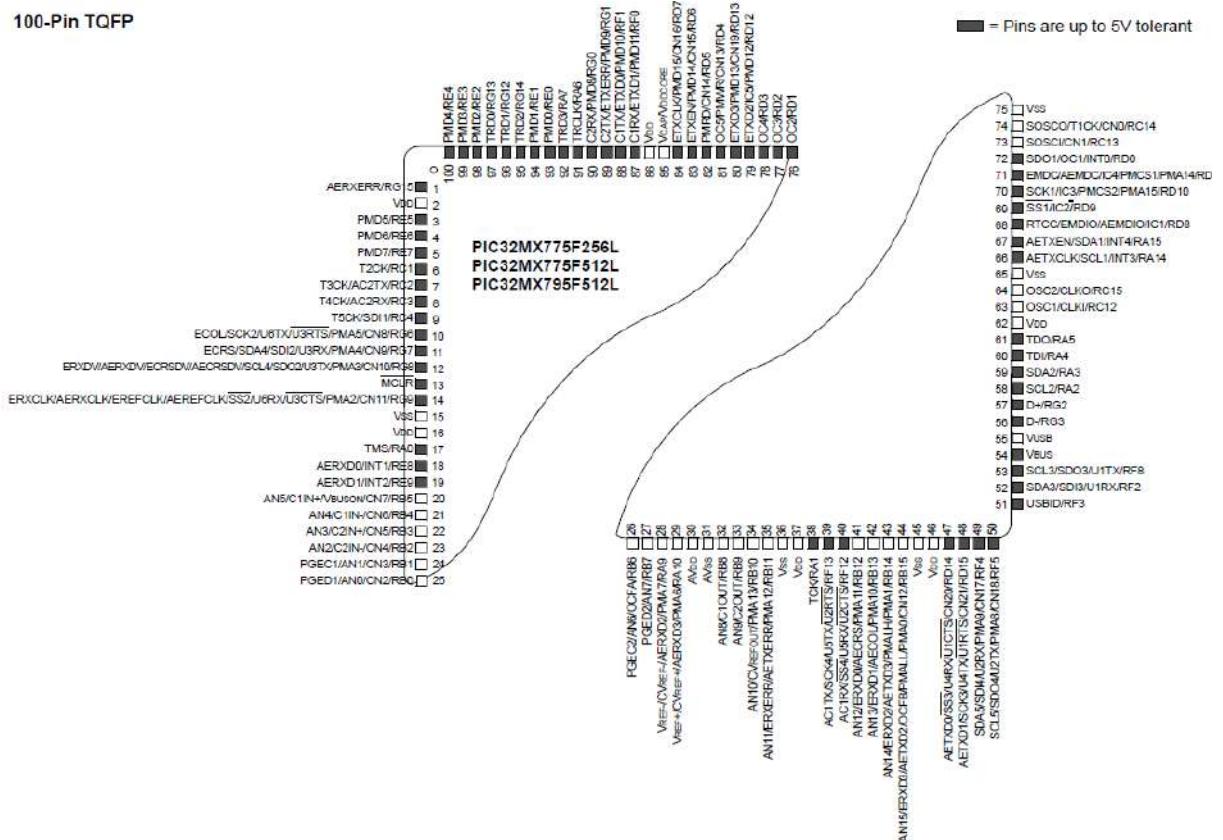
On observe plusieurs sources pour produire le SYSRST (System Reset) :

- Le Reset externe (/MCLR),
- Le Reset par le watchdog,
- La gestion du Reset à l'encclenchement (POR = Power On Reset),
- Le Reset du Brown-Out (tension trop faible),
- Le reset suite à un conflit de configuration,
- Le reset software.

## 2.6. ANNEXE A – LISTE E/S DU PIC32MX795F512L

### 2.6.1. BOITIER 100 PIN TQFP

100-Pin TQFP



### 2.6.2. LISTE DES E/S

Tous les bits de tous les ports ne sont pas présents. Cela dépend du modèle de PIC choisi.

#### 2.6.2.1. PORT A

Nom complet	No pin	Rôles particuliers
TMS/RA0	17	
TCK/RA1	38	
SCL2/RA2	58	I2C 2 serial clock
SDA2/RA3	59	I2C 2 serial data
TDI/RA4	60	
TDO/RA5	61	
TRCLK/RA6	91	
TRD3/RA7	92	
<i>RA8 absent sur ce modèle de PIC</i>		
Vref-/CVref-/AERXD2/PMA7/RA9	28	
Vref+/CVref+/AERXD3/PMA6/RA10	29	
<i>RA11 absent sur ce modèle de PIC</i>		
<i>RA12 absent sur ce modèle de PIC</i>		
<i>RA13 absent sur ce modèle de PIC</i>		
AETXCLK/SCL1/INT3/RA14	66	
AETXEN/SDA1/INT4/RA15	67	

### 2.6.2.1. PORT B (ENTRÉES ANALOGIQUES)

Nom complet	No pin	Rôles particuliers
PGED1/AN0/CN2/RB0	25	
PGEC1/AN1/CN3/RB1	24	
AN2/C2IN-/CN4/RB2	23	
AN3/C2IN+/CN5/RB3	22	
AN4/C1IN-/CN6/RB4	21	
AN5/C1IN+/Vbuson/CN7/RB5	20	
PGEC2/AN6/OCFA/RB6	26	Pour ICD3
PGED2/AN7/RB7	27	Pour ICD3
AN8/C1OUT/RB8	32	
AN9/C2OUT/RB9	33	
AN10/CVrefout/PMA13/RB10	34	
AN11/ERXERR/AETXER/PMA12/RB11	35	
AN12/ERXD0/AECRS/PMA11/RB12	41	
AN13/ERXD1/AECOL/PMA10/RB13	42	
AN14/ERXD2/AETXD3/PMALH/PMA1/RB14	43	
AN15/ERXD3/AETXD2/OCFB/PMALL /PMA0/CN12/RB15	44	

### 2.6.2.1. PORT C

Nom complet	No pin	Rôles particuliers
T2CK/RC1	6	
T3CK/AC2TX/RC2	7	
T4CK/AC2RX/RC3	8	
T5CK/SDI1/RC4	9	SPI 1 SDI
<i>RC5 absent sur ce modèle de PIC</i>		
<i>RC6 absent sur ce modèle de PIC</i>		
<i>RC7 absent sur ce modèle de PIC</i>		
<i>RC8 absent sur ce modèle de PIC</i>		
<i>RC9 absent sur ce modèle de PIC</i>		
<i>RC10 absent sur ce modèle de PIC</i>		
<i>RC11 absent sur ce modèle de PIC</i>		
OSC1/CLK1/RC12	63	Clock uC
SOSCI/CN1/RC13	73	Clock RTC
SOSC0/T1CK/CN0/RC14	74	Clock RTC
OSC2/CLK0/RC15	64	Clock uC

### 2.6.2.1. PORT D

Nom complet	No pin	Rôles particuliers
SDO1/OC1/INT0/RD0	72	SPI 1 SDO
OC2/RD1	76	
OC3/RD2	77	
OC4/RD3	78	
OC5/PMWR/CN13/RD4	81	
PMRD/CN14/RD5	82	
ETXEN/PMD14/CN15/RD6	83	
ETXCLK/PMD15/CN16/RD7	84	
RTCC/EMDIO/AEMDIO/IC1/RD8	68	
_SS1/IC2/RD9	69	
SCK1/IC3/PMCS2/PMA15/RD10	70	SCK 1 (SPI ou I2C)
EMDC/AEMDC/IC4/PMCS1/PMA14/RD11	71	
ETXD2/IC5/PMD12/RD12	79	
ETXD3/PMD13/CN19/RD13	80	
AETXD0/_SS3/U4RX/_U1CTS/CN20/RD14	47	Uart 4 RX
AETXD1/SCK3/U4TX/_U1RTS/CN21/RD15	48	Uart 4 TX / SCK 3

### 2.6.2.1. PORT E

Nom complet	No pin	Rôles particuliers
PMD0/RE0	93	
PMD1/RE1	94	
PMD2/RE2	98	
PMD3/RE3	99	
PMD4/RE4	100	
PMD5/RE5	3	
PMD6/RE6	4	
PMD7/RE7	5	
AERXD0/INT1/RE8	18	
AERXD1/INT2/RE9	19	
<i>RE10 absent sur ce modèle de PIC</i>		
<i>RE11 absent sur ce modèle de PIC</i>		
<i>RE12 absent sur ce modèle de PIC</i>		
<i>RE13 absent sur ce modèle de PIC</i>		
<i>RE14 absent sur ce modèle de PIC</i>		
<i>RE15 absent sur ce modèle de PIC</i>		

### 2.6.2.1. PORT F (UART SPI I2C)

Nom complet	No pin	Rôles particuliers
C1TX/ETXD1/PMD11/RF0	87	
C1RX/ETXD0/PMD10/RF1	88	
SDA3/SDI3/U1RX/RF2	52	Uart 1 RX/ I2C 3/ SPI 3
USBID/RF3	51	
SDA5/SDI4/U2RX/PMA9/CN17/RF4	49	Uart 2 RX/ I2C 5 / SPI 4
SCL5/SDO4/U2TX/PMA8/CN18/RF5	50	Uart 2 TX/ I2C 5 / SPI 4
<i>RF6 absent sur ce modèle de PIC</i>		
<i>RF7 absent sur ce modèle de PIC</i>		
SCL3/SDO3/U1TX/RF8	53	Uart 1 TX/ I2C 3/ SPI 3
<i>RF9 absent sur ce modèle de PIC</i>		
<i>RF10 absent sur ce modèle de PIC</i>		
<i>RF11 absent sur ce modèle de PIC</i>		
AC1RX/_SS4/U5RX/_U2CTS/RF12	40	Uart 5 RX / SPI4_SS
AC1TX/SCK4/U5TX/_U2RTS/RF13	39	Uart 5 TX / SPI4 CLK
<i>RF14 absent sur ce modèle de PIC</i>		
<i>RF15 absent sur ce modèle de PIC</i>		

### 2.6.2.1. PORT G

Nom complet	No pin	Rôles particuliers
C2RX/PMD8/RG0	90	
C2TX/ETXERR/PMD9/RG1	89	
D+/RG2	57	USB D+
D-/RG3	56	USB D-
<i>RG4 absent sur ce modèle de PIC</i>		
<i>RG5 absent sur ce modèle de PIC</i>		
ECOL/SCK2/U6TX/_U3RTS/PMA5/CN8/RG6	10	
ECRS/SDA4/SDI2/ <b>U3RX</b> /PMA4/CN9/RG7	11	Uart 3 RX / I2C 4 / SPI 2
ERXDV/AERXDV/ECRSDV/AECSRSDV	12	Uart 3 TX
SCL4/SDO2/ <b>U3TX</b> /PMA3/CN10/RG8		
ERXCLK/AERXCLK/EREFCLK/AEREFCCLK <u>_SS2/U6RX/_U3CTS/PMA2/CN11/RG9</u>	14	Uart 6 RX
<i>RG10 absent sur ce modèle de PIC</i>		
<i>RG11 absent sur ce modèle de PIC</i>		
TRD1/RG12	96	
TRD0/RG13	97	
TRD2/RG14	95	
<i>RG15 absent sur ce modèle de PIC</i>		

### 2.6.3. LISTE DES CN (CHANGE NOTIFICATION), ORDRE DES PORTS

Voici la liste des 22 CN (CN0-CN21), extraits des tableaux précédents en allant du port B au port G, le port A n'en disposant pas.

Nom complet	No pin	Rôles particuliers
PGED1/AN0/ <b>CN2</b> /RB0	25	
PGEC1/AN1/ <b>CN3</b> /RB1	24	
AN2/C2IN-/CN4/RB2	23	
AN3/C2IN+/CN5/RB3	22	
AN4/C1IN-/CN6/RB4	21	
AN5/C1IN+/Vbuson/ <b>CN7</b> /RB5	20	
AN15/ERXD3/AETXD2/OCFB/PMALL /PMA0/ <b>CN12</b> /RB15	44	
SOSCI/CN1/RC13	73	Clock RTC
SOSC0/T1CK/ <b>CN0</b> /RC14	74	Clock RTC
OC5/PMWR/ <b>CN13</b> /RD4	81	
PMRD/CN14/RD5	82	
ETXEN/PMD14/ <b>CN15</b> /RD6	83	
ETXCLK/PMD15/ <b>CN16</b> /RD7	84	
ETXD3/PMD13/ <b>CN19</b> /RD13	80	
AETXD0/_SS3/U4RX/_U1CTS/ <b>CN20</b> /RD14	47	Uart 4 RX
AETXD1/SCK3/U4TX/_U1RTS/ <b>CN21</b> /RD15	48	Uart 4 TX / SCK 3
SDA5/SDI4/U2RX/PMA9/ <b>CN17</b> /RF4	49	Uart 2 RX/I2C 5 /SPI 4
SCL5/SDO4/U2TX/PMA8/ <b>CN18</b> /RF5	50	Uart 2 TX/I2C 5 /SPI 4
ECOL/SCK2/U6TX/_U3RTS/PMA5/ <b>CN8</b> /RG6	10	
ECRS/SDA4/SDI2/ <b>U3RX</b> /PMA4/ <b>CN9</b> /RG7	11	Uart 3 RX/ I2C 4/SPI 2
ERXDV/AERXDV/ECRSDV/AECRSDV	12	Uart 3 TX
SCL4/SDO2/ <b>U3TX</b> /PMA3/ <b>CN10</b> /RG8		
ERXCLK/AERXCLK/EREFCLK/AEREFCCLK _SS2/ <b>U6RX</b> _U3CTS/PMA2/ <b>CN11</b> /RG9	14	Uart 6 RX

### 2.6.1. LISTE DES CN (CHANGE NOTIFICATION)

Voici la liste des 22 CN de CN0 à CN21.

CNx	Nom complet de la pin	No pin	Rôles particuliers
CN0	SOSC0/T1CK/ <b>CN0</b> /RC14	74	Clock RTC
CN1	SOSCI/ <b>CN1</b> /RC13	73	Clock RTC
CN2	PGED1/AN0/ <b>CN2</b> /RB0	25	
CN3	PGEC1/AN1/ <b>CN3</b> /RB1	24	
CN4	AN2/C2IN-/CN4/RB2	23	
CN5	AN3/C2IN+/ <b>CN5</b> /RB3	22	
CN6	AN4/C1IN-/CN6/RB4	21	
CN7	AN5/C1IN+/Vbuson/ <b>CN7</b> /RB5	20	
CN8	ECOL/SCK2/U6TX/_U3RTS/PMA5/ <b>CN8</b> /RG6	10	
CN9	ECRS/SDA4/SDI2/ <b>U3RX</b> /PMA4/ <b>CN9</b> /RG7	11	Uart 3 RX/ I2C 4/SPI 2
CN10	ERXDV/AERXDV/ECRSDV/AECRSDV SCL4/SDO2/ <b>U3TX</b> /PMA3/ <b>CN10</b> /RG8	12	Uart 3 TX
CN11	ERXCLK/AERXCLK/EREFCLK/AEREFCLK _SS2/ <b>U6RX</b> _U3CTS/PMA2/ <b>CN11</b> /RG9	14	Uart 6 RX
CN12	AN15/ERXD3/AETXD2/OCFB/PMALL /PMA0/ <b>CN12</b> /RB15	44	
CN13	OC5/PMWR/ <b>CN13</b> /RD4	81	
CN14	PMRD/ <b>CN14</b> /RD5	82	
CN15	ETXEN/PMD14/ <b>CN15</b> /RD6	83	
CN16	ETXCLK/PMD15/ <b>CN16</b> /RD7	84	
CN17	SDA5/SDI4/U2RX/PMA9/ <b>CN17</b> /RF4	49	Uart 2 RX/I2C 5 /SPI 4
CN18	SCL5/SDO4/U2TX/PMA8/ <b>CN18</b> /RF5	50	Uart 2 TX/I2C 5 /SPI 4
CN19	ETXD3/PMD13/ <b>CN19</b> /RD13	80	
CN20	AETXD0/_SS3/U4RX/_U1CTS/ <b>CN20</b> /RD14	47	Uart 4 RX
CN21	AETXD1/SCK3/U4TX/_U1RTS/ <b>CN21</b> /RD15	48	Uart 4 TX / SCK 3

## 2.8. HISTORIQUE DES VERSIONS

### 2.8.1. VERSION 1.0 MARS 2014

Création, cette version est à compléter

### 2.8.2. VERSION 1.5 NOVEMBRE 2014

Ajout de la liste des E/S (annexe A). Saut à la version 1.5 pour cohérence avec la nouvelle version du cours.

### 2.8.3. VERSION 1.7 NOVEMBRE 2015

Saut à la version 1.7 pour cohérence avec la nouvelle version du cours.

### 2.8.4. VERSION 1.8 NOVEMBRE 2016

Adaptation du chemin de la documentation du Kit PIC32. Quelques retouches d'orthographe.

### 2.8.5. VERSION 1.8.1 DECEMBRE 2016

Ajout de la liste des CN (Change Notification).

### 2.8.6. VERSION 1.9 OCTOBRE 2017

Reprise et relecture par SCA.

Compléments pipeline et prefetch cache.

### 2.8.7. VERSION 1.91 DECEMBRE 2018

Relecture et changements mineurs "Annexe A – liste E/S du PIC32MX795F512L".

**MINF**  
**Programmation des PIC32MX**

# **Chapitre 3**

## **Jeu d'instructions**



## **Théorie PIC32MX**

**Christian HUBER (CHR)**  
**Serge CASTOLDI (SCA)**  
**Version 1.9 novembre 2017**



# CONTENU

<b>3. Jeu d'instructions du PIC32</b>	<b>3-1</b>
<b>3.1. Structure des instructions</b>	<b>3-1</b>
3.1.1. Instructions d'opération	3-1
3.1.2. Instructions d'opération	3-2
3.1.2.1. Opérations arithmétiques	3-2
3.1.2.2. Opérations de décalage et rotation	3-2
3.1.2.3. Opérations logiques	3-3
3.1.2.4. Opérations d'action conditionnelle	3-3
3.1.2.5. Opérations de multiplication et division	3-3
3.1.2.6. Opérations d'accès à l'accumulateur	3-4
3.1.3. Instructions de saut et branchements	3-4
3.1.3.1. Instructions de saut conditionnel	3-4
3.1.3.2. Instruction J (Jump)	3-4
3.1.3.3. Instruction JAL (Jump And Link)	3-5
3.1.3.4. Instruction JALR (Jump And Link Register)	3-5
3.1.3.5. Instruction JR (Jump Register)	3-5
3.1.3.6. Liste des instructions de saut et branchement	3-6
3.1.4. Instructions load/store	3-7
3.1.4.1. Instructions de load & Store	3-8
3.1.4.2. Instructions de RMW atomique	3-8
<b>3.2. Les registres du PIC32</b>	<b>3-9</b>
<b>3.3. MIPS32 Quick Reference</b>	<b>3-10</b>
<b>3.4. Conclusion</b>	<b>3-12</b>
<b>3.5. Historique des versions</b>	<b>3-12</b>
3.5.1. Version 1.0 janvier 2014	3-12
3.5.2. Version 1.5 novembre 2014	3-12
3.5.3. Version 1.7 novembre 2015	3-12
3.5.4. Version 1.8 novembre 2016	3-12
3.5.5. Version 1.9 novembre 2017	3-12



### 3. JEU D'INSTRUCTIONS DU PIC32

Ce chapitre traite du jeu d'instructions du PIC32.

Le concept du jeu d'instructions du PIC32 s'appuie sur le standard MIPS32. C'est pour cela que l'on ne trouve pas de description du jeu d'instructions dans la documentation spécifique au PIC32.

Le document intitulé "MIPS Architecture for Programmers, Volume II-A: The MIPS32 Instruction Set Manual", que l'on trouve sur le réseau sous ...\\PROJETS\\SLO\\1102x\_SK32MX775F512L\\Data\_sheets\\PIC32 Family Reference Manual, décrit en détail le jeu d'instructions.

#### 3.1. STRUCTURE DES INSTRUCTIONS

Les instructions du PIC32 sont codées sur 32 bits. L'organisation des 32 bits de l'instruction varie s'il s'agit d'une instruction réalisant une opération, un accès à la mémoire ou un branchement (saut).

##### 3.1.1. INSTRUCTIONS D'OPERATION

Pour comprendre l'organisation des instructions effectuant une opération arithmétique ou logique, voici l'exemple d'une instruction d'addition :

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL 000000	rs	rt	rd	0 00000	ADD 100000	

Le code de l'opération tient sur 6 bits. Au niveau des opérandes, 3 registres sont impliqués.

Le format de l'instruction est **ADD rd, rs, rt**

rd, rs et rt indiquent le numéro du GPR (General Purpose Register) impliqué dans l'opération, avec rd registre destination, rs registre source et rt registre temporaire.

L'action d'addition correspond à : **rd ← rs + rt**

### 3.1.2. INSTRUCTIONS D'OPERATION

#### 3.1.2.1. OPERATIONS ARITHMETIQUES

<i>ARITHMETIC OPERATIONS</i>			
ADD	Rd, Rs, Rt	$Rd = Rs + Rt$	(OVERFLOW TRAP)
ADDI	Rd, Rs, CONST16	$Rd = Rs + CONST16^{\pm}$	(OVERFLOW TRAP)
ADDIU	Rd, Rs, CONST16	$Rd = Rs + CONST16^{\pm}$	
ADDU	Rd, Rs, Rt	$Rd = Rs + Rt$	
CLO	Rd, Rs	$Rd = COUNTLEADINGONES(Rs)$	
CLZ	Rd, Rs	$Rd = COUNTLEADINGZEROS(Rs)$	
LA	Rd, LABEL	$Rd = ADDRESS(LABEL)$	
LI	Rd, IMM32	$Rd = IMM32$	
LUI	Rd, CONST16	$Rd = CONST16 \ll 16$	
MOVE	Rd, Rs	$Rd = Rs$	
NEG_U	Rd, Rs	$Rd = -Rs$	
SEB <sup>R2</sup>	Rd, Rs	$Rd = RS_{7:0}^{\pm}$	
SEH <sup>R2</sup>	Rd, Rs	$Rd = RS_{15:0}^{\pm}$	
SUB	Rd, Rs, Rt	$Rd = Rs - Rt$	(OVERFLOW TRAP)
SUBU	Rd, Rs, Rt	$Rd = Rs - Rt$	

#### 3.1.2.2. OPERATIONS DE DECALAGE ET ROTATION

<i>SHIFT AND ROTATE OPERATIONS</i>			
ROTR <sup>R2</sup>	Rd, Rs, BITS5	$Rd = RS_{BITS5-1:0} :: RS_{31:BITS5}$	
ROTRV <sup>R2</sup>	Rd, Rs, Rt	$Rd = RS_{RT4:0-1:0} :: RS_{31:RT4:0}$	
SLL	Rd, Rs, SHIFT5	$Rd = Rs \ll SHIFT5$	
SLLV	Rd, Rs, Rt	$Rd = Rs \ll RT_{4:0}$	
SRA	Rd, Rs, SHIFT5	$Rd = RS^{\pm} \gg SHIFT5$	
SRAV	Rd, Rs, Rt	$Rd = RS^{\pm} \gg RT_{4:0}$	
SRL	Rd, Rs, SHIFT5	$Rd = RS^{\oslash} \gg SHIFT5$	
SRLV	Rd, Rs, Rt	$Rd = RS^{\oslash} \gg RT_{4:0}$	

### 3.1.2.3. OPERATIONS LOGIQUES

LOGICAL AND BIT-FIELD OPERATIONS		
AND	RD, Rs, RT	$RD = RS \& RT$
ANDI	RD, Rs, CONST16	$RD = RS \& CONST16^\emptyset$
EXT <sup>R2</sup>	RD, Rs, P, S	$RS = RSP+S:1:P^\emptyset$
INS <sup>R2</sup>	RD, Rs, P, S	$RD_{P+S:1:P} = RS_{S:1:0}$
<u>NOP</u>		No-op
NOR	RD, Rs, RT	$RD = \sim(RS   RT)$
<u>NOT</u>	RD, Rs	$RD = \sim RS$
OR	RD, Rs, RT	$RD = RS   RT$
ORI	RD, Rs, CONST16	$RD = RS   CONST16^\emptyset$
WSBH <sup>R2</sup>	RD, Rs	$RD = RS_{23:16} :: RS_{31:24} :: RS_{7:0} :: RS_{15:8}$
XOR	RD, Rs, RT	$RD = RS \oplus RT$
XORI	RD, Rs, CONST16	$RD = RS \oplus CONST16^\emptyset$

### 3.1.2.4. OPERATIONS D'ACTION CONDITIONNELLE

CONDITION TESTING AND CONDITIONAL MOVE OPERATIONS		
MOVN	RD, Rs, RT	IF $RT \neq 0$ , $RD = RS$
MOVZ	RD, Rs, RT	IF $RT = 0$ , $RD = RS$
SLT	RD, Rs, RT	$RD = (RS^\pm < RT^\pm) ? 1 : 0$
SLTI	RD, Rs, CONST16	$RD = (RS^\pm < CONST16^\pm) ? 1 : 0$
SLTIU	RD, Rs, CONST16	$RD = (RS^\emptyset < CONST16^\emptyset) ? 1 : 0$
SLTU	RD, Rs, RT	$RD = (RS^\emptyset < RT^\emptyset) ? 1 : 0$

### 3.1.2.5. OPERATIONS DE MULTIPLICATION ET DIVISION

MULTIPLY AND DIVIDE OPERATIONS		
DIV	Rs, RT	$Lo = RS^\pm / RT^\pm; Hi = RS^\pm \text{ MOD } RT^\pm$
DIVU	Rs, RT	$Lo = RS^\emptyset / RT^\emptyset; Hi = RS^\emptyset \text{ MOD } RT^\emptyset$
MADD	Rs, RT	$Acc += RS^\pm \times RT^\pm$
MADDU	Rs, RT	$Acc += RS^\emptyset \times RT^\emptyset$
MSUB	Rs, RT	$Acc -= RS^\pm \times RT^\pm$
MSUBU	Rs, RT	$Acc -= RS^\emptyset \times RT^\emptyset$
MUL	RD, Rs, RT	$RD = RS^\pm \times RT^\pm$
MULT	Rs, RT	$Acc = RS^\pm \times RT^\pm$
MULTU	Rs, RT	$Acc = RS^\emptyset \times RT^\emptyset$

### 3.1.2.6. OPERATIONS D'ACCES A L'ACCUMULATEUR

ACCUMULATOR ACCESS OPERATIONS		
MFHI	R <sub>D</sub>	R <sub>D</sub> = H <sub>I</sub>
MFLO	R <sub>D</sub>	R <sub>D</sub> = L <sub>O</sub>
MTHI	R <sub>S</sub>	H <sub>I</sub> = R <sub>S</sub>
MTLO	R <sub>S</sub>	L <sub>O</sub> = R <sub>S</sub>

### 3.1.3. INSTRUCTIONS DE SAUT ET BRANCHEMENTS

#### 3.1.3.1. INSTRUCTIONS DE SAUT CONDITIONNEL

L'instruction BEQ (Branch on EQUAL) illustre bien ce type d'instruction :

31	26 25	21 20	16 15	0
	BEQ 000100	rs 6	rt 5	offset 5

Son mnémonique est : **BEQ rs, rt, offset** et son action est :

**if rs = rt then branch**

rs et rt spécifient un no de registre.

Les détails de l'exécution sont les suivants :

```

I:      target_offset ← sign_extend(offset || 02)
            condition ← (GPR[rs] = GPR[rt])
I+1:    if condition then
            PC ← PC + target_offset
            endif
  
```

#### 3.1.3.2. INSTRUCTION J (JUMP)

Voici le format de l'instruction J (jump)

31	26 25	0
	J 000010	instr_index 26

Mnémonique : **J target**

La valeur de instr\_index est décalée à gauche de 2 pour former une valeur 28 bits.

```

I:
I+1: PC ← PCGPRLEN..28 || instr_index || 02
  
```

Détails exécution :

### 3.1.3.3. INSTRUCTION JAL (JUMP AND LINK)

Voici le format de l'instruction JAL (jump and link), cette instruction correspond à un CALL (appel de routine).

31	26 25		0
JAL		instr_index	
000011			

6

26

**JAL target**

Mnémonique :

La valeur de instr\_index est décalée à gauche de 2 pour former une valeur 28 bits.

**I : GPR[31] ← PC + 8**Détails exécution : **I+1:PC ← PC<sub>GPRLEN..28</sub> || instr\_index || 0<sup>2</sup>**

### 3.1.3.4. INSTRUCTION JALR (JUMP AND LINK REGISTER)

Voici le format de l'instruction JALR (Jump And Link Register) :

31	26 25	21 20	16 15	11 10	6 5	0
SPECIAL		rs	0	rd	hint	JALR
000000			00000			001001

6

**JALR rs (rd = 31 implied)**Mnémoniques : **JALR rd, rs**Principe exécution : **rd ← return\_addr, PC ← rs**

Cette instruction effectue un CALL, l'adresse de destination est fournie par rs, tandis que l'adresse de retour est mémorisée dans rd.

### 3.1.3.5. INSTRUCTION JR (JUMP REGISTER)

Voici le format de l'instruction JR (Jump Register) :

31	26 25	21 20	11 10	6 5	0
SPECIAL		rs	0	hint	JR
000000			00 0000 0000		001000

6

5

10

5

6

Mnémoniques : **JR rs**Principe exécution : **PC ← rs**

### 3.1.3.6. LISTE DES INSTRUCTIONS DE SAUT ET BRANCHEMENT

<i>JUMPS AND BRANCHES (Note: ONE DELAY SLOT)</i>		
<u>B</u>	OFF18	PC += OFF18 <sup>±</sup>
<u>BAL</u>	OFF18	RA = PC + 8, PC += OFF18 <sup>±</sup>
BEQ	Rs, RT, OFF18	IF Rs = RT, PC += OFF18 <sup>±</sup>
<u>BEQZ</u>	Rs, OFF18	IF Rs = 0, PC += OFF18 <sup>±</sup>
BGEZ	Rs, OFF18	IF Rs ≥ 0, PC += OFF18 <sup>±</sup>
BGEZAL	Rs, OFF18	RA = PC + 8; IF Rs ≥ 0, PC += OFF18 <sup>±</sup>
BGTZ	Rs, OFF18	IF Rs > 0, PC += OFF18 <sup>±</sup>
BLEZ	Rs, OFF18	IF Rs ≤ 0, PC += OFF18 <sup>±</sup>
BLTZ	Rs, OFF18	IF Rs < 0, PC += OFF18 <sup>±</sup>
BLTZAL	Rs, OFF18	RA = PC + 8; IF Rs < 0, PC += OFF18 <sup>±</sup>
BNE	Rs, RT, OFF18	IF Rs ≠ RT, PC += OFF18 <sup>±</sup>
<u>BNEZ</u>	Rs, OFF18	IF Rs ≠ 0, PC += OFF18 <sup>±</sup>
J	ADDR28	PC = PC <sub>31:28</sub> :: ADDR28 <sup>Ø</sup>
JAL	ADDR28	RA = PC + 8; PC = PC <sub>31:28</sub> :: ADDR28 <sup>Ø</sup>
JALR	Rd, Rs	Rd = PC + 8; PC = Rs
JR	Rs	PC = Rs

### 3.1.4. INSTRUCTIONS LOAD/STORE

Voici le format de l'instruction LW (Load Word) pour illustrer l'organisation de ce type d'instructions :

31	26 25	21 20	16 15	0
LW 100011	base 6	rt 5		offset 16

L'instruction s'écrit `LW rt, offset(base)`, et son action est la suivante :

`rt ← memory [base+offset]`

*base* correspond au no du registre contenant l'adresse de base. Pour atteindre la mémoire, il y a combinaison de l'adresse de base et de l'offset. La valeur lue est stockée dans le registre spécifié par *rt*.

Pour comparaison, voici l'instruction SW (Store Word) :

31	26 25	21 20	16 15	0
SW 101011	base 6	rt 5		offset 16

L'instruction s'écrit `SW rt, offset(base)` et son action est la suivante :

`memory [base+offset] ← rt`

La valeur du registre spécifié par *rt* est transférée dans la mémoire à l'adresse obtenue par la combinaison de la valeur du registre spécifié par *base* et de l'*offset*.

### 3.1.4.1. INSTRUCTIONS DE LOAD & STORE

LOAD AND STORE OPERATIONS		
LB	RD, OFF16(Rs)	$RD = \text{MEM8}(Rs + OFF16^\pm)^\pm$
LBU	RD, OFF16(Rs)	$RD = \text{MEM8}(Rs + OFF16^\pm)^\emptyset$
LH	RD, OFF16(Rs)	$RD = \text{MEM16}(Rs + OFF16^\pm)^\pm$
LHU	RD, OFF16(Rs)	$RD = \text{MEM16}(Rs + OFF16^\pm)^\emptyset$
LW	RD, OFF16(Rs)	$RD = \text{MEM32}(Rs + OFF16^\pm)$
LWL	RD, OFF16(Rs)	$RD = \text{LOADWORDLEFT}(Rs + OFF16^\pm)$
LWR	RD, OFF16(Rs)	$RD = \text{LOADWORDRIGHT}(Rs + OFF16^\pm)$
SB	Rs, OFF16(Rt)	$\text{MEM8}(Rt + OFF16^\pm) = RS_{7:0}$
SH	Rs, OFF16(Rt)	$\text{MEM16}(Rt + OFF16^\pm) = RS_{15:0}$
SW	Rs, OFF16(Rt)	$\text{MEM32}(Rt + OFF16^\pm) = RS$
SWL	Rs, OFF16(Rt)	$\text{STOREWORDLEFT}(Rt + OFF16^\pm, Rs)$
SWR	Rs, OFF16(Rt)	$\text{STOREWORDRIGHT}(Rt + OFF16^\pm, Rs)$
<u>ULW</u>	RD, OFF16(Rs)	$RD = \text{UNALIGNED\_MEM32}(Rs + OFF16^\pm)$
<u>USW</u>	Rs, OFF16(Rt)	$\text{UNALIGNED\_MEM32}(Rt + OFF16^\pm) = RS$

### 3.1.4.2. INSTRUCTIONS DE RMW ATOMIQUE

Ces 2 instructions sont appairées. Utilisées judicieusement ensemble, elles permettent des opérations, de lecture-modification-écriture (RMW : Read-Modify-Write) atomiques :

ATOMIC READ-MODIFY-WRITE OPERATIONS		
LL	RD, OFF16(Rs)	$RD = \text{MEM32}(Rs + OFF16^\pm); \text{LINK}$
SC	RD, OFF16(Rs)	IF ATOMIC, $\text{MEM32}(Rs + OFF16^\pm) = RD;$ $RD = \text{ATOMIC} ? 1 : 0$

### 3.2. LES REGISTRES DU PIC32

Pour comprendre les choix possibles, voici le principe d'utilisation des 32 GPR (General Purpose Registers) :

Registers		
0	zero	Always equal to zero
1	at	Assembler temporary; used by the assembler
2-3	v0-v1	Return value from a function call
4-7	a0-a3	First four parameters for a function call
8-15	t0-t7	Temporary variables; need not be preserved
16-23	s0-s7	Function variables; must be preserved
24-25	t8-t9	Two more temporary variables
26-27	k0-k1	Kernel use registers; may change unexpectedly
28	gp	Global pointer
29	sp	Stack pointer
30	fp/s8	Stack frame pointer or subroutine variable
31	ra	Return address of the last subroutine call

### 3.3. MIPS32 QUICK REFERENCE

Jumps And Branches (Note: One Delay Slot)					
AND	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = R <sub>S</sub> & R <sub>T</sub>	B	off18	PC += off18 <sup>t</sup>
ANDI	R <sub>D</sub> , R <sub>S</sub> , CONST16	R <sub>D</sub> = R <sub>S</sub> & CONST16 <sup>o</sup>	BAL	off18	R <sub>A</sub> = PC + 8, PC += off18 <sup>t</sup>
EXT <sup>32</sup>	R <sub>D</sub> , R <sub>S</sub> , P, S	R <sub>D</sub> = R <sub>S</sub> & P <sup>Q</sup>	BEQ	R <sub>S</sub> , R <sub>T</sub> , off18	IF R <sub>S</sub> = R <sub>T</sub> , PC += off18 <sup>t</sup>
IN\$ <sup>32</sup>	R <sub>D</sub> , R <sub>S</sub> , P, S	R <sub>D</sub> = R <sub>S</sub> & P <sup>Q</sup>	BEQZ	R <sub>S</sub> , off18	IF R <sub>S</sub> = 0, PC += off18 <sup>t</sup>
NOP		No-OP	BGEZ	R <sub>S</sub> , off18	IF R <sub>S</sub> ≥ 0, PC += off18 <sup>t</sup>
NOR	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = ~R <sub>S</sub>   R <sub>T</sub>	BGEZAL	R <sub>S</sub> , off18	R <sub>A</sub> = PC + 8, IF R <sub>S</sub> ≥ 0, PC += off18 <sup>t</sup>
NOT	R <sub>D</sub> , R <sub>S</sub>	R <sub>D</sub> = ~R <sub>S</sub>	BGTZ	R <sub>S</sub> , off18	IF R <sub>S</sub> > 0, PC += off18 <sup>t</sup>
OR	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = R <sub>S</sub>   R <sub>T</sub>	BLEZ	R <sub>S</sub> , off18	IF R <sub>S</sub> ≤ 0, PC += off18 <sup>t</sup>
ORI	R <sub>D</sub> , R <sub>S</sub> , CONST16	R <sub>D</sub> = R <sub>S</sub>   CONST16 <sup>o</sup>	BLTZ	R <sub>S</sub> , off18	IF R <sub>S</sub> < 0, PC += off18 <sup>t</sup>
WSBHF <sup>2</sup>	R <sub>D</sub> , R <sub>S</sub>	R <sub>D</sub> = R <sub>S</sub> 16 :: R <sub>S</sub> 24 :: R <sub>S</sub> 32 :: R <sub>S</sub> 48	BLTZAL	R <sub>S</sub> , off18	R <sub>A</sub> = PC + 8, IF R <sub>S</sub> < 0, PC += off18 <sup>t</sup>
XOR	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = R <sub>S</sub> ⊕ R <sub>T</sub>	BNE	R <sub>S</sub> , R <sub>T</sub> , off18	IF R <sub>S</sub> ≠ R <sub>T</sub> , PC += off18 <sup>t</sup>
XORI	R <sub>D</sub> , R <sub>S</sub> , CONST16	R <sub>D</sub> = R <sub>S</sub> ⊕ CONST16 <sup>o</sup>	BNEZ	R <sub>S</sub> , off18	IF R <sub>S</sub> ≠ 0, PC += off18 <sup>t</sup>
J			ADD	ADD128	PC = PC <sub>128</sub> :: ADD128 <sup>o</sup>
			JAL	ADD28	R <sub>A</sub> = PC + 8, PC = PC <sub>31:28</sub> :: ADDR28 <sup>o</sup>
			JALR	R <sub>D</sub> , R <sub>S</sub>	R <sub>D</sub> = PC + 8, PC = R <sub>S</sub>
			JR	R <sub>S</sub>	PC = R <sub>S</sub>
Load And Store Operations					
MOVN	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	IF R <sub>T</sub> ≠ 0, R <sub>D</sub> = R <sub>S</sub>	LB	R <sub>D</sub> , off16(R <sub>S</sub> )	RD = MEM8(R <sub>S</sub> + off16 <sup>t</sup> ) <sup>c</sup>
MOVZ	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	IF R <sub>T</sub> = 0, R <sub>D</sub> = R <sub>S</sub>	LBU	R <sub>D</sub> , off16(R <sub>S</sub> )	RD = MEM8(R <sub>S</sub> + off16 <sup>t</sup> ) <sup>c</sup>
SLT	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = (R <sub>S</sub> <sup>z</sup> < R <sub>T</sub> <sup>z</sup> ) ? 1 : 0	LH	R <sub>D</sub> , off16(R <sub>S</sub> )	RD = MEM16(R <sub>S</sub> + off16 <sup>t</sup> ) <sup>c</sup>
SLTI	R <sub>D</sub> , R <sub>S</sub> , CONST16	R <sub>D</sub> = (R <sub>S</sub> <sup>z</sup> < CONST16 <sup>o</sup> ) ? 1 : 0	LD	R <sub>D</sub> , off16(R <sub>S</sub> )	RD = MEM16(R <sub>S</sub> + off16 <sup>t</sup> ) <sup>c</sup>
SLTU	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = (R <sub>S</sub> <sup>z</sup> < R <sub>T</sub> <sup>z</sup> ) ? 1 : 0	LW	R <sub>D</sub> , off16(R <sub>S</sub> )	RD = MEM16(R <sub>S</sub> + off16 <sup>t</sup> ) <sup>c</sup>
MULT	R <sub>D</sub> , R <sub>S</sub>		SW	R <sub>S</sub> , off16(R <sub>T</sub> )	MEM32(R <sub>T</sub> + off16 <sup>t</sup> ) = RS <sub>T0</sub>
UUI	R <sub>D</sub> , CONST16	R <sub>D</sub> = CONST16 << 16	SWL	R <sub>S</sub> , off16(R <sub>T</sub> )	StoreWordL(R <sub>T</sub> + off16 <sup>t</sup> , RS <sub>T0</sub> )
MMOVE	R <sub>D</sub> , R <sub>S</sub>	R <sub>D</sub> = R <sub>S</sub>	SWR	R <sub>S</sub> , off16(R <sub>T</sub> )	StoreWordR(R <sub>T</sub> + off16 <sup>t</sup> , RS <sub>T0</sub> )
NEGJ	R <sub>D</sub> , R <sub>S</sub>	R <sub>D</sub> = -R <sub>S</sub>	ULW	R <sub>D</sub> , off16(R <sub>S</sub> )	RD = UNALIGNED_MEM32(R <sub>S</sub> + off16 <sup>t</sup> )
SEB <sup>32</sup>	R <sub>D</sub> , R <sub>S</sub>	R <sub>D</sub> = R <sub>S</sub> 7 <sup>z</sup>	USW	R <sub>S</sub> , off16(R <sub>T</sub> )	UNALIGNED_MEM32(R <sub>T</sub> + off16 <sup>t</sup> ) = RS <sub>T0</sub>
SEHF <sup>2</sup>	R <sub>D</sub> , R <sub>S</sub>	R <sub>D</sub> = R <sub>S</sub> 1 <sup>z</sup>			
SUB	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = R <sub>S</sub> - R <sub>T</sub>			
SUBU	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = R <sub>S</sub> - R <sub>T</sub>			
Shift And Rotate Operations					
ROTRE <sup>2</sup>	R <sub>D</sub> , R <sub>S</sub> , BTF <sub>5</sub>	R <sub>D</sub> = R <sub>S</sub> 31:19 :: R <sub>S</sub> 31:BTFS <sub>5</sub>	LL	R <sub>D</sub> , off16(R <sub>S</sub> )	RD = MEM32(R <sub>S</sub> + off16 <sup>t</sup> ); LLH:
ROTRV <sup>32</sup>	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = R <sub>S</sub> 31:19 :: R <sub>S</sub> 31:BTFS <sub>5</sub>	SC	R <sub>D</sub> , off16(R <sub>S</sub> )	IF ATOMIC, MEM32(R <sub>S</sub> + off16 <sup>t</sup> ) = RD;
SLL	R <sub>D</sub> , R <sub>S</sub> , SHIFT <sub>5</sub>	R <sub>D</sub> = R <sub>S</sub> << SHIFT <sub>5</sub>	MTLO	R <sub>S</sub>	RD = ATOMIC ? 1 : 0
SLLV	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = R <sub>S</sub> << RT <sub>40</sub>			
SRA	R <sub>D</sub> , R <sub>S</sub> , SHIFT <sub>5</sub>	R <sub>D</sub> = R <sub>S</sub> <sup>z</sup> >> SHIFT <sub>5</sub>			
SRAV	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = R <sub>S</sub> <sup>z</sup> >> RT <sub>40</sub>			
SRL	R <sub>D</sub> , R <sub>S</sub> , SHIFT <sub>5</sub>	R <sub>D</sub> = R <sub>S</sub> <sup>z</sup> >> SHIFT <sub>5</sub>			
SRLV	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = R <sub>S</sub> <sup>z</sup> >> RT <sub>40</sub>			

MIPS32® Instruction Set Quick Reference					
RD, RT	- DESTINATION REGISTER - SOURCE OPERAND REGISTER - RETURN ADDRESS REGISTER (R31) PC - PROGRAM COUNTER ACC - 64-BIT ACCUMULATOR L <sub>C</sub> , H <sub>I</sub> - ACCUMULATOR LOW (ACC <sub>31:0</sub> ) AND HIGH (ACC <sub>63:32</sub> ) PARTS ± - SIGNIFIED OPERAND OR SIGN EXTENSION Q - UNSIGNED OPERAND OR ZERO EXTENSION :: - CONCATENATION OF BIT FIELDS R2 - MIPS32 RELEASE 2 INSTRUCTION DOTTED - ASSEMBLER Pseudo-INSTRUCTION				
PLEASE REFER TO "MIPS32 ARCHITECTURE FOR PROGRAMMABLE PROCESSORS Volume II: THE MIPS32 Instruction Set" FOR COMPLETE INSTRUCTION SET INFORMATION.					
Arithmetic Operations					
ADD	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = R <sub>S</sub> + R <sub>T</sub> (OVERFLOW TRAP)	DIV	R <sub>D</sub> , R <sub>S</sub>	RD = R <sub>S</sub> / R <sub>T</sub> (OVERFLOW TRAP)
ADDI	R <sub>D</sub> , R <sub>S</sub> , CONST16	R <sub>D</sub> = R <sub>S</sub> + CONST16 <sup>z</sup>	DIVU	R <sub>D</sub> , R <sub>S</sub>	RD = R <sub>S</sub> / R <sub>T</sub> (OVERFLOW TRAP)
ADDIU	R <sub>D</sub> , R <sub>S</sub> , CONST16	R <sub>D</sub> = R <sub>S</sub> + CONST16 <sup>z</sup>	MADD	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	RD = (R <sub>S</sub> <sup>z</sup> < R <sub>T</sub> <sup>z</sup> ) ? 1 : 0
ADDU	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = R <sub>S</sub> + R <sub>T</sub>	MADDU	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	RD = (R <sub>S</sub> <sup>z</sup> < CONST16 <sup>o</sup> ) ? 1 : 0
CLZ	R <sub>D</sub> , R <sub>S</sub>	R <sub>D</sub> = COUNTLEADINGZEROES(R <sub>S</sub> )	MSUB	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	RD = R <sub>S</sub> - R <sub>T</sub>
CLZ	R <sub>D</sub> , R <sub>S</sub>	R <sub>D</sub> = COUNTLEADINGZEROES(R <sub>S</sub> )	MSUBU	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	RD = R <sub>S</sub> - R <sub>T</sub>
LA	R <sub>D</sub> , LABEL	R <sub>D</sub> = ADDRESS(LABEL)	MUL	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	RD = R <sub>S</sub> × R <sub>T</sub>
LJ	R <sub>D</sub> , JOM32	R <sub>D</sub> = JOM32	MULT	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	RD = R <sub>S</sub> × R <sub>T</sub>
LUI	R <sub>D</sub> , CONST16	R <sub>D</sub> = CONST16 << 16	MULTU	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	RD = R <sub>S</sub> × R <sub>T</sub>
MOVE	R <sub>D</sub> , R <sub>S</sub>	R <sub>D</sub> = R <sub>S</sub>	SUB	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	RD = R <sub>S</sub> - R <sub>T</sub>
NEGJ	R <sub>D</sub> , R <sub>S</sub>	R <sub>D</sub> = -R <sub>S</sub>	SUBU	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	RD = R <sub>S</sub> - R <sub>T</sub>
SEB <sup>32</sup>	R <sub>D</sub> , R <sub>S</sub>	R <sub>D</sub> = R <sub>S</sub> 7 <sup>z</sup>			
SEHF <sup>2</sup>	R <sub>D</sub> , R <sub>S</sub>	R <sub>D</sub> = R <sub>S</sub> 1 <sup>z</sup>			
SUB	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = R <sub>S</sub> - R <sub>T</sub>			
SUBU	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = R <sub>S</sub> - R <sub>T</sub>			
Multiply And Divide Operations					
DIV	R <sub>S</sub> , R <sub>T</sub>	RD = R <sub>S</sub> <sup>z</sup> / R <sub>T</sub> <sup>z</sup> ; HI = R <sub>S</sub> <sup>z</sup> MOD R <sub>T</sub> <sup>z</sup>	DIVU	R <sub>S</sub> , R <sub>T</sub>	RD = R <sub>S</sub> <sup>z</sup> / R <sub>T</sub> <sup>z</sup> ; HI = R <sub>S</sub> <sup>z</sup> MOD R <sub>T</sub> <sup>z</sup>
DIVU	R <sub>S</sub> , R <sub>T</sub>	RD = R <sub>S</sub> <sup>z</sup> / R <sub>T</sub> <sup>z</sup> ; HI = R <sub>S</sub> <sup>z</sup> MOD R <sub>T</sub> <sup>z</sup>	MADD	R <sub>S</sub> , R <sub>T</sub>	ACCC += R <sub>S</sub> <sup>z</sup> × R <sub>T</sub> <sup>z</sup>
MADD	R <sub>S</sub> , R <sub>T</sub>	ACCC += R <sub>S</sub> <sup>z</sup> × R <sub>T</sub> <sup>z</sup>	MADDU	R <sub>S</sub> , R <sub>T</sub>	ACCC += R <sub>S</sub> <sup>z</sup> × R <sub>T</sub> <sup>z</sup>
MADDU	R <sub>S</sub> , R <sub>T</sub>	ACCC += R <sub>S</sub> <sup>z</sup> × R <sub>T</sub> <sup>z</sup>	MSUB	R <sub>S</sub> , R <sub>T</sub>	ACCC -= R <sub>S</sub> <sup>z</sup> × R <sub>T</sub> <sup>z</sup>
MSUB	R <sub>S</sub> , R <sub>T</sub>	ACCC -= R <sub>S</sub> <sup>z</sup> × R <sub>T</sub> <sup>z</sup>	MSUBU	R <sub>S</sub> , R <sub>T</sub>	ACCC -= R <sub>S</sub> <sup>z</sup> × R <sub>T</sub> <sup>z</sup>
MUL	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	RD = R <sub>S</sub> <sup>z</sup> × R <sub>T</sub> <sup>z</sup>	MUL	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	RD = R <sub>S</sub> <sup>z</sup> × R <sub>T</sub> <sup>z</sup>
MULT	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	RD = R <sub>S</sub> <sup>z</sup> × R <sub>T</sub> <sup>z</sup>	MULTU	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	RD = R <sub>S</sub> <sup>z</sup> × R <sub>T</sub> <sup>z</sup>
Shift And Rotate Operations					
ROTRE <sup>2</sup>	R <sub>D</sub> , R <sub>S</sub> , BTFS <sub>5</sub>	R <sub>D</sub> = R <sub>S</sub> 31:19 :: R <sub>S</sub> 31:BTFS <sub>5</sub>	ROTRE <sup>32</sup>	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = R <sub>S</sub> 31:19 :: R <sub>S</sub> 31:BTFS <sub>5</sub>
ROTRV <sup>32</sup>	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = R <sub>S</sub> 31:19 :: R <sub>S</sub> 31:BTFS <sub>5</sub>	ROTRE <sup>32</sup>	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = R <sub>S</sub> 31:19 :: R <sub>S</sub> 31:BTFS <sub>5</sub>
SLL	R <sub>D</sub> , R <sub>S</sub> , SHIFT <sub>5</sub>	R <sub>D</sub> = R <sub>S</sub> << SHIFT <sub>5</sub>	SLL	R <sub>D</sub> , R <sub>S</sub> , SHIFT <sub>5</sub>	R <sub>D</sub> = R <sub>S</sub> << SHIFT <sub>5</sub>
SLLV	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = R <sub>S</sub> << RT <sub>40</sub>	SRAV	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = R <sub>S</sub> << RT <sub>40</sub>
SRA	R <sub>D</sub> , R <sub>S</sub> , SHIFT <sub>5</sub>	R <sub>D</sub> = R <sub>S</sub> <sup>z</sup> >> SHIFT <sub>5</sub>	SRL	R <sub>D</sub> , R <sub>S</sub> , SHIFT <sub>5</sub>	R <sub>D</sub> = R <sub>S</sub> <sup>z</sup> >> SHIFT <sub>5</sub>
SRAV	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = R <sub>S</sub> <sup>z</sup> >> RT <sub>40</sub>	SRLV	R <sub>D</sub> , R <sub>S</sub> , R <sub>T</sub>	R <sub>D</sub> = R <sub>S</sub> <sup>z</sup> >> RT <sub>40</sub>
Accumulator Access Operations					
MFHI	R <sub>D</sub>	RD = HI	MFLO	R <sub>D</sub>	RD = LO
MTHI	R <sub>S</sub>	HI = R <sub>S</sub>	MTHI	R <sub>S</sub>	HI = R <sub>S</sub>
MTLO	R <sub>S</sub>	LO = R <sub>S</sub>	MTLO	R <sub>S</sub>	LO = R <sub>S</sub>

		Atomic Read-Modify-Write Example																					
		<pre>atomic_inc:     ldi    \$t0, 0(\$a0)      # load linked     addiu \$t1, \$t0, 1        # increment     sc    \$t1, 0(\$a0)       # store cond'l     breqz \$t1, atomic_inc   # loop if failed     nop</pre>																					
		<p><i>Accessing Unaligned Data</i></p> <p><i>NOTE: ULW AND USW AUTOMATICALLY GENERATE APPROPRIATE CODE</i></p> <table border="1"> <thead> <tr> <th colspan="2">Little-Endian Mode</th> <th colspan="2">Big-Endian Mode</th> </tr> </thead> <tbody> <tr> <td>LWR</td><td>Rd, off16(Rs)</td><td>LWL</td><td>Rd, off16(Rs)</td></tr> <tr> <td>LWL</td><td>Rd, off16+3(Rs)</td><td>LWR</td><td>Rd, off16+3(Rs)</td></tr> <tr> <td>SWR</td><td>Rd, off16(Rs)</td><td>SWL</td><td>Rd, off16(Rs)</td></tr> <tr> <td>SWL</td><td>Rd, off16+3(Rs)</td><td>SWR</td><td>Rd, off16+3(Rs)</td></tr> </tbody> </table>		Little-Endian Mode		Big-Endian Mode		LWR	Rd, off16(Rs)	LWL	Rd, off16(Rs)	LWL	Rd, off16+3(Rs)	LWR	Rd, off16+3(Rs)	SWR	Rd, off16(Rs)	SWL	Rd, off16(Rs)	SWL	Rd, off16+3(Rs)	SWR	Rd, off16+3(Rs)
Little-Endian Mode		Big-Endian Mode																					
LWR	Rd, off16(Rs)	LWL	Rd, off16(Rs)																				
LWL	Rd, off16+3(Rs)	LWR	Rd, off16+3(Rs)																				
SWR	Rd, off16(Rs)	SWL	Rd, off16(Rs)																				
SWL	Rd, off16+3(Rs)	SWR	Rd, off16+3(Rs)																				
		<p><i>Accessing Unaligned Data From C</i></p> <pre>typedef struct {     int u; } __attribute__((packed)) unaligned;  int unaligned_load(void *ptr); void unaligned_store(void *ptr, int value);  void *unaligned_load(void *ptr) {     unaligned *uptr = (unaligned *)ptr;     return uptr-&gt;u; }</pre>																					

Reading The Cycle Counter Register From C		Assembly-Language Function Example	
		<pre>unsigned mips_cycle_counter_read() {     unsigned cc;     asm volatile("mfco %0, \$9" : "=r" (cc));     return (cc &lt;&lt; 1); }</pre>	
18-19 Two more temporary variables		<p><i>Assembly-Language Function Example</i></p> <pre># int asm_max(int a, int b) # { #     int r = (a &lt; b) ? b : a; #     return r; # } .text .set nomacro .set noexec .global asm_max .ent asm_max asm_max:     move \$v0, \$a0      # r = a     slt \$t0, \$a0, \$a1  # a &lt; b ?     jr \$ra, \$t0         # return     movn \$v0, \$a1, \$t0  # if yes, r = b .end .asm_max</pre>	
24-25 k0-k1 Kernel use registers, may change unexpectedly		<p><i>C/ Assembly-Language Function Interface</i></p> <pre>#include &lt;stdio.h&gt; int asm_max(int a, int b); int main() {     int x = asm_max(10, 100);     int y = asm_max(200, 20);     printf("%d %d\n", x, y); }</pre>	
26-27 18-19 Two more temporary variables		<p><i>MIPS SDE-GCC Compiler Defines</i></p> <pre>_mips             MIPS ISA (= 32 for MIPS32) _mips_isa_rev    MIPS ISA Revision (= 2 for MIPS32 R2) _mips_DSP        DSP ASE extensions enabled _MIPS_EB         Big-endian target CPU _MIPS_LSB        Little-endian target CPU _MIPS_ARCH_CPU  Target CPU specified by -march=CPU _MIPS_TUNE_CPU   Pipeline tuning selected by -mtune=CPU</pre>	

Registers		Stack Management		Function Parameters		Return Values		Notes	
0 zero	Always equal to zero								
1 at	Assembler temporary, used by the assembler								
2-3 v0-v1	Return value from a function call								
4-7 a0-a3	First four parameters for a function call								
8-15 t0-t7	Temporary variables, need not be preserved								
16-23 s0-s7	Function variables, must be preserved								
24-25 k0-k1	Kernel use registers, may change unexpectedly								
28 gp	Global pointer								
29 sp	Stack pointer								
30 fp:ss	Stack frame pointer or subroutine variable								
31 ra	Return address of the last subroutine call								

### **3.4. CONCLUSION**

Ce chapitre offre un bref aperçu de l'organisation du jeu d'instructions du PIC32. Il doit permettre à l'étudiant de pouvoir observer le code assembleur produit par le compilateur et parvenir à le comprendre dans les grandes lignes.

### **3.5. HISTORIQUE DES VERSIONS**

#### **3.5.1. VERSION 1.0 JANVIER 2014**

Création du document et découverte du jeu d'instructions MIPS32.

#### **3.5.2. VERSION 1.5 NOVEMBRE 2014**

Passage à la version 1.5 pour cohérence avec l'ensemble des chapitres. Pas de modifications liées à Harmony. Quelques retouches.

#### **3.5.3. VERSION 1.7 NOVEMBRE 2015**

Saut à la version 1.7 pour cohérence avec l'ensemble des chapitres. Pas de modifications liées à Harmony. Correction de la numérotation des titres.

#### **3.5.4. VERSION 1.8 NOVEMBRE 2016**

Saut à la version 1.8 pour cohérence avec l'ensemble des chapitres. Pas de modifications liées à Harmony. Modification du chemin de la documentation liée au Kit PIC32.

#### **3.5.5. VERSION 1.9 NOVEMBRE 2017**

Reprise et relecture par SCA.

**MINF**  
**Programmation des PIC32MX**

# **Chapitre 4**

## **Pile et sous-programmes**



## **Théorie PIC32MX**

**Christian HUBER (CHR)**  
**Serge CASTOLDI (SCA)**  
**Version 1.9 novembre 2017**



## CONTENU

<b>4. Pile et sous-programmes</b>	<b>4-1</b>
<b>4.1. Le mécanisme de la pile</b>	<b>4-1</b>
<b>4.2. Mécanisme de la pile du PIC32</b>	<b>4-2</b>
<b>4.3. Initialisation du Stack et du Heap</b>	<b>4-4</b>
4.3.1. Organisation du stack et du Heap	4-5
4.3.2. Les registres du PIC32	4-5
<b>4.4. PUSH AND POP INSTRUCTIONS</b>	<b>4-6</b>
4.4.1. Réalisation de l'action Push	4-6
4.4.2. Réalisation de l'action Pop	4-7
4.4.3. Action push et pop avec plusieurs valeurs	4-7
<b>4.5. Relation entre pile et fonctions</b>	<b>4-9</b>
4.5.1. Pile et Fonctions	4-9
4.5.2. Débordement de pile (Stack Overflow)	4-12
4.5.3. Implémentation dans le MIPS32	4-13
4.5.4. Leaf Procedures	4-13
<b>4.6. Mécanisme appel et passage des paramètres</b>	<b>4-14</b>
4.6.1. Principe Call et Return	4-14
4.6.2. Appels imbriqués	4-14
4.6.3. Appel et passage de paramètre	4-16
<b>4.7. Conclusion</b>	<b>4-20</b>
<b>4.8. Historique des versions</b>	<b>4-21</b>
4.8.1. Version 1.0 février 2014	4-21
4.8.2. Version 1.5 novembre 2014	4-21
4.8.3. Version 1.7 novembre 2015	4-21
4.8.4. Version 1.7bis novembre 2015	4-21
4.8.5. Version 1.8 novembre 2016	4-21
4.8.6. Version 1.9 novembre 2017	4-21



## 4. PILE ET SOUS-PROGRAMMES

Ce chapitre décrit le mécanisme de la pile, son usage lors des appels de sous-programmes, ainsi que les mécanismes utilisés par le compilateur C pour passer les paramètres aux sous-programmes.

Comme la gestion de la pile du PIC32MX est "software", on trouve en partie des informations sur la gestion de la pile dans la documentation du compilateur :

- Document MPLAB-XC32-Users-Guide.pdf (documentation du compilateur xc32).

Ce document contient 2 parties intéressantes :

- [7.4 Stack](#)
- [Chapter 15. Main, Runtime Start-up and Reset](#)

Ce chapitre se base sur le compilateur xc32 v1.42.

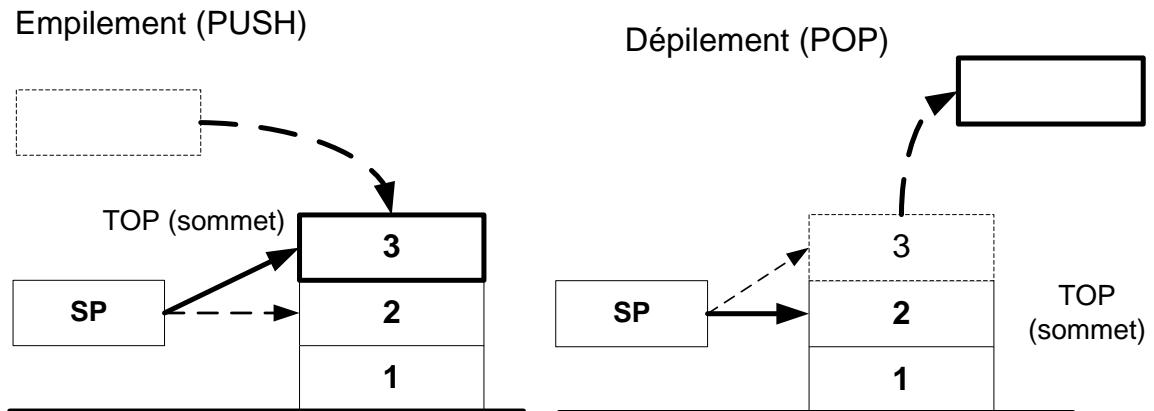
### 4.1. LE MÉCANISME DE LA PILE

Le mécanisme de pile (en anglais *stack*), s'appuie sur deux éléments :

- La zone de mémoire pour la pile (*stack space*),
- Le pointeur de pile (*stack pointer*).

La pile est principalement utilisée pour sauvegarder et restaurer les adresses de retour des sous-programmes.

Le principe de la pile est un stockage LIFO : Last In, First Out. C'est-à-dire comme une pile de carton, le premier élément que l'on peut reprendre est celui que l'on a empilé en dernier.



La dernière valeur placée sur la pile s'appelle le sommet (*top of stack*).

## 4.2. MÉCANISME DE LA PILE DU PIC32

Paragraphe 7.4 de la documentation du compilateur.

### 7.4 STACK

The PIC32 devices use what is referred to in this user's guide as a "software stack". This is the typical stack arrangement employed by most computers and is ordinary data memory accessed by a push-and-pop type instruction and a stack pointer register. The term "hardware stack" is used to describe the stack employed by Microchip 8-bit devices, which is only used for storing function return addresses.

The PIC32 devices use a dedicated stack pointer register `sp` (register 29) for use as a software Stack Pointer. All processor stack operations, including function calls, interrupts and exceptions, use the software stack. It points to the next free location on the stack. The stack grows downward, towards lower memory addresses.

By default, the size of the stack is 1024 bytes. The size of the stack can be changed by specifying the size on the linker command line using the `--defsym_min_stack_size` linker command line option. An example of allocating

a stack of 2048 bytes using the command line is:

```
xc32-gcc foo.c -Wl,--defsym,_min_stack_size=2048
```

The run-time stack grows downward from higher addresses to lower addresses. Two working registers are used to manage the stack:

- Register 29 (`sp`) – This is the Stack Pointer. It points to the next free location on the stack.
- Register 30 (`fp`) – This is the Frame Pointer. It points to the current function's frame.

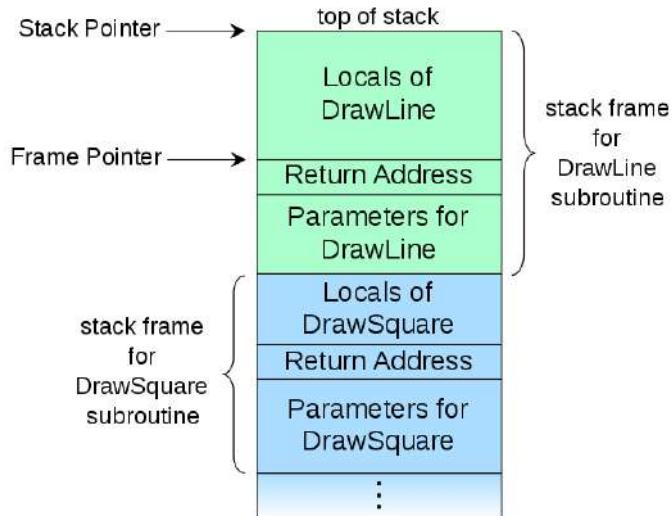
No stack overflow detection is supplied.

The C/C++ run-time start-up module initializes the stack pointer during the start-up and initialization sequence, see [Section 15.3.3 "Initialize Stack Pointer and Heap"](#).

On peut extraire de ce document les informations suivantes :

- Gestion software de la pile, contrairement au PIC18F.
- La taille par défaut du stack est de 1024 bytes, extensible par une commande.
- La stack grandit vers le bas (des adresses hautes vers les adresses basses).
- Registre 29 (SP) - Stack pointer (pointeur de pile). Il pointe sur la prochaine place libre sur le stack.
- Registre 30 (FP) - Frame pointer. Il pointe au début du frame courant. C'est une copie du stack pointer avant l'appel de la fonction courante.

La figure ci-dessous illustre les notions de stack pointer et frame pointer :



Source : Wikipédia

La fonction DrawLine est appelée par la fonction DrawSquare.

Les opérations sont, dans l'ordre :

1. La fonction DrawSquare place sur la pile les paramètres d'appel de DrawLine.
2. L'adresse de retour est placée sur la pile, puis la fonction DrawLine est appelée.
3. La fonction DrawLine place ses variables locales sur la pile.

Chaque fonction possède son stack frame (son "bloc" de pile qu'elle utilise). Les variables locales peuvent être référencées par rapport au frame pointer.

## 4.3. INITIALISATION DU STACK ET DU HEAP

Paragraphe 15.3.3

### 15.3.3 Initialize Stack Pointer and Heap

The Stack Pointer (`sp`) register must be initialized in the start-up code. To enable the start-up code to initialize the `sp` register, the linker must initialize a variable which points to the end of KSEG0/KSEG1 data memory<sup>1</sup>.

The linker allocates the stack to KSEG0 on devices featuring an L1 data cache. It allocates the stack to KSEG1 on devices that do not have an L1 cache.

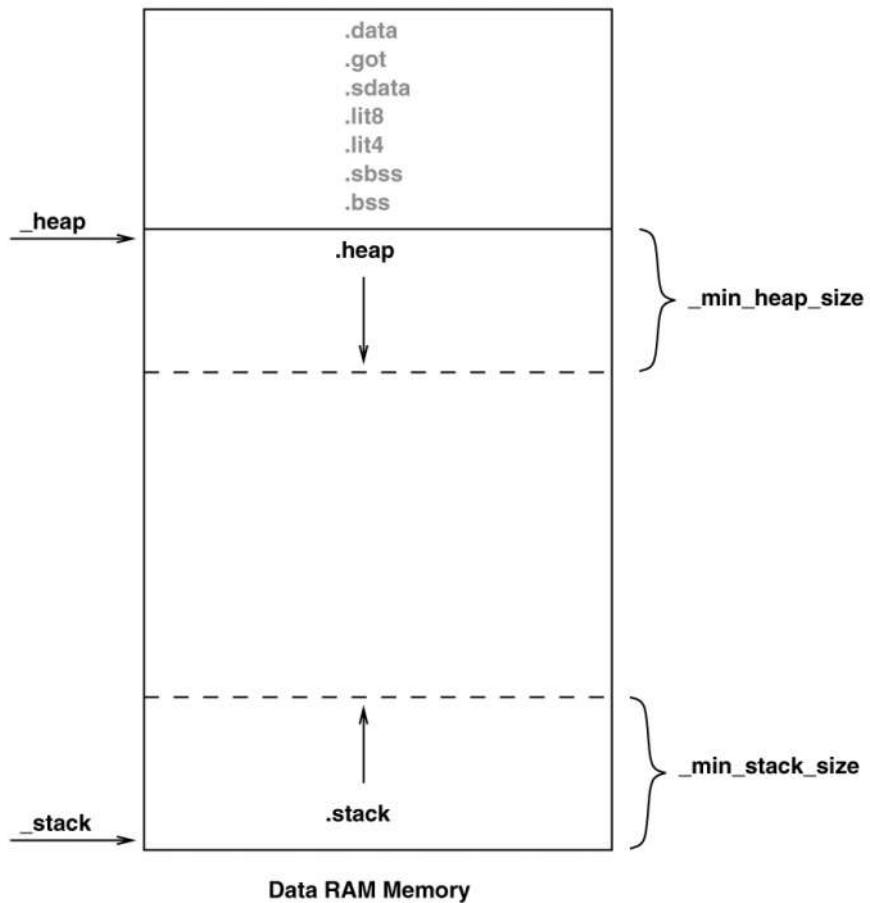
This variable is named `_stack`. The user can change the minimum amount of stack space allocated by providing the command line option `--defsym _min_stack_size=N` to the linker. `_min_stack_size` is provided by the linker script with a default value of 1024. On a similar note, the user may wish to utilize a heap with their application. While the start-up code does not need to initialize the heap, the standard C libraries (`sbrk`) must be made aware of the heap location and its size. The linker creates a variable to identify the beginning of the heap. The location of the heap is the end of the utilized KSEG0/KSEG1 data memory.

The linker allocates the heap to KSEG0 on devices that have an L1 cache. It allocates the heap to KSEG1 on devices that do not have an L1 cache.

This variable is named `_heap`. A user can change the minimum amount of heap space allocated by providing the command line option `--defsym _min_heap_size=M` to the linker. If the heap is used when the heap size is set to zero, the behavior is the same as when the heap usage exceeds the minimum heap size. Namely, it overflows into the space allocated for the stack.

The heap and the stack use the unallocated KSEG0/KSEG1 data memory, with the heap starting from a low address in KSEG0/KSEG1 data memory, and growing upwards towards the stack while the stack starts at a higher address in KSEG1 data memory and grows downwards towards the heap. The linker attempts to allocate the heap and stack together in the largest gap of memory available in the KSEG0/KSEG1 data memory region. If enough space is not available based on the minimum amount of heap size and stack size requested, the linker issues an error.

### 4.3.1. ORGANISATION DU STACK ET DU HEAP



### 4.3.2. LES REGISTRES DU PIC32

REGISTERS		
0	zero	Always equal to zero
1	at	Assembler temporary; used by the assembler
2-3	v0-v1	Return value from a function call
4-7	a0-a3	First four parameters for a function call
8-15	t0-t7	Temporary variables; need not be preserved
16-23	s0-s7	Function variables; must be preserved
24-25	t8-t9	Two more temporary variables
26-27	k0-k1	Kernel use registers; may change unexpectedly
28	gp	Global pointer
29	sp	Stack pointer
30	fp/s8	Stack frame pointer or subroutine variable
31	ra	Return address of the last subroutine call

C'est le registre 29 sp stack pointer qui est utilisé pour gérer la pile. Le registre 30 fp/s8 frame pointer permet de pointer sur des blocs (frame) dans la pile.

## 4.4. PUSH AND POP INSTRUCTIONS

Sources :

Les explications qui suivent sont tirées du site web :

<http://www.cs.umd.edu/class/spring2003/cmsc311/Notes/Mips/stack.html>

Understanding the Stack - Daté du 22.06.2003, consulté le 2.11.2017

Certains processeurs disposent d'instructions **push** et **pop** spécifiques. Dans le cas de l'architecture MIPS, il n'y en a pas. Il est possible de réaliser l'équivalent de ces instructions en manipulant directement le stack pointer.

Par convention, on utilise le registre \$r29 (sp) comme pointeur de pile.

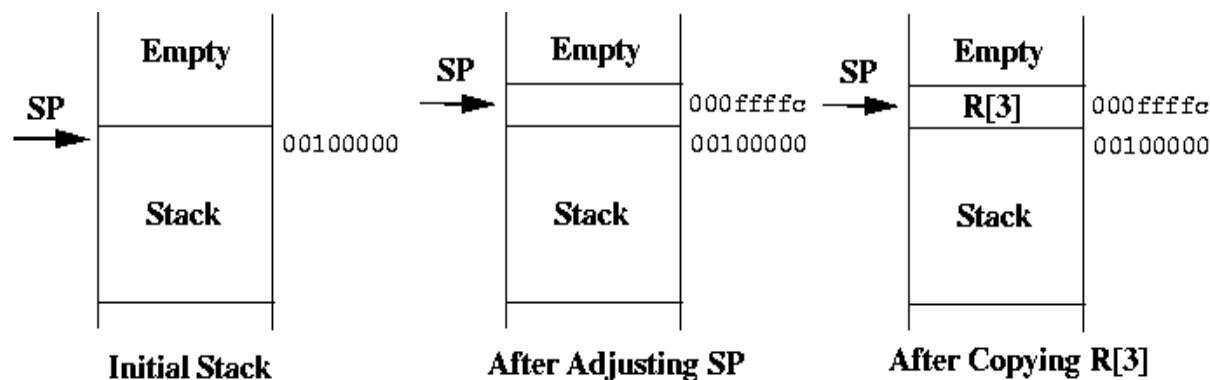
### 4.4.1. RÉALISATION DE L'ACTION PUSH

Voici comme exemple la réalisation du **push \$r3**, ce qui correspond à déposer le registre \$r3 sur la pile.

```
push: addi $sp, $sp, -4      # Decrement stack pointer by 4
      sw   $r3, 0($sp)       # Save $r3 to stack
```

Le label push n'est pas nécessaire; il sert de commentaire.

Voici un diagramme qui illustre l'action push :



Le diagramme de gauche montre la situation avant l'action.

Le diagramme du centre illustre l'effet du décrément de 4 du stack pointer (l'adresse diminue, ce qui fait graphiquement monter le pointeur). Une donnée de 32 bits (4 bytes) est mise sur la pile, d'où la valeur 4.

Le diagramme de droite montre la situation après la copie du contenu du registre \$r3. La valeur de \$r3 est placée à l'adresse **0x000f'fffc**.

Remarque : il est possible d'arriver au même résultat de la manière suivante :

```
push: sw $r3, -4($sp)    # Copy $r3 to stack
```

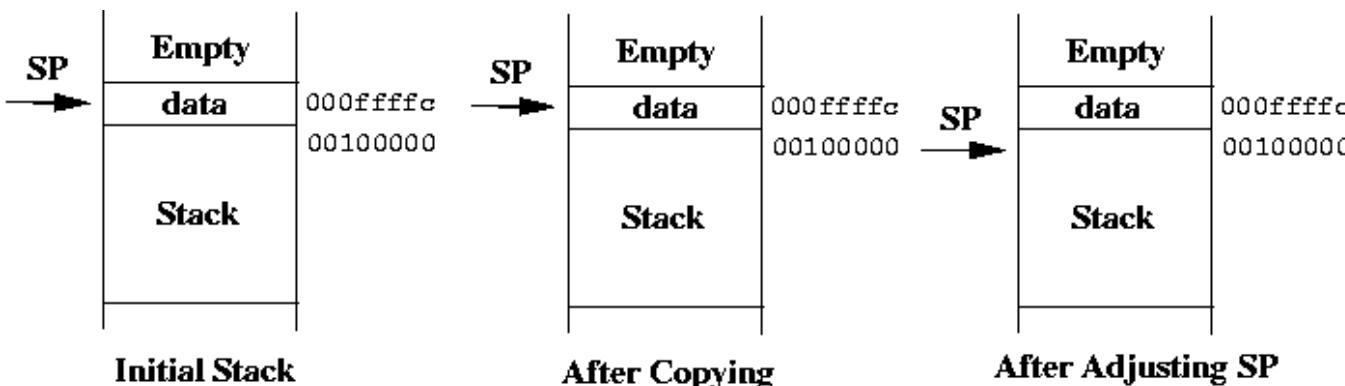
Cependant, dans la pratique on préfère la 1ère méthode pour avoir des offsets positifs par rapport au SP, surtout lorsque l'on push plusieurs éléments.

#### 4.4.2. RÉALISATION DE L'ACTION POP

La récupération de la valeur (popping off the stack) correspond à la manœuvre inverse. D'abord on copie le data du stack dans le registre, puis on ajuste la valeur du stack pointer.

```
pop: lw $r3, 0($sp)      # Copy from stack to $r3
        addi $sp, $sp, 4     # Increment sp by 4
```

Voici l'illustration de l'action Pop :



Le diagramme de gauche montre la situation initiale.

Le diagramme du centre montre la situation après la copie. La valeur "data" est copiée dans le registre que l'on ne voit pas sur le diagramme.

Le diagramme de droite montre la situation après ajustement du stack. Le data reste sur le stack mais n'est plus accessible. Il sera vraisemblablement écrasé lors d'une action Push.

#### 4.4.3. ACTION PUSH ET POP AVEC PLUSIEURS VALEURS

Lorsque l'on doit placer plusieurs valeurs sur le stack, on effectue alors un ajustement unique du stack qui correspond à la taille de l'ensemble. De même pour la récupération, on fera un ajustement unique.

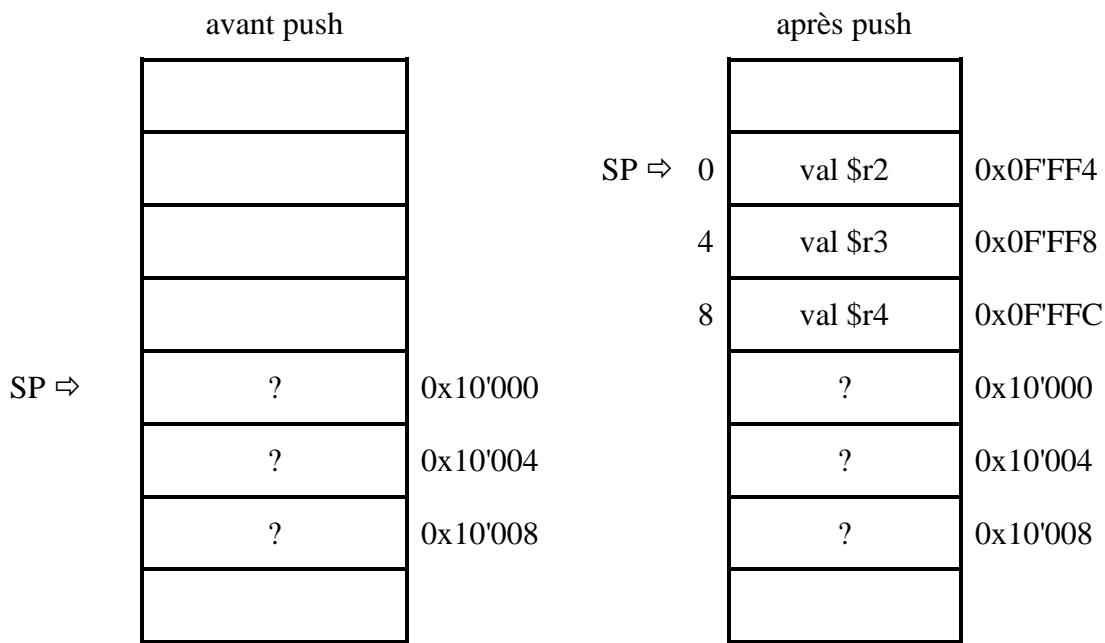
Par exemple, si l'on souhaite placer les registres \$r2, \$r3 et \$r4 sur la pile, cela donne le code suivant :

```
push: addi $sp, $sp, -12    # Decrement sp by 12
        sw   $r2, 0($sp)      # Save $r2 to stack
        sw   $r3, 4($sp)      # Save $r3 to stack
        sw   $r4, 8($sp)      # Save $r4 to stack
```

Comme chaque registre a une taille de 4 bytes, il est nécessaire de décrémenter la valeur du stack pointer de 12 ( $3 * 4$ ) pour disposer de l'espace nécessaire pour stocker les 3 registres.

L'ordre de stockage des registres est libre; le choix s'est naturellement porté sur l'ordre croissant des registres.

#### 4.4.3.1. ILLUSTRATION DU PUSH DE 3 REGISTRES

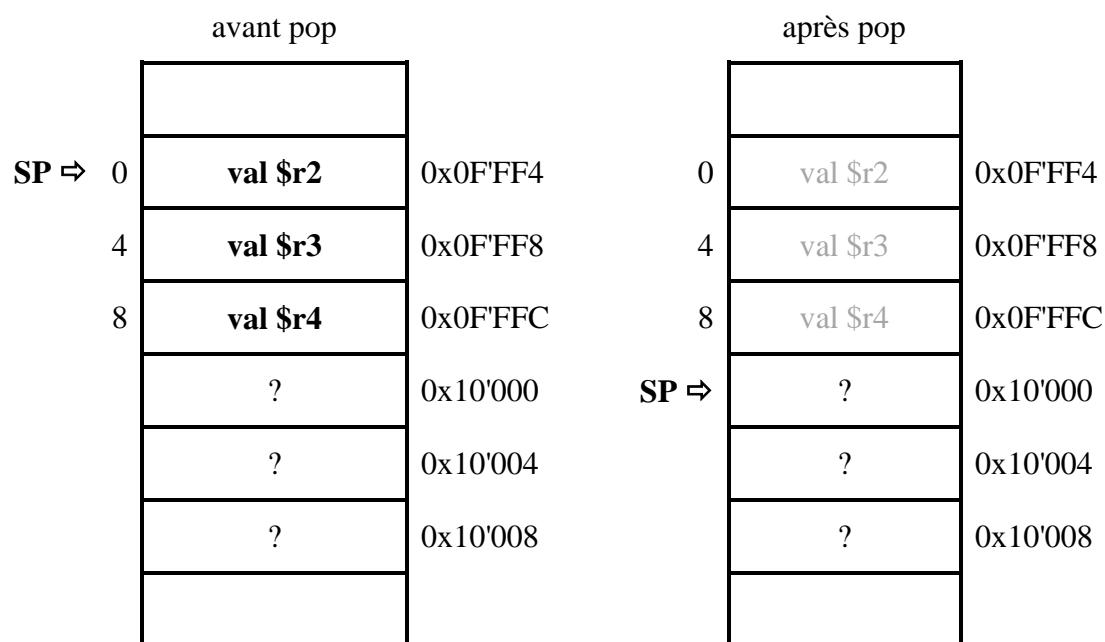


#### 4.4.3.2. ACTION POP DES 3 REGISTRES

L'action pop correspond à la manœuvre inverse : copies puis ajustement.

```
pop:    lw      $r2, 0($sp)    # Copy from stack to $r2
        lw      $r3, 4($sp)    # Copy from stack to $r3
        lw      $r4, 8($sp)    # Copy from stack to $r4
        addi   $sp, $sp, 12    # Increment sp by 12
```

#### 4.4.3.3. ILLUSTRATION DU POP DE 3 REGISTRES



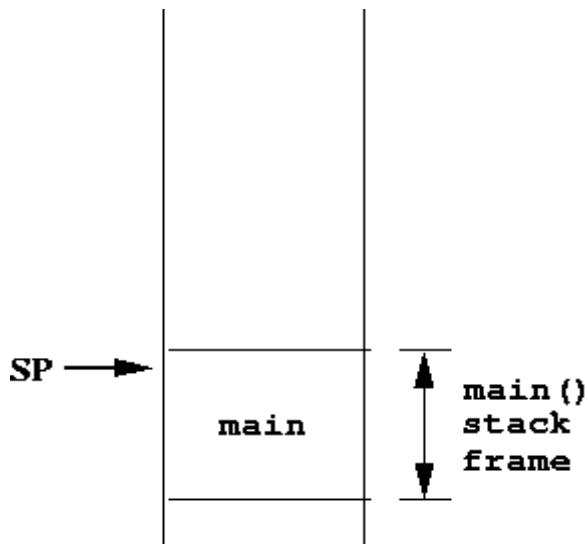
Remarque : après l'action pop, les valeurs ont été copiées, mais elles restent dans la pile.  
La position du SP est modifiée.

## 4.5. RELATION ENTRE PILE ET FONCTIONS

### 4.5.1. PILE ET FONCTIONS

Let's now see how the stack is used to implement functions. For each function call, there's a section of the stack reserved for the function. This is usually called a *stack frame*.

Let's imagine we're starting in **main()** in a C program. The stack looks something like this:

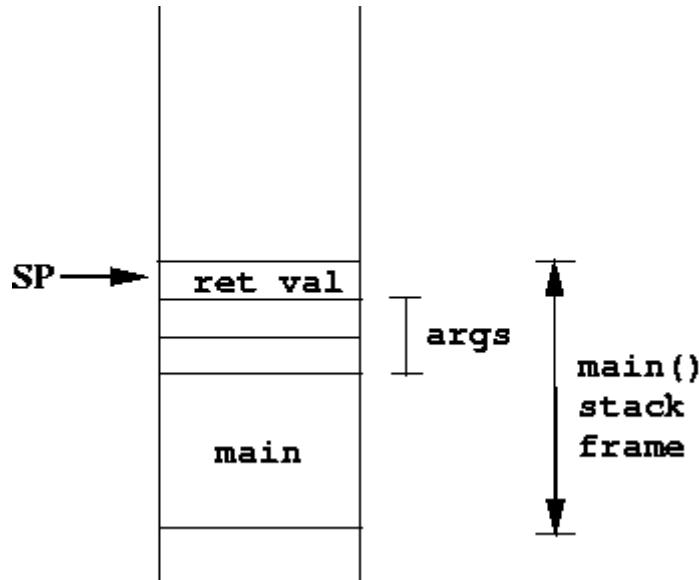


We'll call this the *stack frame* for **main()**. It is also called the *activation record*. A stack frame exists whenever a function has started, but yet to complete.

Suppose, inside of body of **main()** there's a call to **foo()**. Suppose **foo()** takes two arguments. One way to pass the arguments to **foo()** is through the stack. Thus, there needs to be assembly language code in **main()** to "push" arguments for **foo()** onto the stack.

#### 4.5.1.1. SITUATION STACK LORS DE LA PRÉPARATION DE L'APPEL DE FOO

Voici le contenu du stack frame du main() lors de la préparation de l'appel de la fonction foo().

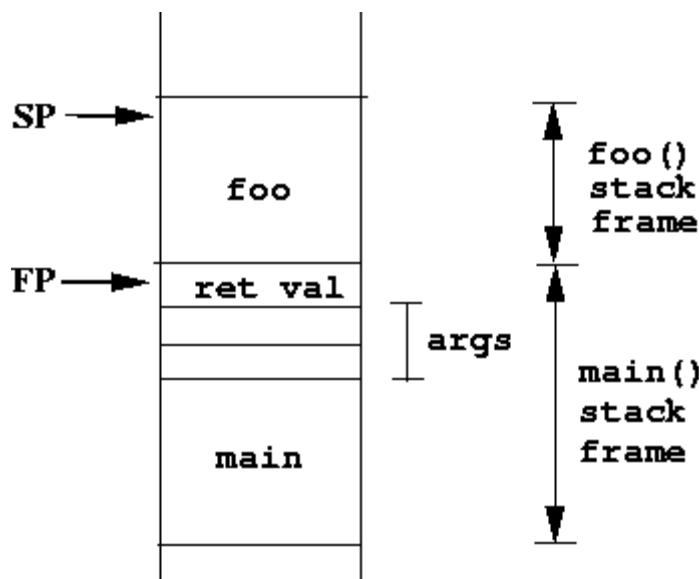


On constate une augmentation de la taille du stack frame du main. Les arguments sont placés dans ce stack frame et une place est prévue pour stocker la valeur de retour de la fonction.

☞ La valeur de retour (ret val) sera mise en place à la fin de l'exécution de la fonction.

#### 4.5.1.2. SITUATION LORS DE L'EXÉCUTION DE FOO

La fonction **foo()** peut avoir besoin de variables locales, ce qui conduit à prendre une zone supplémentaire sur le stack (en anglais : *foo() needs to push some space on the stack*).



Cela crée la nécessité d'un pointeur supplémentaire : le FP (frame pointer), pour mémoriser l'ancienne position du SP.

**foo()** can access the arguments passed to it from **main()** because the code in **main()** places the arguments just as **foo()** expects it.

We've added a new pointer called **FP** which stands for *frame pointer*. **The frame pointer points to the location where the stack pointer was**, just before **foo()** moved the stack pointer for **foo()**'s own local variables.

Having a frame pointer is convenient when a function is likely to move the stack pointer several times throughout the course of running the function. The idea is to keep the frame pointer fixed for the duration of **foo()**'s stack frame. The stack pointer, in the meanwhile, can change values.

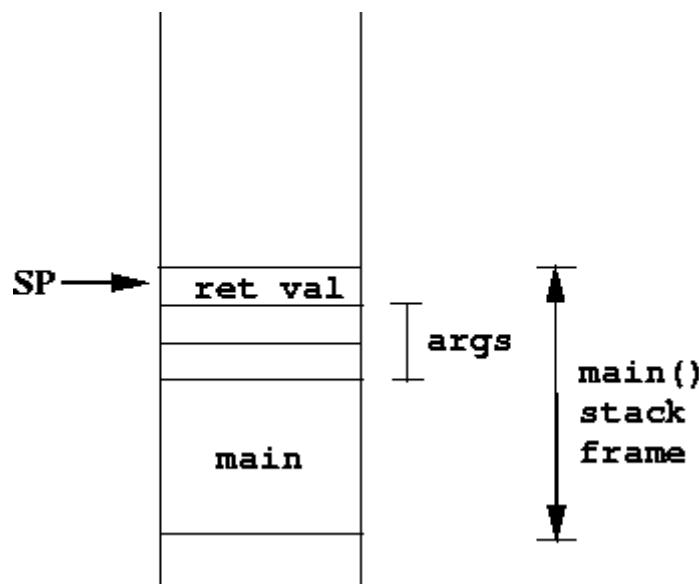
Thus, we can use the frame pointer to compute the locations in memory for both arguments as well as local variables. Since it doesn't move, the computations for those locations should be some fixed offset from the frame pointer.

And, once it's time to exit **foo()**, you just have to set the stack pointer to where the frame pointer is, which effectively pops off **foo()**'s stack frame. It's quite handy to have a frame pointer.

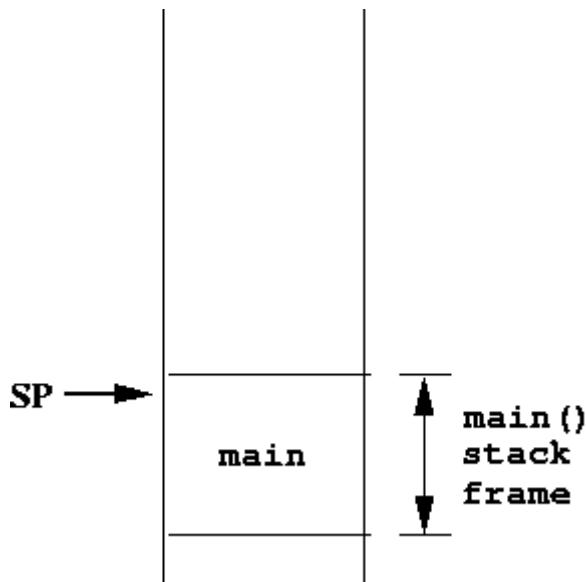
We can imagine the stack growing if **foo()** calls another function, say, **bar()**. **foo()** would push arguments on the stack just as **main()** pushed arguments on the stack for **foo()**.

#### 4.5.1.3. SITUATION APRÈS L'EXÉCUTION DE FOO

So when we exit **foo()** the stack looks just as it did before we pushed on **foo()**'s stack frame, except this time the return value has been filled in.



Once **main()** has the return value, it can pop that and the arguments to **foo()** off the stack.



#### 4.5.2. DÉBORDEMENT DE PILE (STACK OVERFLOW)

While stacks are generally large, they don't occupy all of memory. It is possible to run out of stack space.

For example, consider the code we had for **factorial**.

```
int fact( int n ) {
    if ( n == 0 )
        return 1 ;
    else
        return fact( n - 1 ) * n ;
}
```

Suppose **fact(-1)** is called. Then, the base case is never reached (well, it might be reached once we decrement so far that **n** wraps around to 0 again). This causes one stack frame after another to be pushed. Once the stack limit has been reached, we enter into invalid memory addresses, and the operating system takes over and kills your programming, telling you your program has a stack overflow.

Probably the most common cause of stack overflow is a recursive function that doesn't hit the base case soon enough. For fans of recursion, this can be a problem, so just keep that in mind.

Some languages (say, ML : Metalanguage) can convert certain kinds of recursive functions (called "tail-recursive" functions) into loops, so that only a constant amount of space is used.

### 4.5.3. IMPLÉMENTATION DANS LE MIPS32

In the previous discussion of function calls, we said that arguments are pushed on the stack and space for the return value is also pushed.

This is how CPUs used to do it. With the RISC revolution (admittedly, nearly 20 years old now) and large numbers of registers used in typical RISC machines, the goal is to (try and) avoid using the stack.

Why? The stack is in physical memory, which is RAM. Compared to accessing registers, accessing memory is much slower, probably on the order of 100 to 500 times as slow to access RAM than to access a register.

MIPS has many registers, so it does the following:

- There are four registers used to pass arguments: \$a0, \$a1, \$a2, \$a3.
- If a function has more than four arguments, or if any of the arguments is a large structure that's passed by value, then the stack is used.
- There must be a set procedure for passing arguments that's known to everyone based on the types of the functions. That way, the caller of the function knows how to pass the arguments, and the function being called knows how to access them. Clearly, if this protocol is not established and followed, the function being called would not get its arguments properly, and would likely compute bogus values or, worse, crash.
- The return value is placed in registers \$v0, and if necessary, in \$v1.

In general, this makes calling functions a little easier. In particular, the calling function usually does not need to place anything on the stack for the function being called.

However, this is clearly not a panacea. In particular, imagine **main()** calls **foo()**. Arguments are passed using **\$a0** and **\$a1**, say.

What happens when **foo()** calls **bar()**? If **foo()** has to pass arguments too, then by convention, it's supposed to pass them using **\$a0** and **\$a1**, etc. What if **foo()** needs the argument values from **main()** afterwards?

To prevent its own arguments from getting overwritten, **foo()** needs to save the arguments to the stack.

Thus, we don't entirely avoid using the stack.

### 4.5.4. LEAF PROCEDURES

In general, using registers to pass arguments and for return values doesn't prevent the use of the stack. Thus, it almost seems like we postpone the inevitable use of the stack. Why bother using registers for return values and arguments?

Eventually, we have to run code from a leaf procedure. This is a function that does not make any calls to any other functions. Clearly, if there were no leaf procedures, we wouldn't exit the program, at least, not in the usual way (If this isn't obvious to you, try to think about why there must be leaf procedures).

In a leaf procedure, there are no calls to other functions, so there's no need to save arguments on the stack. There's also no need to save return values on the stack. You just use the argument values from the registers, and place the return value in the return value registers.

## 4.6. MÉCANISME APPEL ET PASSAGE DES PARAMÈTRES

### 4.6.1. PRINCIPE CALL ET RETURN

Voici le principe de réalisation du CALL à une fonction et du RETURN.

#### 4.6.1.1. RÉALISATION DU CALL

Le "call" utilise l'instruction **JAL** (Jump And Link) :

```
jal address      # $ra = $pc + 4; goto address;
```

Le JAL effectue 2 actions:

- le stockage de l'adresse de retour (la prochaine instruction) dans \$ra :  
\$ra = \$pc + 4
- Le saut à l'adresse de la fonction

#### 4.6.1.2. RÉALISATION DU RETURN

Le "return" utilise l'instruction **JR** (Jump Register) :

```
jr $ra          # goto $ra = address retour;
```

### 4.6.2. APPELS IMBRIQUÉS

Voici un complément obtenu de :

<http://jc.hydrus.net/cs61c/handouts/proced1.pdf>

Le mécanisme du JAL et du JR ne convient que pour un appel à un seul niveau et sans paramètre. Pour supporter des appels imbriqués, il est impératif de mémoriser \$ra sur la pile.

Les 4 principes à respecter :

- Les valeurs des registres \$s0-\$s7 doivent être préservées : Ces registres doivent avoir la même valeur en quittant la procédure qu'à l'entrée dans la procédure.
- Idem pour registre \$ra : L'adresse contenue doit être disponible à la fin de la procédure et être valable.
- Idem pour les valeurs (adresses) contenues dans \$sp et \$fp : ces dernières doivent également être préservées.
- Les registres, \$t0-\$t9, \$a0-\$a3, et \$v0-v1 peuvent être modifiés par la fonction.

#### **4.6.2.1. PRINCIPE DE L'APPEL IMBRIQUÉ**

Voici une situation d'une fonction nommée *procedure* qui en appelle une 2ème (*procedure2*).

L'appel est réalisé par *jal procedure*.

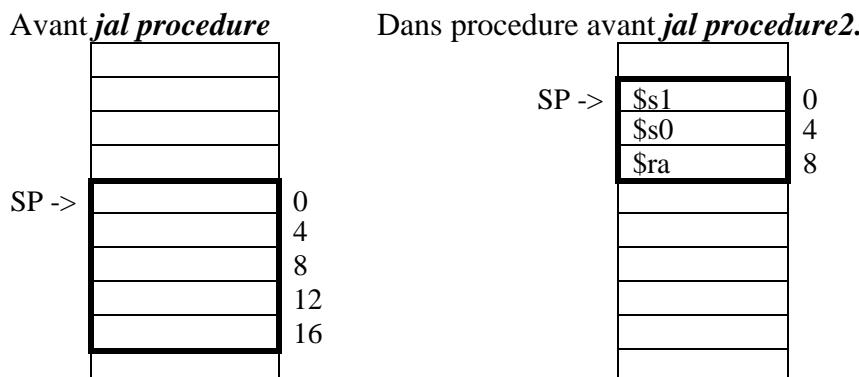
```

procedure:
Prologue { addi $sp, $sp, -12
           sw $ra, 8($sp)
           sw $s0, 4($sp)
           sw $s1, 0($sp)
           addi $s0, $zero, 50
           addi $s0, $zero, -23
Body   { add $a0, $s0, $s1
           jal procedure2
           lw $s1, 0($sp)
           lw $s0, 4($sp)
           lw $ra, 8($sp)
           addi $sp, $sp, 12
Epilogue { jr $ra

```

Cette fonction ne reçoit pas de paramètre et ne retourne pas de valeur.

- Dans le **prologue**, il y a création de place sur le stack, puis sauvegarde de 3 registres.
  - Dans le corps de la fonction (*body*), il y a modification des valeurs des 3 registres et l'appel de procédure2.
  - Dans l'**épilogue**, il y a récupération des 3 registres et réajustement du stack en prévision du retour.



Remarque : Cet exemple ne permet pas de connaître le contenu du stack frame du contexte appelant.

### 4.6.3. APPEL ET PASSAGE DE PARAMÈTRE

Bien souvent, lors de l'appel d'une fonction, il est nécessaire de lui fournir des paramètres et aussi de récupérer le résultat.

Au niveau du MIPS32, 4 registres sont prévus pour cela : **\$a0** à **\$a3**. Si davantage d'arguments sont nécessaires, ils seront placés dans la pile. Dans ce cas, ils doivent être placés dans la pile avant les éléments habituellement sauvés, car l'utilisateur de la fonction est celui qui alloue de la mémoire et la libère après l'appel de la fonction.

Voici la situation de la fonction Ftest qui elle-même appelle la fonction FDivAB.

```
// Fonction FDivAB
int FDivAB (int A, int B)
{
    return (A / B);
}

// Ftest
int Ftest (int ValA, int ValB)
{
    int Ratio;
    Ratio = FDivAB(ValA, ValB);
    return Ratio;
}
```

Ainsi que le programme principal :

```
void main() {
    int Res1, Res2;
    int V1 = 10;
    int V2 = 20;
    int V3 = 30;
    int V4 = 40;

    // Appels de la fonction Ftest
    Res1 = Ftest(V1, V2);
    Res2 = Ftest(V3, V4);
}
```

#### 4.6.3.1. FONCTION MAIN EN ASSEMBLEUR

Voici le programme principal en assembleur :

```

24:           void main() {
9D000088  27BDFFD0 ADDIU SP, SP, -48
9D00008C  AFBF002C SW RA, 44(SP)
9D000090  AFBE0028 SW S8, 40(SP)
9D000094  03A0F021 ADDU S8, SP, ZERO } prologue
25:
26:
9D000098  2402000A ADDIU V0, ZERO, 10
9D00009C  AFC20010 SW V0, 16(S8) // v1 en 16(S8)
27:
9D0000A0  24020014 ADDIU V0, ZERO, 20
9D0000A4  AFC20014 SW V0, 20(S8) // v2 en 20(S8)
28:
9D0000A8  2402001E ADDIU V0, ZERO, 30
9D0000AC  AFC20018 SW V0, 24(S8) // v3 en 24(S8)
29:
9D0000B0  24020028 ADDIU V0, ZERO, 40
9D0000B4  AFC2001C SW V0, 28(S8) // v4 en 28(S8)
30:
31:           // Appel de la fonction Ftest
32:           Res1 = Ftest(V1, V2);
9D0000B8  8FC40010 LW A0, 16(S8)
9D0000BC  8FC50014 LW A1, 20(S8)
9D0000C0  0F400010 JAL Ftest
9D0000C4  00000000 NOP
9D0000C8  AFC20020 SW V0, 32(S8) // Res1 en 32(S8)
33:
9D0000CC  8FC40018 Res2 = Ftest(V3, V4);
9D0000D0  8FC5001C LW A0, 24(S8)
9D0000D4  0F400010 LW A1, 28(S8)
9D0000D8  00000000 JAL Ftest
9D0000DC  AFC20024 NOP
9D0000E0  03C0E821 SW V0, 36(S8) // Res2 en 36(S8)
34:
9D0000E4  8FBF002C }
9D0000E8  8FBE0028 ADDU SP, S8, ZERO } épilogue
9D0000EC  27BD0030 LW RA, 44(SP)
9D0000F0  03E00008 LW S8, 40(SP)
9D0000F4  00000000 ADDIU SP, SP, 48
                           JR RA
                           NOP

```

On constate :

- Même le **main** contient un prologue et un épilogue
- Le prologue se charge de préparer la place sur la pile et de sauvegarder RA (adresse de retour) et S8/FP (frame pointer).
- Ensuite, tous les stockages de variables locales sont faits à l'aide du frame pointer S8/FP.
- L'épilogue rétablit la situation du début de fonction (l'inverse des opérations du prologue), puis retourne à l'appelant (JR RA).

#### **4.6.3.2. FONCTION FTEST EN ASSEMBLEUR**

Voici l'équivalent en assembleur de la fonction **Ftest** :

```

int Ftest (int ValA, int ValB)
{
    prologue {
        ADDIU SP, SP, -32          // Crée place sur stack
        SW RA, 28(SP)             // push $ra (return addr)
        SW S8, 24(SP)              // push $s8 (frame pointer)
        ADDU S8, SP, ZERO          // $s8 = $sp (nouveau fp)
        SW A0, 32(S8)              // push $a0 (sauvegarde ValA)
        SW A1, 36(S8)              // push $a1 (sauvegarde ValB)

        int Ratio;
        Ratio = FDivAB(ValA, ValB);
        LW A0, 32(S8)              // pop $a0 (passage param. ValA)
        LW A1, 36(S8)              // pop $a1 (passage param. ValB)
        JAL FDivAB
        NOP
        SW V0, 16(S8)              // push $v0

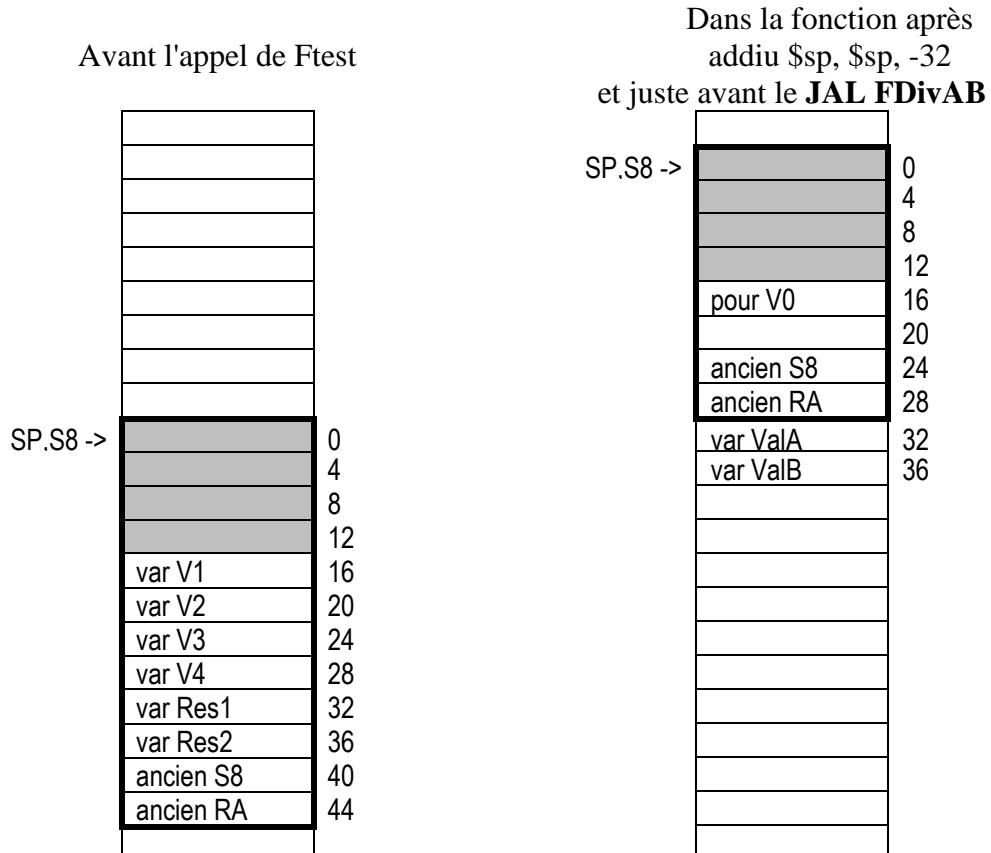
        return Ratio;
        LW V0, 16(S8)              // pop $v0
    }

    épilogue {
        ADDU SP, S8, ZERO          // $sp = $s8
        LW RA, 28(SP)              // pop $ra
        LW S8, 24(SP)              // pop $s8
        ADDIU SP, SP, 32            // maj du SP
        JR RA                      // retour
    }
}

```

#### **4.6.3.3. EVOLUTION PILE AVEC FONCTION FTST**

Pour bien comprendre la manœuvre, voici la situation du stack :



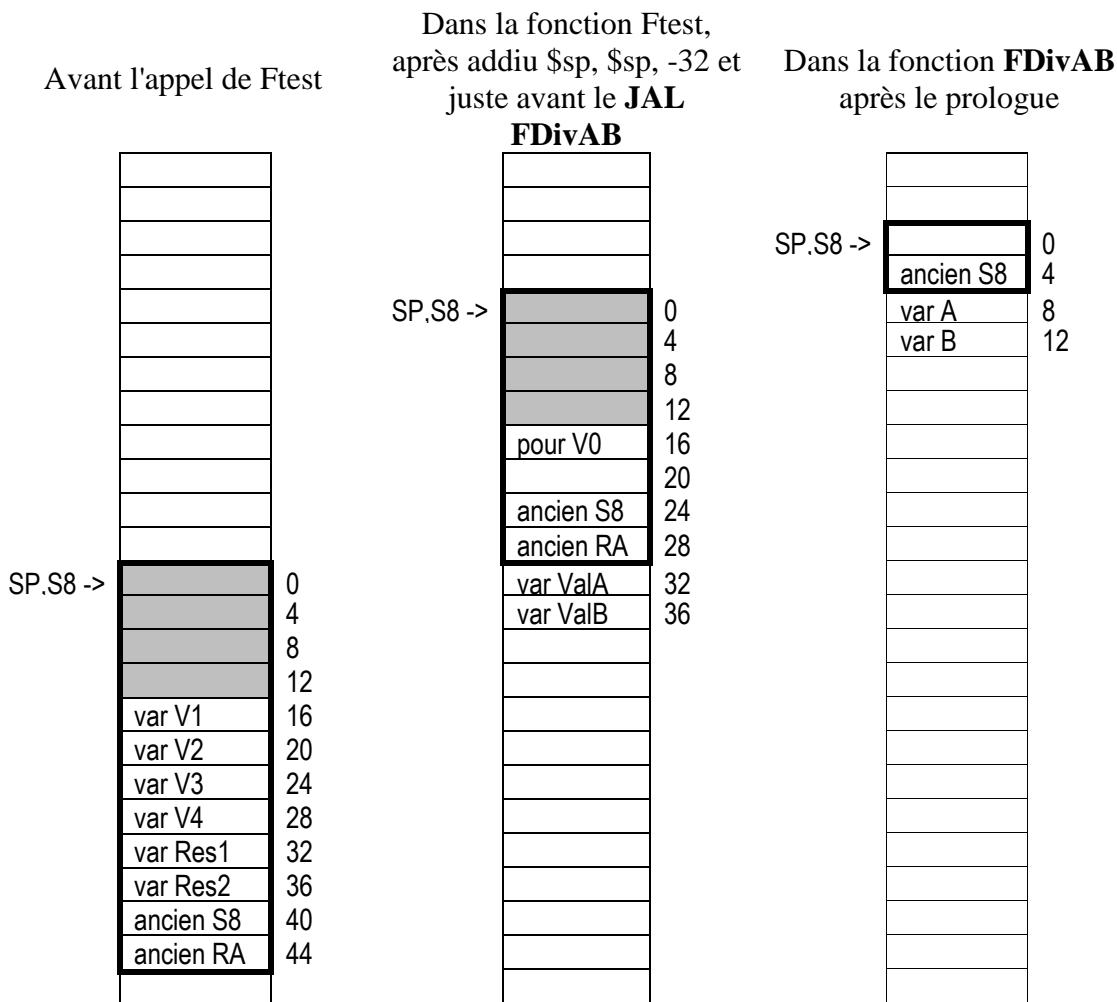
Le listing en assembleur du main permet de comprendre le contenu de la pile.

On remarque que dans le stack frame du main, les 4 premiers mots sont laissés libres pour le futur stockage de \$a0, \$a1, \$a2 et \$a3.

On remarque aussi que les variables du main sont placées dans le stack frame. C'est d'ailleurs également le cas pour toute autre fonction.

#### 4.6.3.4. EVOLUTION DE LA PILE AVEC APPEL IMBRIQUÉ

Pour mieux comprendre ce qui se passe, il est intéressant d'obtenir la situation de la pile lorsque que l'on effectue l'appel à la fonction intérieure.



Appel de la fonction FDivAB dans Ftest :

```

Ratio = FDivAB(ValA, ValB);
LW A0, 32($8)           // pop $a0 (ValA)
LW A1, 36($8)           // pop $a1 (ValB)
JAL FDivAB
    
```

On remarque que le stack frame du contexte appelant laisse de la place pour les variables. La fonction FDivAB ne sauve pas RA car elle n'appelle pas d'autre fonction (leaf procedure).

#### 4.6.3.5. FONCTION FDivAB EN ASSEMBLEUR

Voici l'équivalent en assembleur de la fonction FDivAB :

```
// Fonction FDivAB
int FDivAB (int A, int B)
{
    prologue {
        ADDIU SP, SP, -8      // Crée place sur stack
        SW S8, 4(SP)          // push $s8 (frame pointer)
        ADDU S8, SP, ZERO     // $s8 = $sp (nouveau fp)
        SW A0, 8(S8)           // push $a0 (passage param. A)
        SW A1, 12(S8)          // push $a1 (passage param. B)
        return (A / B);
        LW V1, 8(S8)           // Effectue A / B
        LW V0, 12(S8)
        DIV V1, V0
        TEQ V0, ZERO
        MFHI V0
        MFLO V0               // Résultat dans V0
    }
    épilogue {
        ADDU SP, S8, ZERO     // $sp = $s8
        LW S8, 4(SP)          // pop $s8
        ADDIU SP, SP, 8         // maj du SP
        JR RA                  // Retour
    }
}
```

On observe l'utilisation directe de RA sans sauvegarde.

## 4.7. CONCLUSION

Ce chapitre a tenté de montrer le mécanisme de la pile et de son utilisation avec des frame pour assurer le passage de paramètres aux fonctions et la gestion des retours.

## 4.8. HISTORIQUE DES VERSIONS

### 4.8.1. VERSION 1.0 FÉVRIER 2014

Création du document (en grande partie en anglais) et découverte du mécanisme.

### 4.8.2. VERSION 1.5 NOVEMBRE 2014

Saut à la version 1.5 pour cohérence avec la nouvelle version du cours liée à Harmony.  
Amélioration des explications en anglais par des titres et remarques en français.

### 4.8.3. VERSION 1.7 NOVEMBRE 2015

Saut à la version 1.7 pour cohérence avec la nouvelle version du cours liée à Harmony.  
L'exemple n'a pas été adapté à Harmony. Maintien des explications en anglais.

### 4.8.4. VERSION 1.7BIS NOVEMBRE 2015

Reprise des exemples d'évolution de la pile sur la base d'un exemple plus simple.  
Suppression de l'exemple avec DispHex.

### 4.8.5. VERSION 1.8 NOVEMBRE 2016

Adaptation des références à la documentation. Traduction en français de la section push et pop.  
La section gestion pile et trame reste en anglais. Complément de l'exemple pratique.

### 4.8.6. VERSION 1.9 NOVEMBRE 2017

Reprise et relecture par SCA.

**MINF**  
**Programmation des PIC32MX**

# **Chapitre 5**

## **Les interruptions**



## **Théorie PIC32MX**

**Christian HUBER (CHR)**  
**Serge CASTOLDI (SCA)**  
**Version 1.9 novembre 2017**



## CONTENU

<b>5. Les interruptions du PIC32MX</b>	<b>5-1</b>
<b>5.1. Notion d'interruption</b>	<b>5-1</b>
5.1.1. Aspect temporel du traitement d'une interruption	5-2
5.1.1. Gestion du signal d'interruption	5-3
5.1.2. Fréquence limite d'une interruption	5-4
<b>5.2. Mécanisme des interruptions du PIC32MX</b>	<b>5-5</b>
5.2.1. Quelques caractéristiques	5-5
5.2.2. Le contrôleur d'interruption	5-5
5.2.3. Les registres SFR pour la gestion des interruptions	5-6
5.2.3.1. Le registre INTCON	5-7
5.2.3.2. Le registre INTSTAT	5-8
5.2.3.3. Les registres IFSx	5-8
5.2.3.4. Les registres IECx	5-8
5.2.3.5. Les registres IPCx	5-8
5.2.4. Relation entre sources d'interruption et registres	5-9
5.2.5. Mécanisme de traitement des interruptions	5-11
5.2.5.1. A propos des shadow registers	5-13
5.2.6. Calcul de l'adresse du vecteur d'interruption	5-13
5.2.6.1. Valeurs utilisées par le compilateur	5-13
5.2.6.2. Adresse des vecteurs	5-13
5.2.6.3. Calcul en mode simple vecteur	5-13
5.2.6.4. Calcul en mode multiple vecteur	5-14
5.2.7. Contenu des vecteurs d'interruption	5-14
5.2.8. Contenu système de la réponse à l'interruption	5-14
5.2.9. Priorités des Interruptions	5-14
5.2.9.1. Cas des sous-priorités	5-14
<b>5.3. Gestion des interruptions avec Harmony &amp; XC32</b>	<b>5-15</b>
5.3.1. Fichier à inclure	5-15
5.3.2. Sélection du mode	5-15
5.3.2.1. PLIB_INT_MultiVectorSelect	5-15
5.3.2.2. PLIB_INT_SingleVectorSelect	5-15
5.3.3. Configuration de l'interruption	5-16
5.3.3.1. PLIB_INT_SourceEnable	5-16
5.3.3.2. PLIB_INT_VectorPrioritySet	5-17
5.3.3.3. PLIB_INT_VectorSubPrioritySet	5-18
5.3.3.4. Exemple de configuration	5-18
5.3.4. Relation routine de réponse et vecteur	5-19
5.3.4.1. Indication du niveau de priorité	5-19
5.3.4.2. Situation avec Harmony	5-19
5.3.4.3. Définition des numéros de vecteurs	5-20
5.3.4.4. Détails macro _ISR	5-20
5.3.4.5. Détails macro _ISR_AT_VECTOR	5-21
5.3.4.6. Vector attribute	5-21
<b>5.4. Exemple pratique avec timer</b>	<b>5-22</b>
5.4.1. Principe de l'exemple	5-22
5.4.2. Configuration timer 1 et interruption	5-22

5.4.2.1. Autorisation de l'interruption lors du Start	5-23
5.4.3. Routine réponse interruption du timer 1	5-23
5.4.3.1. Mise à zéro du flag d'interruption	5-23
5.4.4. Listing assembleur	5-24
5.4.4.1. Contenu Vecteur timer 1	5-24
5.4.4.2. Réponse interruption timer1	5-24
5.4.5. Configuration timer 4 et interruption	5-26
5.4.5.1. Autorisation de l'interruption lors du Start	5-27
5.4.6. Routine réponse interruption du timer 4	5-27
5.4.6.1. Réponse interruption timer4 en assembleur	5-28
5.4.7. Action pour faire fonctionner le Test	5-29
<b>5.5. Les interruptions externes</b>	<b>5-30</b>
5.5.1. Liste des interruptions externes	5-30
5.5.1.1. Définitions des sources	5-30
5.5.1.2. Définitions des Vecteurs (plib_int)	5-30
5.5.1.3. Définitions des Vecteurs (macro __ISR)	5-30
5.5.2. Configuration de la polarité du flanc	5-31
5.5.2.1. Type énuméré INT_EXTERNAL_SOURCES	5-31
5.5.2.2. PLIB_INT_ExternalRisingEdgeSelect, exemple	5-31
<b>5.6. Temps de réaction d'une interruption externe</b>	<b>5-32</b>
5.6.1. Principe de l'exemple	5-32
5.6.1.1. Contrainte Kit pour la réalisation	5-32
5.6.2. Configuration des Int. Ext. avec le MHC	5-32
5.6.2.1. Configuration Int3 (Instance 0)	5-33
5.6.2.2. Configuration Int4 (Instance 1)	5-33
5.6.2.3. Modification priorité interruption timer 4	5-33
5.6.3. Code généré pour les interruptions ext. par le MHC	5-33
5.6.3.1. Configuration Int3	5-34
5.6.3.2. Configuration Int4	5-34
5.6.4. Les fonctions SYS_INT	5-34
5.6.4.1. La macro SYS_INT_VectorPrioritySet	5-34
5.6.4.2. La macro SYS_INT_VectorSubprioritySet	5-34
5.6.4.3. La macro SYS_INT_SourceEnable	5-34
5.6.4.4. La macro SYS_INT_ExternalInterruptTriggerSet	5-35
5.6.5. Réponses aux interruptions externes	5-35
5.6.5.1. Réponse interruption Int3	5-35
5.6.5.2. Réponse interruption Int4	5-35
5.6.6. Action pour faire fonctionner le Test	5-36
5.6.6.1. Modification pour le Timer2	5-36
5.6.7. Mesure des temps de réaction	5-37
5.6.7.1. Vue d'ensemble	5-37
5.6.7.2. Temps réaction Int3	5-37
5.6.7.3. Temps réaction Int4	5-38
5.6.8. Différence dans les routines de réponse	5-39
5.6.8.1. ISR Int3 en assembleur	5-39
5.6.8.2. ISR Int4 en assembleur	5-40
5.6.9. Conclusion sur les interruptions externes	5-42
5.6.9.1. Fréquence limite	5-42
5.6.9.2. Situation favorable	5-42
<b>5.7. Conclusion</b>	<b>5-42</b>

<b>5.8. Historique des versions</b>	<b>5-43</b>
5.8.1. Version 1.0 février 2014	5-43
5.8.2. Version 1.5 décembre 2014	5-43
5.8.3. Version 1.7 décembre 2015	5-43
5.8.4. Version 1.8 novembre 2016	5-43
5.8.5. Version 1.9 novembre 2017	5-43



## 5. LES INTERRUPTIONS DU PIC32MX

Ce chapitre traite du mécanisme et de la gestion des interruptions du PIC32MX.

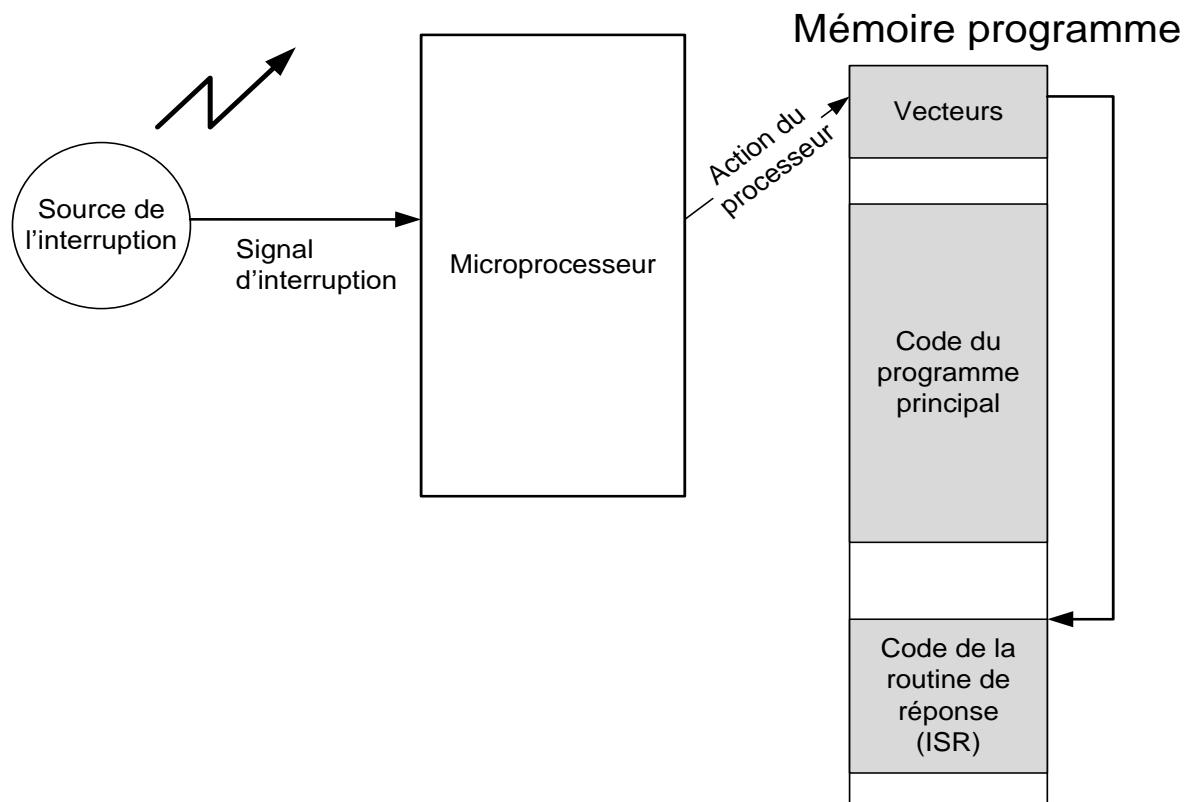
Les documents de référence sont :

- Document MPLAB-XC32-Users-Guide.pdf (documentation du compilateur xc32) : Chapter 14 : Interrupts  
Ce chapitre se base sur le compilateur xc32 v1.42.
- La documentation "PIC32 Family Reference Manual" : Section 8 : Interrupts
- Pour les détails spécifiques au PIC32MX795F512L, il faut se référer au document "PIC32MX5XX/6XX/7XX Family Data Sheet" : Section 7 : Interrupt controller

### 5.1. NOTION D'INTERRUPTION

Avant de découvrir les détails du mécanisme, voici quelques notions de base concernant les interruptions.

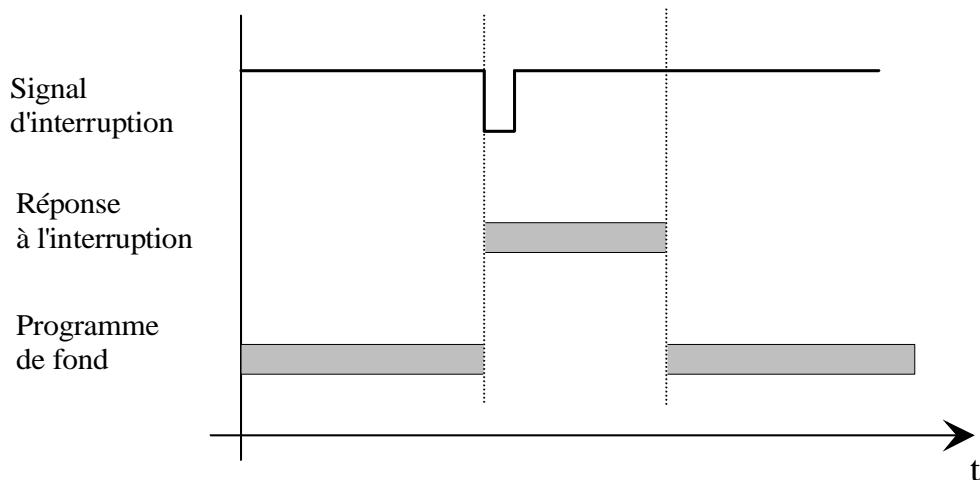
Voici les différents éléments à considérer :



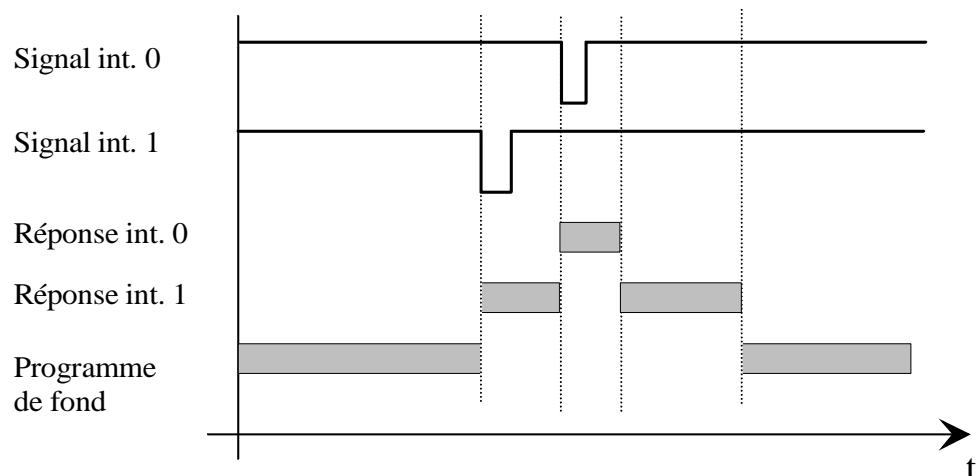
Le mécanisme des interruptions hardware permet d'établir un lien entre un phénomène extérieur au CPU et un sous-programme, dans le but d'exécuter ce sous-programme chaque fois que le phénomène extérieur survient, et ceci en interrompant momentanément le déroulement du programme en exécution à ce moment-là.

Une interruption est signalée au processeur par le changement d'état d'une ligne connectée sur les entrées d'interruption du processeur ou par le changement d'un élément interne au microcontrôleur (cas des timers par exemple).

### 5.1.1. ASPECT TEMPOREL DU TRAITEMENT D'UNE INTERRUPTION



La plus part des processeurs ont plusieurs entrées d'interruption. Ces entrées n'ont pas la même priorité, ce qui permet d'interrompre l'exécution du sous-programme lié à une interruption (que l'on appelle "routine de réponse à une interruption", en anglais **ISR** pour "*Interrupt Service Routine*") par une interruption plus prioritaire.

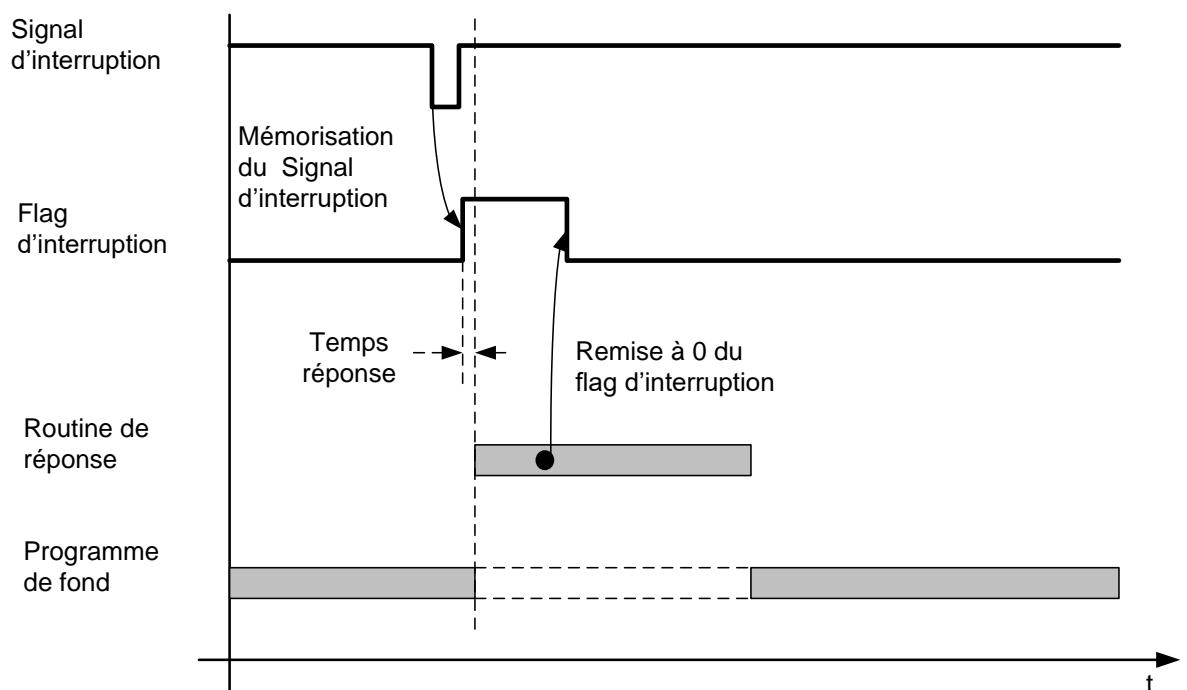


### 5.1.1. GESTION DU SIGNAL D'INTERRUPTION

En principe, le signal d'interruption est une impulsion. Sa durée doit être suffisante pour que le processeur puisse la détecter et la mémoriser (image de l'interruption dans un bit d'un registre = flag d'interruption).

La remise à 0 du flag d'interruption doit en général être faite par logiciel dans la routine d'interruption.

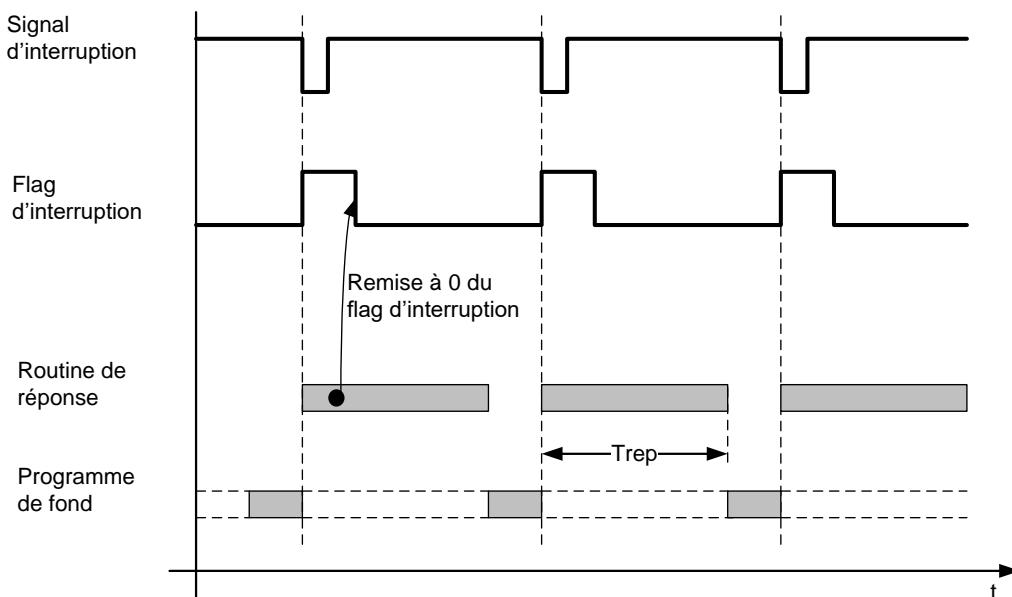
Toutefois, des cas existent où la remise à zéro est automatique (faite par hardware) au moment de l'entrée dans la routine d'interruption. Par exemple, lorsque la source d'interruption est une réception USART (port de communication série), l'action de lire le circuit de communication pourrait indirectement remettre à 0 le flag d'interruption.



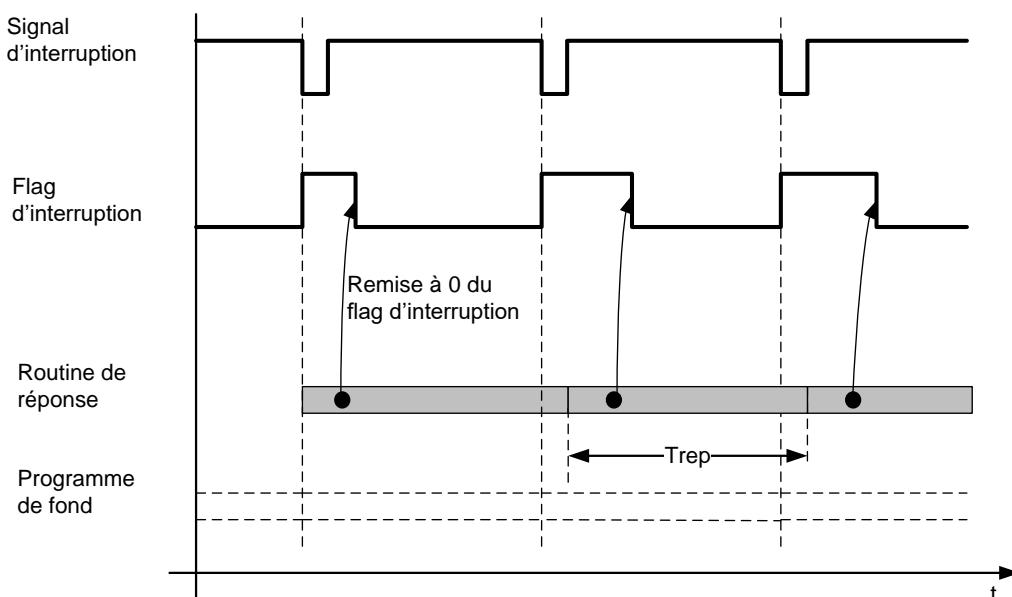
### 5.1.2. FRÉQUENCE LIMITÉE D'UNE INTERRUPTION

Lorsque par exemple on utilise un timer pour générer cycliquement une interruption, il faut veiller à ce que le temps d'exécution de la routine de réponse à l'interruption reste inférieur à la période du signal d'interruption. Dans la pratique si l'on souhaite conserver une certaine activité du programme de fond, il faut conserver de l'ordre de 20 % du temps pour ce dernier, donc le temps de traitement de la réponse ne doit pas dépasser 80% de la période du signal d'interruption.

Dans tous les cas, une bonne pratique est de ne traiter **que le strictement nécessaire dans la routine d'interruption** et d'effectuer le reste de la tâche dans l'application (tâche de fond, hors interruption). **Tout blocage, boucle potentiellement bloquante, traitement long ou calcul mathématique complexe est à éviter au maximum dans une routine d'interruption.**



Si le temps de traitement est supérieur à la période du signal d'interruption, il y a perte du synchronisme et consommation complète du temps CPU, le programme de fond ne sera plus exécuté.



## 5.2. MÉCANISME DES INTERRUPTIONS DU PIC32MX

Dans ce paragraphe, nous allons étudier le mécanisme des interruptions du PIC32MX, en commençant par les éléments d'architecture et les possibilités offertes.

Le PIC32MX gère les requêtes d'interruption en provenance des différents périphériques ainsi que des interruptions externes.

### 5.2.1. QUELQUES CARACTÉRISTIQUES

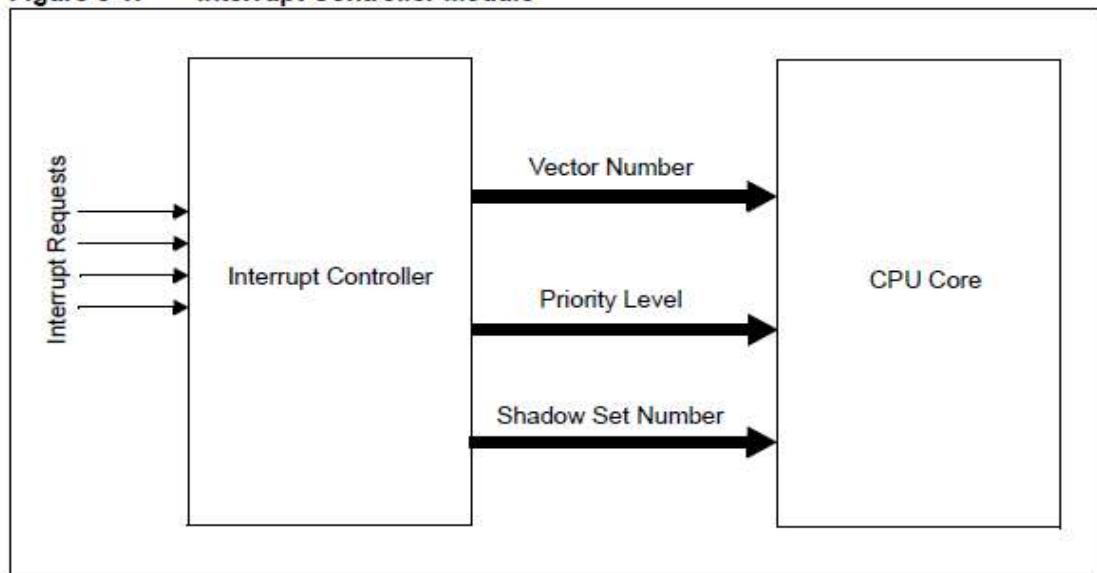
Voici quelques caractéristiques principales:

- Jusqu'à 96 sources d'interruptions.
- Jusqu'à 64 vecteurs d'interruption.
- Mode de gestion simple vecteur ou multi-vecteur. Le mode simple vecteur est un héritage qui ne sera à priori utilisé que rarement.
- Cinq interruptions externes avec configuration de la polarité du flanc.
- Sept niveaux de priorité pour chaque vecteur.
- Quatre niveaux de sous-priorité pour chaque priorité.
- L'adresse de la table des vecteurs d'interruption est configurable par l'utilisateur.
- L'espacement entre les vecteurs d'interruption est configurable par l'utilisateur.

### 5.2.2. LE CONTRÔLEUR D'INTERRUPTION

Le PIC32MX possède un contrôleur d'interruption séparé du cœur du processeur.

**Figure 8-1: Interrupt Controller Module**



### 5.2.3. LES REGISTRES SFR POUR LA GESTION DES INTERRUPTIONS

Le mécanisme de gestion des interruptions utilise certains registres SFR. Il y a 3 registres 32 bits IFS et IEC pour couvrir les 96 sources d'interruption. Il y a 16 registres IPC pour établir les niveaux de priorités.

- **INTCON:** Interrupt Control Register
- **INTSTAT:** Interrupt Status Register
- **TPTMR:** Temporal Proximity Timer Register
- **IFSx:** Interrupt Flag Status Registers
- **IECx:** Interrupt Enable Control Registers
- **IPCx:** Interrupt Priority Control Registers

La table 8-1 résume la situation :

**Table 8-1: Interrupts Register Summary**

Address Offset	Name	Bit Range	Bit 31/23/15/7	Bit 30/22/14/6	Bit 29/21/13/5	Bit 28/20/12/4	Bit 27/19/11/3	Bit 26/18/10/2	Bit 25/17/9/1	Bit 24/16/8/0									
0x00	INTCON <sup>(1,2,3)</sup>	31:24	—	—	—	—	—	—	—	—									
		23:16	—	—	—	—	—	—	—	SS0									
		15:8	—	FRZ	—	MVEC	—	TPC<2:0>											
		7:0	—	—	—	INT4EP	INT3EP	INT2EP	INT1EP	INT0EP									
0x10	INTSTAT <sup>(1,2,3)</sup>	31:24	—	—	—	—	—	—	—	—									
		23:16	—	—	—	—	—	—	—	—									
		15:8	—	—	—	—	—	RIPL<2:0>											
		7:0	—	—	VEC<5:0>														
0x20	TPTMR <sup>(1,2,3)</sup>	31:24	TPTMR<31:24>																
		23:16	TPTMR<23:16>																
		15:8	TPTMR<15:8>																
		7:0	TPTMR<7:0>																
0x30-0x50	IFSx <sup>(1,2,3)</sup>	31:24	IFS31	IFS30	IFS29	IFS28	IFS27	IFS26	IFS25	IFS24									
		23:16	IFS23	IFS22	IFS21	IFS20	IFS19	IFS18	IFS17	IFS16									
		15:8	IFS15	IFS14	IFS13	IFS12	IFS11	IFS10	IFS09	IFS08									
		7:0	IFS07	IFS06	IFS05	IFS04	IFS03	IFS02	IFS01	IFS00									
0x60-0x80	IECx <sup>(1,2,3)</sup>	31:24	IEC31	IEC30	IEC29	IEC28	IEC27	IEC26	IEC25	IEC24									
		23:16	IEC23	IEC22	IEC21	IEC20	IEC19	IEC18	IEC17	IEC16									
		15:8	IEC15	IEC14	IEC13	IEC12	IEC11	IEC10	IEC09	IEC08									
		7:0	IEC07	IEC06	IEC05	IEC04	IEC03	IEC02	IEC01	IEC00									
0x90-0x180	IPCx <sup>(1,2,3)</sup>	31:24	—	—	—	IP03<2:0>			IS03<1:0>										
		23:16	—	—	—	IP02<2:0>			IS02<1:0>										
		15:8	—	—	—	IP01<2:0>			IS01<1:0>										
		7:0	—	—	—	IP00<2:0>			IS00<1:0>										

### 5.2.3.1. LE REGISTRE INTCON

Ce registre (*INTerrupt CONtrol register*) permet la configuration du mécanisme de traitement des interruptions.

Voici le rôle des différents bits :

bit 31-17	<b>Reserved:</b> Write '0'; ignore read
bit 16	<b>SS0:</b> Single Vector Shadow Register Set bit 1 = Single vector is presented with a shadow register set 0 = Single vector is not presented with a shadow register set
bit 15	<b>Reserved:</b> Write '0'; ignore read
bit 14	<b>FRZ:</b> Freeze in Debug Exception Mode bit 1 = Freeze operation when CPU is in Debug Exception mode 0 = Continue operation even when CPU is in Debug Exception mode  <b>Note:</b> FRZ is writable in Debug Exception mode only, it is forced to '0' in normal mode.
bit 13	<b>Reserved:</b> Write '0'; ignore read
bit 12	<b>MVEC:</b> Multi Vector Configuration bit 1 = Interrupt controller configured for multi vectored mode 0 = Interrupt controller configured for single vectored mode
bit 11	<b>Reserved:</b> Write '0'; ignore read

#### Register 8-1: INTCON: Interrupt Control Register<sup>(1,2,3)</sup> (Continued)

bit 10-8	<b>TPC&lt;2:0&gt;:</b> Temporal Proximity Control bits  111 = Interrupts of group priority 7 or lower start the TP timer 110 = Interrupts of group priority 6 or lower start the TP timer 101 = Interrupts of group priority 5 or lower start the TP timer 100 = Interrupts of group priority 4 or lower start the TP timer 011 = Interrupts of group priority 3 or lower start the TP timer 010 = Interrupts of group priority 2 or lower start the TP timer 001 = Interrupts of group priority 1 start the IP timer 000 = Disables proximity timer
bit 7-5	<b>Reserved:</b> Write '0'; ignore read
bit 4	<b>INT4EP:</b> External Interrupt 4 Edge Polarity Control bit 1 = Rising edge 0 = Falling edge
bit 3	<b>INT3EP:</b> External Interrupt 3 Edge Polarity Control bit 1 = Rising edge 0 = Falling edge
bit 2	<b>INT2EP:</b> External Interrupt 2 Edge Polarity Control bit 1 = Rising edge 0 = Falling edge
bit 1	<b>INT1EP:</b> External Interrupt 1 Edge Polarity Control bit 1 = Rising edge 0 = Falling edge
bit 0	<b>INT0EP:</b> External Interrupt 0 Edge Polarity Control bit 1 = Rising edge 0 = Falling edge

On y trouve la sélection simple ou multi vecteurs, ainsi que le choix du flanc des interruptions externes.

### 5.2.3.2. LE REGISTRE INTSTAT

Ce registre (*INTerrupt STATus register*) fournit des informations sur la situation des interruptions.

Voici le rôle de chacun des bits composant ce registre.

bit 31-11	<b>Reserved:</b> Write '0'; ignore read
bit 10-8	<b>RIPL&lt;2:0&gt;:</b> Requested Priority Level bits 000-111 = The priority level of the latest interrupt presented to the CPU  <b>Note:</b> This value should only be used when the interrupt controller is configured for Single Vector mode.
bit 7-6	<b>Reserved:</b> Write '0'; ignore read
bit 5-0	<b>VEC&lt;5:0&gt;:</b> Interrupt Vector bits 00000-11111 = The interrupt vector that is presented to the CPU  <b>Note:</b> This value should only be used when the interrupt controller is configured for Single Vector mode.

### 5.2.3.3. LES REGISTRES IFSX

Ces registres (*Interrupt Flag Status register*) fournissent des informations sur la situation des demandes d'interruptions. Lors d'une demande d'interruption, le bit correspondant à la source est mis à "1".

### 5.2.3.4. LES REGISTRES IECX

Ces registres (*Interrupt Enable Control register*) établissent l'autorisation d'interruption pour chaque source.

bit 31-0	<b>IEC31-IEC00:</b> Interrupt Enable bits 1 = Interrupt is enabled 0 = Interrupt is disabled
----------	--

### 5.2.3.5. LES REGISTRES IPCX

Ces registres (*Interrupt Priority Control register*) établissent la situation de priorité et de sous-priorité pour chaque source. Quatre sources par registres.

Register 8-6: IPCx: Interrupt Priority Control Register<sup>(1,2,3,4)</sup>

R-X	R-X	R-X	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	—		IP03<2:0>		IS03<1:0>	
bit 31							bit 24

bit 31-29	<b>Reserved:</b> Write '0'; ignore read
bit 28-26	<b>IP03&lt;2:0&gt;:</b> Interrupt Priority bits 111 = Interrupt priority is 7 110 = Interrupt priority is 6 101 = Interrupt priority is 5 100 = Interrupt priority is 4 011 = Interrupt priority is 3 010 = Interrupt priority is 2 001 = Interrupt priority is 1 000 = Interrupt is disabled
bit 25-24	<b>IS03&lt;1:0&gt;:</b> Interrupt Subpriority bits 11 = Interrupt subpriority is 3 10 = Interrupt subpriority is 2 01 = Interrupt subpriority is 1 00 = Interrupt subpriority is 0

### 5.2.4. RELATION ENTRE SOURCES D'INTERRUPTION ET REGISTRES

TABLE 7-1: INTERRUPT IRQ, VECTOR AND BIT LOCATION

Interrupt Source <sup>(1)</sup>	IRQ	Vector Number	Interrupt Bit Location			
			Flag	Enable	Priority	Sub-Priority
Highest Natural Order Priority						
CT – Core Timer Interrupt	0	0	IFS0<0>	IEC0<0>	IPC0<4:2>	IPC0<1:0>
CS0 – Core Software Interrupt 0	1	1	IFS0<1>	IEC0<1>	IPC0<12:10>	IPC0<9:8>
CS1 – Core Software Interrupt 1	2	2	IFS0<2>	IEC0<2>	IPC0<20:18>	IPC0<17:16>
INT0 – External Interrupt 0	3	3	IFS0<3>	IEC0<3>	IPC0<28:26>	IPC0<25:24>
T1 – Timer1	4	4	IFS0<4>	IEC0<4>	IPC1<4:2>	IPC1<1:0>
IC1 – Input Capture 1	5	5	IFS0<5>	IEC0<5>	IPC1<12:10>	IPC1<9:8>
OC1 – Output Compare 1	6	6	IFS0<6>	IEC0<6>	IPC1<20:18>	IPC1<17:16>
INT1 – External Interrupt 1	7	7	IFS0<7>	IEC0<7>	IPC1<28:26>	IPC1<25:24>
T2 – Timer2	8	8	IFS0<8>	IEC0<8>	IPC2<4:2>	IPC2<1:0>
IC2 – Input Capture 2	9	9	IFS0<9>	IEC0<9>	IPC2<12:10>	IPC2<9:8>
OC2 – Output Compare 2	10	10	IFS0<10>	IEC0<10>	IPC2<20:18>	IPC2<17:16>
INT2 – External Interrupt 2	11	11	IFS0<11>	IEC0<11>	IPC2<28:26>	IPC2<25:24>
T3 – Timer3	12	12	IFS0<12>	IEC0<12>	IPC3<4:2>	IPC3<1:0>
IC3 – Input Capture 3	13	13	IFS0<13>	IEC0<13>	IPC3<12:10>	IPC3<9:8>
OC3 – Output Compare 3	14	14	IFS0<14>	IEC0<14>	IPC3<20:18>	IPC3<17:16>
INT3 – External Interrupt 3	15	15	IFS0<15>	IEC0<15>	IPC3<28:26>	IPC3<25:24>
T4 – Timer4	16	16	IFS0<16>	IEC0<16>	IPC4<4:2>	IPC4<1:0>
IC4 – Input Capture 4	17	17	IFS0<17>	IEC0<17>	IPC4<12:10>	IPC4<9:8>
OC4 – Output Compare 4	18	18	IFS0<18>	IEC0<18>	IPC4<20:18>	IPC4<17:16>
INT4 – External Interrupt 4	19	19	IFS0<19>	IEC0<19>	IPC4<28:26>	IPC4<25:24>
T5 – Timer5	20	20	IFS0<20>	IEC0<20>	IPC5<4:2>	IPC5<1:0>
IC5 – Input Capture 5	21	21	IFS0<21>	IEC0<21>	IPC5<12:10>	IPC5<9:8>
OC5 – Output Compare 5	22	22	IFS0<22>	IEC0<22>	IPC5<20:18>	IPC5<17:16>
SPI1E – SPI1 Fault	23	23	IFS0<23>	IEC0<23>	IPC5<28:26>	IPC5<25:24>
SPI1RX – SPI1 Receive Done	24	23	IFS0<24>	IEC0<24>	IPC5<28:26>	IPC5<25:24>
SPI1TX – SPI1 Transfer Done	25	23	IFS0<25>	IEC0<25>	IPC5<28:26>	IPC5<25:24>
U1E – UART1 Error						
SPI3E – SPI3 Fault	26	24	IFS0<26>	IEC0<26>	IPC6<4:2>	IPC6<1:0>
I2C3B – I2C3 Bus Collision Event						
U1RX – UART1 Receiver	27	24	IFS0<27>	IEC0<27>	IPC6<4:2>	IPC6<1:0>
I2C3S – I2C3 Slave Event						
U1TX – UART1 Transmitter						
SPI3TX – SPI3 Transfer Done	28	24	IFS0<28>	IEC0<28>	IPC6<4:2>	IPC6<1:0>
I2C3M – I2C3 Master Event						
I2C1B – I2C1 Bus Collision Event	29	25	IFS0<29>	IEC0<29>	IPC6<12:10>	IPC6<9:8>
I2C1S – I2C1 Slave Event	30	25	IFS0<30>	IEC0<30>	IPC6<12:10>	IPC6<9:8>
I2C1M – I2C1 Master Event	31	25	IFS0<31>	IEC0<31>	IPC6<12:10>	IPC6<9:8>
CN – Input Change Interrupt	32	26	IFS1<0>	IEC1<0>	IPC6<20:18>	IPC6<17:16>
AD1 – ADC1 Convert Done	33	27	IFS1<1>	IEC1<1>	IPC6<28:26>	IPC6<25:24>

Note 1: Not all interrupt sources are available on all devices. See Table 1, Table 2 and Table 3 for the list of available peripherals.

**TABLE 7-1: INTERRUPT IRQ, VECTOR AND BIT LOCATION (CONTINUED)**

Interrupt Source <sup>(1)</sup>	IRQ	Vector Number	Interrupt Bit Location			
			Flag	Enable	Priority	Sub-Priority
PMP – Parallel Master Port	34	28	IFS1<2>	IEC1<2>	IPC7<4:2>	IPC7<1:0>
CMP1 – Comparator Interrupt	35	29	IFS1<3>	IEC1<3>	IPC7<12:10>	IPC7<9:8>
CMP2 – Comparator Interrupt	36	30	IFS1<4>	IEC1<4>	IPC7<20:18>	IPC7<17:16>
U3E – UART2A Error SPI2E – SPI2 Fault I2C4B – I2C4 Bus Collision Event	37	31	IFS1<5>	IEC1<5>	IPC7<28:26>	IPC7<25:24>
U3RX – UART2A Receiver SPI2RX – SPI2 Receive Done I2C4S – I2C4 Slave Event	38	31	IFS1<6>	IEC1<6>	IPC7<28:26>	IPC7<25:24>
U3TX – UART2A Transmitter SPI2TX – SPI2 Transfer Done IC4M – I2C4 Master Event	39	31	IFS1<7>	IEC1<7>	IPC7<28:26>	IPC7<25:24>
U2E – UART3A Error SPI4E – SPI4 Fault I2C5B – I2C5 Bus Collision Event	40	32	IFS1<8>	IEC1<8>	IPC8<4:2>	IPC8<1:0>
U2RX – UART3A Receiver SPI4RX – SPI4 Receive Done I2C5S – I2C5 Slave Event	41	32	IFS1<9>	IEC1<9>	IPC8<4:2>	IPC8<1:0>
U2TX – UART3A Transmitter SPI4TX – SPI4 Transfer Done IC5M – I2C5 Master Event	42	32	IFS1<10>	IEC1<10>	IPC8<4:2>	IPC8<1:0>
I2C2B – I2C2 Bus Collision Event	43	33	IFS1<11>	IEC1<11>	IPC8<12:10>	IPC8<9:8>
I2C2S – I2C2 Slave Event	44	33	IFS1<12>	IEC1<12>	IPC8<12:10>	IPC8<9:8>
I2C2M – I2C2 Master Event	45	33	IFS1<13>	IEC1<13>	IPC8<12:10>	IPC8<9:8>
FSCM – Fail-Safe Clock Monitor	46	34	IFS1<14>	IEC1<14>	IPC8<20:18>	IPC8<17:16>
RTCC – Real-Time Clock and Calendar	47	35	IFS1<15>	IEC1<15>	IPC8<28:26>	IPC8<25:24>
DMA0 – DMA Channel 0	48	36	IFS1<16>	IEC1<16>	IPC9<4:2>	IPC9<1:0>
DMA1 – DMA Channel 1	49	37	IFS1<17>	IEC1<17>	IPC9<12:10>	IPC9<9:8>
DMA2 – DMA Channel 2	50	38	IFS1<18>	IEC1<18>	IPC9<20:18>	IPC9<17:16>
DMA3 – DMA Channel 3	51	39	IFS1<19>	IEC1<19>	IPC9<28:26>	IPC9<25:24>
DMA4 – DMA Channel 4	52	40	IFS1<20>	IEC1<20>	IPC10<4:2>	IPC10<1:0>
DMA5 – DMA Channel 5	53	41	IFS1<21>	IEC1<21>	IPC10<12:10>	IPC10<9:8>
DMA6 – DMA Channel 6	54	42	IFS1<22>	IEC1<22>	IPC10<20:18>	IPC10<17:16>
DMA7 – DMA Channel 7	55	43	IFS1<23>	IEC1<23>	IPC10<28:26>	IPC10<25:24>
FCE – Flash Control Event	56	44	IFS1<24>	IEC1<24>	IPC11<4:2>	IPC11<1:0>
USB – USB Interrupt	57	45	IFS1<25>	IEC1<25>	IPC11<12:10>	IPC11<9:8>
CAN1 – Control Area Network 1	58	46	IFS1<26>	IEC1<26>	IPC11<20:18>	IPC11<17:16>
CAN2 – Control Area Network 2	59	47	IFS1<27>	IEC1<27>	IPC11<28:26>	IPC11<25:24>
ETH – Ethernet Interrupt	60	48	IFS1<28>	IEC1<28>	IPC12<4:2>	IPC12<1:0>
IC1E – Input Capture 1 Error	61	5	IFS1<29>	IEC1<29>	IPC1<12:10>	IPC1<9:8>
IC2E – Input Capture 2 Error	62	9	IFS1<30>	IEC1<30>	IPC2<12:10>	IPC2<9:8>
IC3E – Input Capture 3 Error	63	13	IFS1<31>	IEC1<31>	IPC3<12:10>	IPC3<9:8>
IC4E – Input Capture 4 Error	64	17	IFS2<0>	IEC2<0>	IPC4<12:10>	IPC4<9:8>

Note 1: Not all interrupt sources are available on all devices. See [Table 1](#), [Table 2](#) and [Table 3](#) for the list of available peripherals.

**TABLE 7-1: INTERRUPT IRQ, VECTOR AND BIT LOCATION (CONTINUED)**

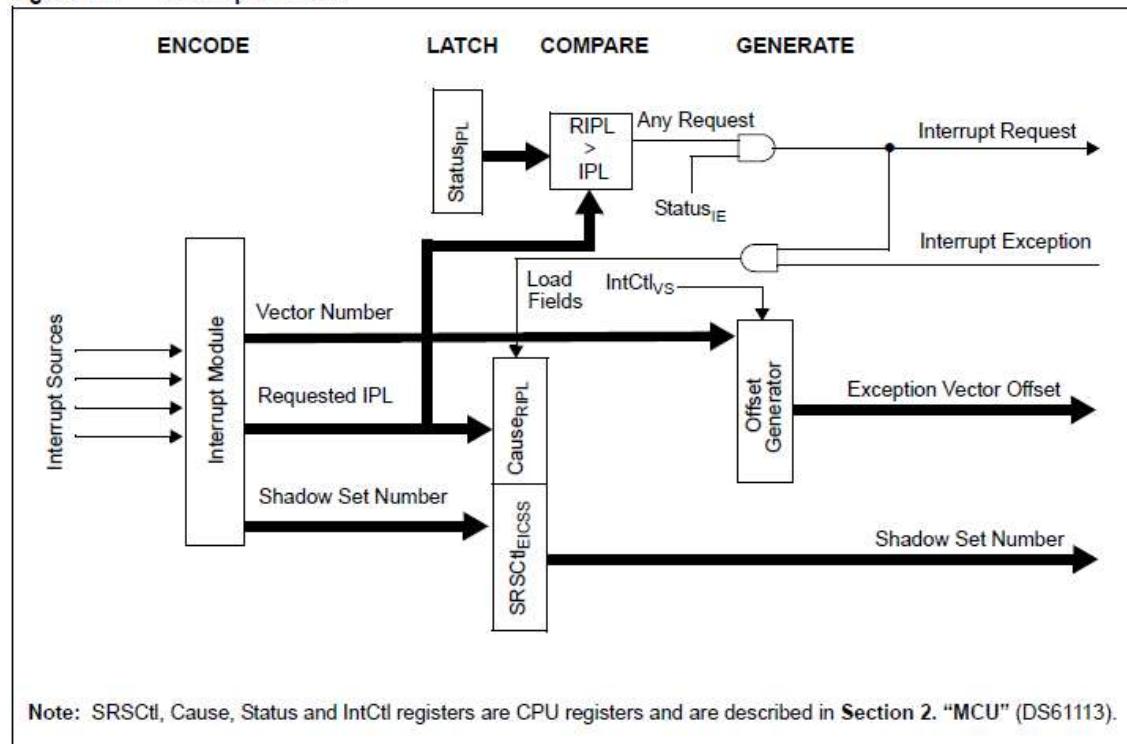
Interrupt Source <sup>(1)</sup>	IRQ	Vector Number	Interrupt Bit Location			
			Flag	Enable	Priority	Sub-Priority
IC4E – Input Capture 5 Error	65	21	IFS2<1>	IEC2<1>	IPC5<12:10>	IPC5<9:8>
PMPE – Parallel Master Port Error	66	28	IFS2<2>	IEC2<2>	IPC7<4:2>	IPC7<1:0>
U4E – UART4 Error	67	49	IFS2<3>	IEC2<3>	IPC12<12:10>	IPC12<9:8>
U4RX – UART4 Receiver	68	49	IFS2<4>	IEC2<4>	IPC12<12:10>	IPC12<9:8>
U4TX – UART4 Transmitter	69	49	IFS2<5>	IEC2<5>	IPC12<12:10>	IPC12<9:8>
U6E – UART6 Error	70	50	IFS2<6>	IEC2<6>	IPC12<20:18>	IPC12<17:16>
U6RX – UART6 Receiver	71	50	IFS2<7>	IEC2<7>	IPC12<20:18>	IPC12<17:16>
U6TX – UART6 Transmitter	72	50	IFS2<8>	IEC2<8>	IPC12<20:18>	IPC12<17:16>
U5E – UART5 Error	73	51	IFS2<9>	IEC2<9>	IPC12<28:26>	IPC12<25:24>
U5RX – UART5 Receiver	74	51	IFS2<10>	IEC2<10>	IPC12<28:26>	IPC12<25:24>
U5TX – UART5 Transmitter	75	51	IFS2<11>	IEC2<11>	IPC12<28:26>	IPC12<25:24>
(Reserved)	—	—	—	—	—	—
Lowest Natural Order Priority						

Note 1: Not all interrupt sources are available on all devices. See [Table 1](#), [Table 2](#) and [Table 3](#) for the list of available peripherals.

### 5.2.5. MÉCANISME DE TRAITEMENT DES INTERRUPTIONS

La figure 8-2 illustre le mécanisme de traitement des sources d'interruptions :

Figure 8-2: Interrupt Process



Voici la traduction du paragraphe 8.3 du "PIC32 family reference manual", qui accompagne la figure 8-2 :

Toutes les demandes d'interruption (IRQs) sont échantillonnées au flanc montant de SYCLK et sont mémorisées dans les registres IFSx correspondant. La demande d'interruption (*pending IRQ*) est indiquée par la mise à 1 du bit dans un des registres IFSx.

La demande d'interruption n'est prise en compte que si le bit correspondant dans un des registres d'autorisation (interrupt enable (IECx) register) est à 1. Lorsque la demande est recevable, elle est encodée en un numéro de vecteur (0 à 63).

Remarque : comme il y a plus de sources d'interruptions que de vecteurs, certaines sources d'interruptions partagent le même no de vecteur (se référer à la table 7-1, §5.2.4 ci-dessus).

A chaque no de vecteur correspond un niveau de priorité (***Requested IPL***) et un "***shadow set number***". Le niveau de priorité est déterminé par le contenu du registre IPCx associé à ce vecteur. En mode multi-vecteur l'utilisateur peut sélectionner un niveau de priorité qui recevra un jeu de "*shadow register*" correspondant. En mode simple vecteur, toutes les interruptions peuvent avoir un jeu de "*shadow register*" dédié.

Le contrôleur d'interruption sélectionne parmi les demandes celle qui a la plus haute priorité et fournit au coeur du processeur (*processor core*) les 3 informations correspondantes (numéro de vecteur, niveau de priorité et no de jeu de "*shadow register*").

Si la priorité du vecteur fourni au coeur est plus élevée que la priorité courante indiquée par le "*CPU Interrupt Priority bits IPL*" (Status<12:10>), alors l'interruption est traitée. Autrement la demande reste en attente (pending) jusqu'à ce que la priorité en cours soit plus faible que la demande.

Lorsque le coeur du processeur traite une interruption, le PC (Program Counter) est copié dans le EPC (Exception Program Counter) et le "*Exception Level (EXL) bit*" (Status<1>) est mis à 1. La mise à un du bit EXL a pour effet de désactiver la prise en compte d'autres interruptions jusqu'à ce que le bit EXL soit remis à 0. Ensuite, le coeur modifie le PC pour se brancher à l'adresse du vecteur calculée à partir du no du vecteur. En regardant de plus près un prologue d'interruption généré par le compilateur xc32, on trouve effectivement entre autres, le réglage du nouveau niveau de priorité du CPU (correspondant à l'interruption), puis le clear du bit EXL. De cette manière, le CPU peut à nouveau être interrompu par une interruption plus prioritaire

Le registre INTSTAT contient le numéro du vecteur d'interruption (bits VEC, INTSTAT<5:0>) et le niveau de priorité demandée (bits RIPL, INTSTAT<10:8>), ceci pour la demande en attente. Ces valeurs peuvent différer de celles qui ont amené le coeur à commuter sur le traitement d'exception.

A la fin de la réponse à l'interruption, le processeur retourne à l'état précédent lorsque l'instruction **ERET** (Exception Return) est exécutée. L'instruction ERET remet à zéro le bit EXL, restaure le PC et commute le jeu de "*shadow register*" sur le précédent.

### 5.2.5.1. A PROPOS DES SHADOW REGISTERS

L'architecture du PIC32 met à disposition 2 groupes de registres GPR. Cela permet, par exemple lors du saut dans une routine d'interruption, de switcher de banc de registres au lieu de sauver une partie des registres généraux. Le prologue et l'épilogue sont alors raccourcis, résultant en un code plus rapide.

### 5.2.6. CALCUL DE L'ADRESSE DU VECTEUR D'INTERRUPTION

Le calcul est différent entre le mode simple vecteur ou multiple vecteur. Il s'appuie sur la valeur du registre CPU EBASE (Exception Base (EBase<31:12>), qui fournit une adresse de base alignée sur des pages de 4 KB située dans le segment de noyau (KSEG Kernel SEGment).

#### 5.2.6.1. VALEURS UTILISÉES PAR LE COMPILATEUR

Le fichier script de link par défaut p32MX795F512L.ld (spécifique à chaque uC - trouvable dans les sous-répertoires du compilateur xc32) fournit les valeurs par défaut pour notre processeur :

- vector\_spacing = 0x00000001
- ebase\_address = 0x9FC01000

#### 5.2.6.2. ADRESSE DES VECTEURS

Un fichier .map de diagnostic et généré à la compilation du projet. On y trouve la liste des vecteurs d'interruption. Voici les adresses des quelques premiers :

section	address	length [bytes]	(dec)	Description
.app_excpt	0x9fc01180	0x10	16	General-Exception
.vector_0	0x9fc01200	0x8	8	Interrupt Vector 0
.vector_1	0x9fc01220	0x8	8	Interrupt Vector 1
.vector_2	0x9fc01240	0x8	8	Interrupt Vector 2
.vector_3	0x9fc01260	0x8	8	Interrupt Vector 3
<b>.vector_4</b>	<b>0x9fc01280</b>	<b>0x8</b>	<b>8</b>	<b>Interrupt Vector 4</b>
.vector_5	0x9fc012a0	0x8	8	Interrupt Vector 5
.vector_6	0x9fc012c0	0x8	8	Interrupt Vector 6
.vector_7	0x9fc012e0	0x8	8	Interrupt Vector 7

#### 5.2.6.3. CALCUL EN MODE SIMPLE VECTEUR

Adresse unique donnée par la formule :

$$\text{Single Vector Address} = \text{EBase} + 0x200$$

Avec Exception Base = 0x9FC01000, on obtient 0x9FC01200.

### 5.2.6.4. CALCUL EN MODE MULTIPLE VECTEUR

L'adresse en mode multi vecteur est calculée à partir des valeurs de EBase et VS (IntCtl<9:5>) (les registres IntCtl et le Status sont situés dans le CPU). Les bits VS (Vector Spacing) fournissent l'espace entre les adresses de 2 vecteurs adjacents.

L'espace entre les adresses de vecteur peut être de 32, 64, 128, 256 et 512 bytes.

Le principe de calcul est le suivant :

$$\text{Vector address} = \text{vector number} * (\text{VS} << 5) + 0x200 + \text{vector base}$$

Par exemple pour le vecteur numéro 4 (timer 1) :

$$\text{Exception Base} = 0x9FC01000$$

$$\text{Vector Spacing(VS)} = 1 : (1 << 5) = 32 (0x20)$$

$$\text{Vector address(Timer1)} = 4 * 0x20 + 0x200 + 0x9FC01000 = 0x9FC01280$$

Ce qui correspond à la table ci-dessus.

### 5.2.7. CONTENU DES VECTEURS D'INTERRUPTION

Les vecteurs d'interruption, vu leur espacement modeste, vont contenir une instruction de saut à la routine de réponse à l'interruption. Une routine très courte peut éventuellement directement y être logée. Il serait également possible de les espacer davantage.

### 5.2.8. CONTENU SYSTÈME DE LA RÉPONSE À L'INTERRUPTION

Le compilateur va ajouter au code visible de la réponse à l'interruption, les instructions nécessaires à la sauvegarde de certains registres, à l'exécution de la routine de réponse à l'interruption, puis à la restauration des registres sauvegardés et pour finir l'instruction RET.

En mode simple vecteur, le compilateur devra générer les instructions nécessaires à la sélection de la routine de réponse en relation avec la source d'interruption.

En mode multi vecteur la sélection de la source sera limitée aux différentes sources pour un même vecteur d'où un temps de traitement plus court.

### 5.2.9. PRIORITÉS DES INTERRUPTIONS

Dans le mode multi vecteur, l'utilisateur peut assigner un niveau de priorité et un niveau de sous-priorité pour chacun des vecteurs d'interruptions. Le niveau de priorité va de 1 (la priorité la plus basse) à 7 (la plus haute).

Si la priorité est établie à 0, le vecteur d'interruption est invalidé pour l'interruption et le mécanisme de réveil (*wake-up purposes*).

Si une interruption survient avec un niveau de priorité plus élevé que celui en cours, la nouvelle interruption est prise en compte et interrompt la routine de réponse à l'interruption de niveau inférieur.

#### 5.2.9.1. CAS DES SOUS-PRIORITÉS

Dans le cas de **demandes simultanées** de deux interruptions avec le même niveau de priorité, la demande avec le niveau de sous-priorité le plus élevé sera traité en premier. Le niveau de sous-priorité va de 0 (la plus basse sous-priorité) à 3 (la plus haute).

⊗ La sous-priorité ne crée pas de niveau de priorité intermédiaire, c'est uniquement un arbitrage lors de demande simultanée.

## 5.3. GESTION DES INTERRUPTIONS AVEC HARMONY & XC32

Le compilateur XC32 fournit un certain nombre de macros et fonctions pour configurer les interruptions et agir dans la réponse sur le flag d'interruption. Il offre aussi des directives pour transformer une fonction en une routine de réponse à une interruption donnée.

Cette partie se réfère à la documentation de Harmony, disponible dans le fichier <Répertoire Harmony>v1\_<n>\doc\help\_harmony.pdf, section *Framework Libraries Help > Interrupt Peripheral Library* (plib\_int). Elle a été réalisée sur la base de la version Harmony 1.08.

### 5.3.1. FICHIER À INCLUDE

Pour utiliser les fonctions de la PLIB\_INT il est nécessaire d'inclure le fichier **plib\_int.h**.

### 5.3.2. SÉLECTION DU MODE

On dispose de 2 fonctions pour la sélection du mode multi ou single vector :

	<code>PLIB_INT_MultiVectorSelect</code>	Configures the Interrupt Controller for Multiple Vector mode.
	<code>PLIB_INT_SingleVectorSelect</code>	Configures the Interrupt Controller for Single Vector mode.

#### 5.3.2.1. PLIB\_INT\_MULTIVECTORSELECT

```
void PLIB_INT_MultiVectorSelect(INT_MODULE_ID index);
```

Cette fonction configure le mécanisme des interruptions avec l'utilisation des différents vecteurs à disposition.

Dans un projet obtenu avec le configurateur Harmony, cette fonction est appelée dans la fonction `SYS_INT_Initialize`.

```
void SYS_INT_Initialize ( void )
{
    /* enable the multi vector */
    PLIB_INT_MultiVectorSelect( INT_ID_0 );
}
```

Il est à noter que la valeur de l'index doit être de `INT_ID_0` pour les PIC32MX, qui ne possèdent qu'un seul module de gestion des interruptions.

#### 5.3.2.2. PLIB\_INT\_SINGLEVECTORSELECT

```
void PLIB_INT_SingleVectorSelect(INT_MODULE_ID index);
```

Cette fonction configure le mécanisme des interruptions avec l'utilisation d'un vecteur unique. La valeur d'index est aussi `INT_ID_0`.

### 5.3.3. CONFIGURATION DE L'INTERRUPTION

La configuration d'une interruption correspond à 3 actions :

- Autoriser la source d'interruption,
- Etablir le niveau de priorité,
- Etablir le niveau de sous-priorité.

A chaque action correspond une fonction.

#### 5.3.3.1. PLIB\_INT\_SOURCEENABLE

```
void PLIB_INT_SourceEnable(INT_MODULE_ID index, INT_SOURCE source);
```

Cette fonction autorise la source de l'interruption.

Le type énuméré **INT\_SOURCE** liste les sources possibles. Voici le début et la fin de la liste :

```
typedef enum {
    INT_SOURCE_SOFTWARE_0,
    INT_SOURCE_SOFTWARE_1,
    INT_SOURCE_EXTERNAL_0,
    INT_SOURCE_EXTERNAL_1,
    INT_SOURCE_EXTERNAL_2,
    INT_SOURCE_EXTERNAL_3,
    INT_SOURCE_EXTERNAL_4,
    INT_SOURCE_TIMER_CORE,
    INT_SOURCE_TIMER_1,
    INT_SOURCE_TIMER_2,
    INT_SOURCE_TIMER_3,
    INT_SOURCE_TIMER_4,
    INT_SOURCE_TIMER_5,
    INT_SOURCE_INPUT_CAPTURE_1,
    INT_SOURCE_INPUT_CAPTURE_1_ERROR,
    INT_SOURCE_INPUT_CAPTURE_2,
    INT_SOURCE_INPUT_CAPTURE_2_ERROR,
    INT_SOURCE_INPUT_CAPTURE_3,
    INT_SOURCE_INPUT_CAPTURE_3_ERROR,
    INT_SOURCE_INPUT_CAPTURE_4,
    INT_SOURCE_INPUT_CAPTURE_4_ERROR,
    INT_SOURCE_INPUT_CAPTURE_5,
    INT_SOURCE_INPUT_CAPTURE_5_ERROR,
    INT_SOURCE_OUTPUT_COMPARE_1,
    INT_SOURCE_OUTPUT_COMPARE_2,
    INT_SOURCE_OUTPUT_COMPARE_3,
    INT_SOURCE_OUTPUT_COMPARE_4,
    INT_SOURCE_OUTPUT_COMPARE_5,
    INT_SOURCE_SPI_1_ERROR,
    INT_SOURCE_SPI_1_RECEIVE,
    INT_SOURCE_SPI_1_TRANSMIT,
    INT_SOURCE_SPI_2_ERROR,
    INT_SOURCE_SPI_2_RECEIVE,
    INT_SOURCE_SPI_2_TRANSMIT,
    INT_SOURCE_SPI_3_ERROR,
    INT_SOURCE_SPI_3_RECEIVE,
    INT_SOURCE_SPI_3_TRANSMIT,
    INT_SOURCE_I2C_1_MASTER,
    INT_SOURCE_I2C_2_ERROR,
    INT_SOURCE_I2C_2_SLAVE,
    INT_SOURCE_I2C_2_MASTER,
    INT_SOURCE_I2C_3_ERROR,
    INT_SOURCE_I2C_3_SLAVE,
    INT_SOURCE_I2C_3_MASTER,
    INT_SOURCE_I2C_4_ERROR,
    INT_SOURCE_I2C_4_SLAVE,
    INT_SOURCE_I2C_4_MASTER,
    INT_SOURCE_I2C_5_ERROR,
    INT_SOURCE_I2C_5_SLAVE,
    INT_SOURCE_I2C_5_MASTER,
    INT_SOURCE_CHANGE_NOTICE,
    INT_SOURCE_ADC_1,
    INT_SOURCE_PARALLEL_PORT,
    INT_SOURCE_PARALLEL_PORT_ERROR,
    INT_SOURCE_COMPARATOR_1,
    INT_SOURCE_COMPARATOR_2,
    INT_SOURCE_CLOCK_MONITOR,
    INT_SOURCE_RTCC,
    INT_SOURCE_DMA_0,
    INT_SOURCE_DMA_1,
    INT_SOURCE_DMA_2,
    INT_SOURCE_DMA_3,
    INT_SOURCE_DMA_4,
    INT_SOURCE_DMA_5,
    INT_SOURCE_DMA_6,
    INT_SOURCE_DMA_7,
    INT_SOURCE_FLASH_CONTROL,
    INT_SOURCE_USB_1,
    INT_SOURCE_CAN_1,
    INT_SOURCE_CAN_2,
    INT_SOURCE_ETH_1
} INT_SOURCE;
```

### 5.3.3.2. PLIB\_INT\_VECTORPRIORITYSET

```
void PLIB_INT_VectorPrioritySet(INT_MODULE_ID index, INT_VECTOR vector, INT_PRIORITY_LEVEL priority);
```

Cette fonction établit le niveau de priorité du vecteur d'interruption.

☞ Il ne faut pas confondre SOURCE d'interruption et VECTEUR d'interruption. Par contre il doit y avoir correspondance entre la source et le vecteur.

Le type énuméré **INT\_VECTOR** liste les vecteurs à disposition. Voici le début et la fin de la liste :

```
typedef enum {
    INT_VECTOR_CT,
    INT_VECTOR_CS0,
    INT_VECTOR_CS1,
    INT_VECTOR_INT0,
    INT_VECTOR_T1,
    INT_VECTOR_IC1,
    INT_VECTOR_OC1,
    INT_VECTOR_INT1,
    INT_VECTOR_T2,
    INT_VECTOR_IC2,
    INT_VECTOR_OC2,
    INT_VECTOR_INT2,
    INT_VECTOR_T3,
    INT_VECTOR_IC3,
    INT_VECTOR_OC3,
    INT_VECTOR_INT3,
    INT_VECTOR_T4,
    INT_VECTOR_IC4,
    INT_VECTOR_OC4,
    INT_VECTOR_INT4,
    INT_VECTOR_T5,
    INT_VECTOR_IC5,
    INT_VECTOR_UART2,
    INT_VECTOR_SPI4,
    INT_VECTOR_I2C5,
    INT_VECTOR_I2C2,
    INT_VECTOR_FSCM,
    INT_VECTOR_RTCC,
    INT_VECTOR_DMA0,
    INT_VECTOR_DMA1,
    INT_VECTOR_DMA2,
    INT_VECTOR_DMA3,
    INT_VECTOR_DMA4,
    INT_VECTOR_DMA5,
    INT_VECTOR_DMA6,
    INT_VECTOR_DMA7,
    INT_VECTOR_FCE,
    INT_VECTOR_USB,
    INT_VECTOR_CAN1,
    INT_VECTOR_CAN2,
    INT_VECTOR_ETH,
    INT_VECTOR_UART4,
    INT_VECTOR_UART6,
    INT_VECTOR_UART5
} INT_VECTOR;
```

Le type énuméré **INT\_PRIORITY\_LEVEL** liste les niveaux de priorité à disposition :

```
typedef enum {
    INT_PRIORITY_LEVEL0,
    INT_PRIORITY_LEVEL1,
    INT_PRIORITY_LEVEL2,
    INT_PRIORITY_LEVEL3,
    INT_PRIORITY_LEVEL4,
    INT_PRIORITY_LEVEL5,
    INT_PRIORITY_LEVEL6,
    INT_PRIORITY_LEVEL7
} INT_PRIORITY_LEVEL;
```

Le LEVEL0 est indiqué Disabled !

Members	Description
INT_PRIORITY_LEVEL0	Disabled
INT_PRIORITY_LEVEL1	Priority 1
INT_PRIORITY_LEVEL2	Priority 2

☞ Le niveau 1 est la plus basse priorité tandis que le niveau 7 est la plus haute priorité.

### 5.3.3.3. PLIB\_INT\_VECTORSUBPRIORITYSET

```
void PLIB_INT_VectorSubPrioritySet(INT_MODULE_ID index, INT_VECTOR vector,
INT_SUBPRIORITY_LEVEL subPriority);
```

Cette fonction établit le niveau de sous-priorité du vecteur d'interruption.

Le type énuméré **INT\_SUBPRIORITY\_LEVEL** liste les niveaux de priorité à disposition :

```
typedef enum {
    INT_SUBPRIORITY_LEVEL0,
    INT_SUBPRIORITY_LEVEL1,
    INT_SUBPRIORITY_LEVEL2,
    INT_SUBPRIORITY_LEVEL3
} INT_SUBPRIORITY_LEVEL;
```

La plus basse sous-priorité est le niveau 0 et la plus haute sous-priorité est le niveau 3.

### 5.3.3.4. EXEMPLE DE CONFIGURATION

Voici comme exemple la configuration de l'interruption du timer1 (obtenu avec le MHC) :

```
/* Setup Interrupt */
PLIB_INT_SourceEnable(INT_ID_0, INT_SOURCE_TIMER_1);
PLIB_INT_VectorPrioritySet(INT_ID_0, INT_VECTOR_T1,
                            INT_PRIORITY_LEVEL3);
PLIB_INT_VectorSubPrioritySet(INT_ID_0, INT_VECTOR_T1,
                            INT_SUBPRIORITY_LEVEL0);
```

On constate qu'au niveau de la source, le timer1 est indiqué par TIMER\_1 alors que pour le vecteur il est indiqué T1.

⚠ Lors de la réalisation de la routine de réponse à l'interruption avec la macro \_\_ISR, il faut impérativement utiliser le même niveau de priorité.

### 5.3.4. RELATION ROUTINE DE RÉPONSE ET VECTEUR

Le compilateur XC32 met à disposition des macros et attributs pour transformer une fonction en une routine de réponse à une interruption et créer le lien avec le vecteur correspondant.

☝ Il est à noter que pour la routine de réponse, les éléments syntaxiques sont uniquement liés au compilateur et non pas à la PLIB\_INT de Harmony.

Le fichier header `<sys\attribs.h>` fournit des macros pour simplifier l'application des *attributes* aux fonctions d'interruption. On trouve aussi des "vector macros" dans les fichiers d'entête propres à un modèle de processeur.

On trouve les fichiers d'entête des processeurs du compilateur XC32 sous :  
`<Répertoire xc32>\v<n>\pic32mx\include\proc`

On dispose de 2 macros `_ISR`:

- `_ISR(V, IPL)`
- `_ISR_AT_VECTOR(V, IPL)`

#### 5.3.4.1. INDICATION DU NIVEAU DE PRIORITÉ

Pour indiquer le niveau de priorité n (n de 1 à 7), on dispose de 3 formes:

`IPLnSRS`, `IPLnSOFT` et `IPLnAUTO`.

- Avec SRS le compilateur utilisera un Shadow Register Set dans le traitement de sauvegarde et restauration → prologue et épilogue et raccourcis. On peut par exemple imaginer cette stratégie pour une interruption dont le temps de réaction, la fréquence ou encore la durée seraient critiques.

☝ En mode multi vecteurs, qui est le mode que nous utiliserons en pratique, une seule priorité d'interruption peut utiliser le SRS. Le niveau concerné se configure via le MHC, dans `DEVCFG3`.

- Avec SOFT le compilateur effectuera le traitement de sauvegarde restauration en utilisant la pile.
- Avec AUTO le compilateur détermine s'il peut utiliser un SRS ou s'il doit utiliser la pile.

#### 5.3.4.2. SITUATION AVEC HARMONY

Lorsque l'on utilise le MHC, on constate l'utilisation de la macro `_ISR`, mais avec une variante de la macro pour préciser le niveau d'interruption, avec `ipl` en minuscule.  
D'où les 3 formes : `iplnSRS`, `iplnSOFT` et `iplnAUTO`.

☝ Nous utiliserons cette forme par la suite.

### 5.3.4.3. DÉFINITION DES NUMÉROS DE VECTEURS

Dans le fichier p32mx795f512l.h, on trouve toutes les définitions des vecteurs d'interruptions.

⌚ Avec la macro `_ISR`, on doit utiliser la définition des vecteurs qui se trouve dans les fichiers du processeur et non pas le type énuméré de la `PLIB_INT`.

/* Vector Numbers */	
#define _CORE_TIMER_VECTOR	0
#define _CORE_SOFTWARE_0_VECTOR	1
#define _CORE_SOFTWARE_1_VECTOR	2
#define _EXTERNAL_0_VECTOR	3
#define _TIMER_1_VECTOR	4
#define _INPUT_CAPTURE_1_VECTOR	5
#define _OUTPUT_COMPARE_1_VECTOR	6
#define _EXTERNAL_1_VECTOR	7
#define _TIMER_2_VECTOR	8
#define _INPUT_CAPTURE_2_VECTOR	9
#define _OUTPUT_COMPARE_2_VECTOR	10
#define _EXTERNAL_2_VECTOR	11
#define _TIMER_3_VECTOR	12
#define _INPUT_CAPTURE_3_VECTOR	13
#define _OUTPUT_COMPARE_3_VECTOR	14
#define _EXTERNAL_3_VECTOR	15
#define _TIMER_4_VECTOR	16
#define _INPUT_CAPTURE_4_VECTOR	17
#define _OUTPUT_COMPARE_4_VECTOR	18
#define _EXTERNAL_4_VECTOR	19
#define _TIMER_5_VECTOR	20
#define _INPUT_CAPTURE_5_VECTOR	21
#define _OUTPUT_COMPARE_5_VECTOR	22
#define _SPI_1_VECTOR	23
#define _I2C_3_VECTOR	24
#define _I2C_1A_VECTOR	24
#define _SPI_3_VECTOR	24
#define _SPI_1A_VECTOR	24
#define _UART_1_VECTOR	24
#define _UART_1A_VECTOR	24
#define _I2C_1_VECTOR	25
#define _CHANGE_NOTICE_VECTOR	26
#define _ADC_VECTOR	27
#define _PMP_VECTOR	28
#define _COMPARATOR_1_VECTOR	29
#define _COMPARATOR_2_VECTOR	30
#define _I2C_4_VECTOR	31
#define _I2C_2A_VECTOR	31
#define _SPI_2_VECTOR	31
#define _SPI_2A_VECTOR	31
#define _UART_3_VECTOR	31
#define _UART_2A_VECTOR	31
#define _I2C_5_VECTOR	32
#define _I2C_3A_VECTOR	32
#define _SPI_4_VECTOR	32
#define _SPI_3A_VECTOR	32
#define _UART_2_VECTOR	32
#define _UART_3A_VECTOR	32
#define _I2C_2_VECTOR	33
#define _FAIL_SAFE_MONITOR_VECTOR	34
#define _RTCC_VECTOR	35
#define _DMA_0_VECTOR	36
#define _DMA_1_VECTOR	37
#define _DMA_2_VECTOR	38
#define _DMA_3_VECTOR	39
#define _DMA_4_VECTOR	40
#define _DMA_5_VECTOR	41
#define _DMA_6_VECTOR	42
#define _DMA_7_VECTOR	43
#define _USB_1_VECTOR	45
#define _CAN_1_VECTOR	46
#define _CAN_2_VECTOR	47
#define _ETH_VECTOR	48
#define _UART_4_VECTOR	49
#define _UART_1B_VECTOR	49
#define _UART_6_VECTOR	50
#define _UART_2B_VECTOR	50
#define _UART_5_VECTOR	51
#define _UART_3B_VECTOR	51
#define _FCE_VECTOR	44

### 5.3.4.4. DÉTAILS MACRO `_ISR`

La macro `_ISR(V, IPL)` affecte le numéro du vecteur et l'associe avec le niveau de priorité spécifiée. Cette macro a pour effet de placer un saut à la routine de réponse à l'interruption, dans la zone du vecteur correspondant.

#### 5.3.4.4.1. Macro `_ISR`, exemple harmony

L'exemple ci-dessous correspond à la routine de réponse à l'interruption du timer 1 :

```
void __ISR(_TIMER_1_VECTOR, ip13AUTO) Timer1Handler(void)
```

Spécification du niveau de priorité 3 en AUTO.

### 5.3.4.5. DÉTAILS MACRO \_\_ISR\_AT\_VECTOR

La macro \_\_ISR\_AT\_VECTOR(V, IPL) affecte le numéro du vecteur et l'associe avec le niveau de priorité spécifiée. Cette macro a pour effet de placer la routine de réponse à l'interruption directement dans la zone du vecteur correspondant.

Remarque : le Vector Spacing doit être augmenté, ou la routine particulièrement courte, afin de disposer d'assez de place pour le code.

#### 5.3.4.5.1. Macro \_\_ISR\_AT\_VECTOR, exemple

L'exemple ci-dessous correspond à la routine de réponse à l'interruption du timer 2 :

```
void __ISR_AT_VECTOR(_TIMER_2_VECTOR, IPL7SRS)
    Timer2Handler(void)
```

Spécification du niveau de priorité 7 en SRS.

⌚ Provoque une erreur "function at exception vector 8 too large".

Une solution serait de modifier la valeur du Vector Spacing. Le lecteur est invité à se reporter à la documentation du compilateur xc32 pour ceci.

### 5.3.4.6. VECTOR ATTRIBUTE

Une autre façon d'établir la relation de la fonction avec le vecteur est d'utiliser le vector attribute. On obtient le même résultat qu'avec le \_\_ISR, mais la syntaxe est un peu plus lourde. Par exemple :

```
void __attribute__((interrupt(IPL6SOFT))) __attribute__
((vector(_TIMER_3_VECTOR))) Timer3Handler (void)
```

## 5.4. EXEMPLE PRATIQUE AVEC TIMER

Voici un exemple complet utilisant 2 timers avec interruption obtenu avec le Microchip Harmony Configurator. Nous suivrons les détails pour le timer 1 et le timer 4.

### 5.4.1. PRINCIPE DE L'EXEMPLE

Utilisation de 2 timers et 2 sorties, l'action dans les routines de réponse à l'interruption consiste à inverser la led correspondante.



Choix d'une période de 1 ms pour le timer 1 (priorité 3) et de 100 us pour le timer 4 (priorité 7, SRS).

### 5.4.2. CONFIGURATION TIMER 1 ET INTERRUPTION

Dans le fichier system\_init.c, on trouve l'appel de la fonction DRV\_TMR0\_Initialize(), que l'on trouve dans le fichier drv\_tmr\_static.c.

Voici les actions de configuration du timer 1 et de son interruption.

```

void DRV_TMR0_Initialize(void)
{
    /* Initialize Timer Instance0 */
    /* Disable Timer */
    PLIB_TMR_Stop(TMR_ID_1);
    /* Select clock source */
    PLIB_TMR_ClockSourceSelect(TMR_ID_1,
                               DRV_TMR_CLKSOURCE_INTERNAL);
    /* Select prescalar value */
    PLIB_TMR_PrescaleSelect(TMR_ID_1,
                           TMR_PRESCALE_VALUE_8);
    /* Enable 16 bit mode */
    PLIB_TMR_Mode16BitEnable(TMR_ID_1);
    /* Clear counter */
    PLIB_TMR_Counter16BitClear(TMR_ID_1);
    /*Set period */
    PLIB_TMR_Period16BitSet(TMR_ID_1, 10000);
    /* Setup Interrupt */
    PLIB_INT_VectorPrioritySet(INT_ID_0, INT_VECTOR_T1,
                               INT_PRIORITY_LEVEL3);
    PLIB_INT_VectorSubPrioritySet(INT_ID_0, INT_VECTOR_T1,
                               INT_SUBPRIORITY_LEVEL0);
}
    
```

¶ On constate que la source de l'interruption n'est pas autorisée au niveau de la fonction DRV\_TMR0\_Initialize(), de même que le timer 1 n'est pas activé.

#### 5.4.2.1. AUTORISATION DE L'INTERRUPTION LORS DU START

On observe que dans la fonction DRV\_TMR0\_Start() il y a appel de la fonction \_DRV\_TMR0\_Resume() qui s'occupe de l'autorisation de la source de l'interruption.

```
static void _DRV_TMR0_Resume(bool resume)
{
    if (resume)
    {
        PLIB_INT_SourceFlagClear(INT_ID_0,
                                  INT_SOURCE_TIMER_1);
        PLIB_INT_SourceEnable(INT_ID_0,
                              INT_SOURCE_TIMER_1);
        PLIB_TMR_Start(TMR_ID_1);
    }
}

bool DRV_TMR0_Start(void)
{
    /* Start Timer*/
    _DRV_TMR0_Resume(true);
    DRV_TMR0_Running = true;

    return true;
}
```

#### 5.4.3. ROUTINE RÉPONSE INTERRUPTION DU TIMER 1

Dans le fichier system\_interrupt.c on trouve la macro \_\_ISR correspondant au timer1.

```
void __ISR(_TIMER_1_VECTOR, ipl3AUTO)
          IntHandlerDrvTmrInstance0(void)
{
    PLIB_INT_SourceFlagClear(INT_ID_0, INT_SOURCE_TIMER_1);
    // Test de la période du timer
    BSP_LEDToggle(BSP_LED_1);
}
```

Le niveau d'interruption est indiqué avec "ipl3AUTO".

##### 5.4.3.1. MISE À ZÉRO DU FLAG D'INTERRUPTION

On constate qu'il est nécessaire de mettre à 0 le flag d'interruption en utilisant une fonction spécifique de la PLIB\_INT:

```
PLIB_INT_SourceFlagClear(INT_ID_0, INT_SOURCE_TIMER_1);
```

```
void PLIB_INT_SourceFlagClear(INT_MODULE_ID index, INT_SOURCE source);
```

Cette fonction utilise le type énuméré INT\_SOURCE.

## 5.4.4. LISTING ASSEMBLEUR

### 5.4.4.1. CONTENU VECTEUR TIMER 1

Le vecteur du timer1 est le numéro 4, donc son adresse est :

*Exception Base = 0x9FC01000*

*Vector Spacing(VS) = 1, (1 << 5) = 32 (0x20)*

*Vector address(Timer1) = 4 \* 0x20 + 0x200 + 0x9FC01000 = 0x9FC01280*

⊗ Le listing assembleur ne montre pas le contenu des vecteurs. En utilisant le debugger avec l'observation du contenu de la mémoire on n'a pas la possibilité d'accéder à la zone des vecteurs.

### 5.4.4.2. RÉPONSE INTERRUPTION TIMER1

☝ Compilation avec optimisation 0.

On suppose que le contenu du vecteur 4 effectue un saut en 9D003800.

```

74:           void __ISR(_TIMER_1_VECTOR, ipl3AUTO)
                  IntHandlerDrvTmrInstance0(void)
75:           {
9D003800 415DE800  RDPGPR SP, SP
9D003804 401B7000  MFC0 K1, EPC
9D003808 401A6002  MFC0 K0, SRSCtl
9D00380C 27BDFF88  ADDIU SP, SP, -120
9D003810 AFBB0074  SW K1, 116(SP)
9D003814 401B6000  MFC0 K1, Status
9D003818 AFBA006C  SW K0, 108(SP)
9D00381C AFBB0070  SW K1, 112(SP)
9D003820 7C1B7844  INS K1, ZERO, 1, 15
9D003824 377B0C00  ORI K1, K1, 3072
9D003828 409B6000  MTC0 K1, Status
9D00382C AFA3001C  SW V1, 28(SP)
9D003830 AFA20018  SW V0, 24(SP)
9D003834 8FA3006C  LW V1, 108(SP)
9D003838 3063000F  ANDI V1, V1, 15
9D00383C 14600012  BNE V1, ZERO, 0x9D003888
9D003840 00000000  NOP
9D003844 AFBF005C  SW RA, 92(SP)
9D003848 AFBE0058  SW S8, 88(SP)
9D00384C AFB90054  SW T9, 84(SP)
9D003850 AFB80050  SW T8, 80(SP)
9D003854 AFAF004C  SW T7, 76(SP)
9D003858 AFAE0048  SW T6, 72(SP)
9D00385C AFAD0044  SW T5, 68(SP)
9D003860 AFAC0040  SW T4, 64(SP)
9D003864 AFAB003C  SW T3, 60(SP)
9D003868 AFAA0038  SW T2, 56(SP)
9D00386C AFA90034  SW T1, 52(SP)
9D003870 AFA80030  SW T0, 48(SP)
9D003874 AFA7002C  SW A3, 44(SP)
9D003878 AFA60028  SW A2, 40(SP)
9D00387C AFA50024  SW A1, 36(SP)
9D003880 AFA40020  SW A0, 32(SP)

```

```

9D003884 AFA10014 SW AT, 20(SP)
9D003888 00000000 NOP
9D00388C 00001012 MFLO V0
9D003890 AFA20064 SW V0, 100(SP)
9D003894 00001810 MFHI V1
9D003898 AFA30060 SW V1, 96(SP)
9D00389C 03A0F021 ADDU S8, SP, ZERO

```

Cette série d'instructions correspond à la partie du prologue qui s'occupe principalement de gérer l'état du processeur et sauver les registres RA, T9 à T0 et A3 à A0 sur la pile.

```

76: PLIB_INT_SourceFlagClear(INT_ID_0, INT_SOURCE_TIMER_1);
9D0038A0 00002021 ADDU A0, ZERO, ZERO
9D0038A4 24050004 ADDIU A1, ZERO, 4
9D0038A8 0F40163D JAL 0x9D0058F4
9D0038AC 00000000 NOP

77:                                // Test de la période du timer
78:                                BSP_LEDToggle(BSP_LED_1);
9D0038B0 24040001 ADDIU A0, ZERO, 1
9D0038B4 0F4013C5 JAL BSP_LEDToggle
9D0038B8 00000000 NOP

79:                                }
9D0038BC 03C0E821 ADDU SP, S8, ZERO
9D0038C0 8FA20064 LW V0, 100(SP)
9D0038C4 00400013 MTLO V0
9D0038C8 8FA30060 LW V1, 96(SP)
9D0038CC 00600011 MTHI V1
9D0038D0 8FA2006C LW V0, 108(SP)
9D0038D4 3042000F ANDI V0, V0, 15
9D0038D8 14400014 BNE V0, ZERO, 0x9D00392C
9D0038DC 00000000 NOP
9D0038E0 8FBF005C LW RA, 92(SP)
9D0038E4 8FBE0058 LW S8, 88(SP)
9D0038E8 8FB90054 LW T9, 84(SP)
9D0038EC 8FB80050 LW T8, 80(SP)
9D0038F0 8FAF004C LW T7, 76(SP)
9D0038F4 8FAE0048 LW T6, 72(SP)
9D0038F8 8FAD0044 LW T5, 68(SP)
9D0038FC 8FAC0040 LW T4, 64(SP)
9D003900 8FAB003C LW T3, 60(SP)
9D003904 8FAA0038 LW T2, 56(SP)
9D003908 8FA90034 LW T1, 52(SP)
9D00390C 8FA80030 LW T0, 48(SP)
9D003910 8FA7002C LW A3, 44(SP)
9D003914 8FA60028 LW A2, 40(SP)
9D003918 8FA50024 LW A1, 36(SP)
9D00391C 8FA40020 LW A0, 32(SP)
9D003920 8FA3001C LW V1, 28(SP)
9D003924 8FA20018 LW V0, 24(SP)
9D003928 8FA10014 LW AT, 20(SP)
9D00392C 00000000 NOP
9D003930 41606000 DI ZERO
9D003934 000000C0 EHB
9D003938 8FBA0074 LW K0, 116(SP)

```

```

9D00393C  8FBB0070    LW K1, 112(SP)
9D003940  409A7000    MTC0 K0, EPC
9D003944  8FBA006C    LW K0, 108(SP)
9D003948  27BD0078    ADDIU SP, SP, 120
9D00394C  409A6002    MTC0 K0, SRSCtl
9D003950  41DDE800    WRPGPR SP, SP
9D003954  409B6000    MTC0 K1, Status
9D003958  42000018    ERET

```

On trouve dans l'épilogue la restauration des registres T9 à T0, ainsi que A3 à A0. La routine se termine par l'instruction **ERET** alors qu'une simple fonction se termine par JR RA.

#### 5.4.5. CONFIGURATION TIMER 4 ET INTERRUPTION

Dans le fichier system\_init.c, on trouve dans la fonction `DRV_TMR1_Initialize`, les actions de configuration du timer 4 et de son interruption.

```

void DRV_TMR1_Initialize(void)
{
    /* Initialize Timer Instance1 */
    /* Disable Timer */
    PLIB_TMR_Stop(TMR_ID_4);
    /* Select clock source */
    PLIB_TMR_ClockSourceSelect(TMR_ID_4,
                               DRV_TMR_CLKSOURCE_INTERNAL);
    /* Select prescalar value */
    PLIB_TMR_PrescaleSelect(TMR_ID_4,
                           TMR_PRESCALE_VALUE_1);
    /* Enable 16 bit mode */
    PLIB_TMR_Mode16BitEnable(TMR_ID_4);
    /* Clear counter */
    PLIB_TMR_Counter16BitClear(TMR_ID_4);
    /* Set period */
    PLIB_TMR_Period16BitSet(TMR_ID_4, 8000);
    /* Setup Interrupt */
    PLIB_INT_VectorPrioritySet(INT_ID_0, INT_VECTOR_T4,
                               INT_PRIORITY_LEVEL7);
    PLIB_INT_VectorSubPrioritySet(INT_ID_0, INT_VECTOR_T4,
                               INT_SUBPRIORITY_LEVEL0);
}

```

¶ Comme pour le timer 1, le timer 4 n'est pas activé lors de son initialisation.

#### 5.4.5.1. AUTORISATION DE L'INTERRUPTION LORS DU START

A nouveau, comme pour le timer 1, on observe que dans la fonction `DRV_TMR1_Start()` il y a appel de la fonction `_DRV_TMR1_Resume()` qui s'occupe de l'autorisation de la source de l'interruption :

```
static void _DRV_TMR1_Resume(bool resume)
{
    if (resume)
    {
        PLIB_INT_SourceFlagClear(INT_ID_0,
                                  INT_SOURCE_TIMER_4);
        PLIB_INT_SourceEnable(INT_ID_0,
                              INT_SOURCE_TIMER_4);
        PLIB_TMR_Start(TMR_ID_4);
    }
}

bool DRV_TMR1_Start(void)
{
    /* Start Timer*/
    DRV_TMR1_Resume(true);
    DRV_TMR1_Running = true;

    return true;
}
```

#### 5.4.6. ROUTINE RÉPONSE INTERRUPTION DU TIMER 4

Dans le fichier `system_interrupt.c`, on trouve dans la macro `__ISR` correspondant au timer 4. La priorité a été réglée en `ipl7SRS` :

```
void __ISR( _TIMER_4_VECTOR, ipl7SRS)
{
    _IntHandlerDrvTmrInstance1(void)
{
    PLIB_INT_SourceFlagClear(INT_ID_0, INT_SOURCE_TIMER_4);
    // Test de la periode du timer
    BSP_LEDToggle(BSP_LED_4);
}
```

### 5.4.6.1. RÉPONSE INTERRUPTION TIMER4 EN ASSEMBLEUR

Voici la réponse à l'interruption du Timer 4 en assembleur :

```

81:           void __ISR(_TIMER_4_VECTOR, ipl7SRS)
                  IntHandlerDrvTmrInstance1(void)
82:           {
9D004DBC 415DE800 RDPGPR SP, SP
9D004DC0 401A7000 MFC0 K0, EPC
9D004DC4 401B6000 MFC0 K1, Status
9D004DC8 27BDFFD8 ADDIU SP, SP, -40
9D004DCC AFBB0024 SW K1, 36(SP)
9D004DD0 7C1B7844 INS K1, ZERO, 1, 15
9D004DD4 377B1C00 ORI K1, K1, 7168
9D004DD8 409B6000 MTC0 K1, Status
9D004DDC 00001012 MFLO V0
9D004DE0 AFA20014 SW V0, 20(SP)
9D004DE4 00001810 MFHI V1
9D004DE8 AFA30010 SW V1, 16(SP)
9D004DEC 03A0F021 ADDU S8, SP, ZERO
83: PLIB_INT_SourceFlagClear(INT_ID_0, INT_SOURCE_TIMER_4);
9D004DF0 00002021 ADDU A0, ZERO, ZERO
9D004DF4 24050010 ADDIU A1, ZERO, 16
9D004DF8 0F40163D JAL 0x9D0058F4
9D004DFC 00000000 NOP
84:           // Test de la periode du timer
85:           BSP_LEDToggle(BSP_LED_4);
9D004E00 24040006 ADDIU A0, ZERO, 6
9D004E04 0F4013C5 JAL BSP_LEDToggle
9D004E08 00000000 NOP
86:
87:           }
9D004E0C 03C0E821 ADDU SP, S8, ZERO
9D004E10 8FA20014 LW V0, 20(SP)
9D004E14 00400013 MTLO V0
9D004E18 8FA30010 LW V1, 16(SP)
9D004E1C 00600011 MTHI V1
9D004E20 8FB00024 LW K1, 36(SP)
9D004E24 27BD0028 ADDIU SP, SP, 40
9D004E28 41DDE800 WRPGPR SP, SP
9D004E2C 409B6000 MTC0 K1, Status
9D004E30 42000018 ERET

```

On constate qu'avec la spécification SRS, il y a beaucoup moins de valeurs sauvegardées (puis restaurées) sur la pile. Le prologue et l'épilogue sont bien plus courts !

### 5.4.7. ACTION POUR FAIRE FONCTIONNER LE TEST

Il suffit de placer les actions suivantes dans APP\_Initialize().

```
void APP_Initialize ( void )
{
    /* Place the App state machine in its initial state. */
    appData.state = APP_STATE_INIT;

    lcd_init();
    lcd_bl_on();
    printf_lcd("Chap5 TestInt      ");
    lcd_gotoxy(1,2);
    printf_lcd("C. Huber 29.11.2016");
    DRV_TMR0_Start();
    DRV_TMR1_Start();
}
```

## 5.5. LES INTERRUPTIONS EXTERNES

Le PIC32MX795F512L ou H dispose de 5 interruptions externes appelées INT0, INT1, INT2, INT3 et INT4.

Une particularité des interruptions externes est la possibilité de configurer la polarité du flanc qui déclenche la demande d'interruption.

### 5.5.1. LISTE DES INTERRUPTIONS EXTERNE

Voici les 5 interruptions externes du PIC32MX795F512L :

Nom complet	No pin	Rôles particuliers
SDO1/OC1/ <b>INT0/RD0</b>	72	SPI 1 SDO
AERXD0/ <b>INT1/RE8</b>	18	
AERXD1/ <b>INT2/RE9</b>	19	
AETXCLK/SCL1/ <b>INT3/RA14</b>	66	I2C 1 SCL
AETXEN/SDA1/ <b>INT4/RA15</b>	67	I2C 1 SDA

#### 5.5.1.1. DÉFINITIONS DES SOURCES

Dans le type énuméré **INT\_SOURCE**, les interruptions externes sont définies ainsi :

```
INT_SOURCE_EXTERNAL_0 = 3,
INT_SOURCE_EXTERNAL_1 = 7,
INT_SOURCE_EXTERNAL_2 = 11,
INT_SOURCE_EXTERNAL_3 = 15,
INT_SOURCE_EXTERNAL_4 = 19,
```

#### 5.5.1.2. DÉFINITIONS DES VECTEURS (PLIB\_INT)

Dans le type énuméré **INT\_VECTOR**, les vecteurs des interruptions externes sont définies ainsi :

```
INT_VECTOR_INT0 = 0x18,
INT_VECTOR_INT1 = 0x38,
INT_VECTOR_INT2 = 0x58,
INT_VECTOR_INT3 = 0x78,
INT_VECTOR_INT4 = 0x98,
```

#### 5.5.1.3. DÉFINITIONS DES VECTEURS (MACRO \_\_ISR)

Dans le fichier p32mx795f512l.h on trouve les définitions des vecteurs à utiliser avec la macro **\_\_ISR**.

#define _EXTERNAL_0_VECTOR	3
#define _EXTERNAL_1_VECTOR	7
#define _EXTERNAL_2_VECTOR	11
#define _EXTERNAL_3_VECTOR	15
#define _EXTERNAL_4_VECTOR	19

## 5.5.2. CONFIGURATION DE LA POLARITÉ DU FLANC

La plib\_int met à disposition deux fonctions pour le choix du flanc montant (rising) ou du flanc descendant (falling).

```
void PLIB_INT_ExternalFallingEdgeSelect(INT_MODULE_ID index, INT_EXTERNAL_SOURCES source);  
void PLIB_INT_ExternalRisingEdgeSelect(INT_MODULE_ID index, INT_EXTERNAL_SOURCES source);
```

∅ Il n'est pas possible de configurer pour obtenir une interruption à tous les flancs (montants et descendants).

### 5.5.2.1. TYPE ÉNUMÉRÉ INT\_EXTERNAL\_SOURCES

Voici le type énuméré INT\_EXTERNAL\_SOURCES dont les valeurs servent à établir la valeur du bit correspondant dans le registre INTCON :

```
typedef enum {  
    INT_EXTERNAL_INT_SOURCE0 = 0x01,  
    INT_EXTERNAL_INT_SOURCE1 = 0x02,  
    INT_EXTERNAL_INT_SOURCE2 = 0x04,  
    INT_EXTERNAL_INT_SOURCE3 = 0x08,  
    INT_EXTERNAL_INT_SOURCE4 = 0x10  
} INT_EXTERNAL_SOURCES;
```

### 5.5.2.2. PLIB\_INT\_EXTERNALRISINGEDGESELECT, EXEMPLE

Dans cet exemple repris de la documentation Harmony (paragraphe "PLIB\_INT\_ExternalRisingEdgeSelect Function"), on voit la possibilité d'établir une liste des interruptions externes qui doivent réagir au flanc montant en combinant avec un OU bit à bit ()).

#### Example

```
PLIB_INT_ExternalRisingEdgeSelect( INT_ID_0,  
                                  ( INT_EXTERNAL_INT_SOURCE0 |  
                                  INT_EXTERNAL_INT_SOURCE1 ) );
```

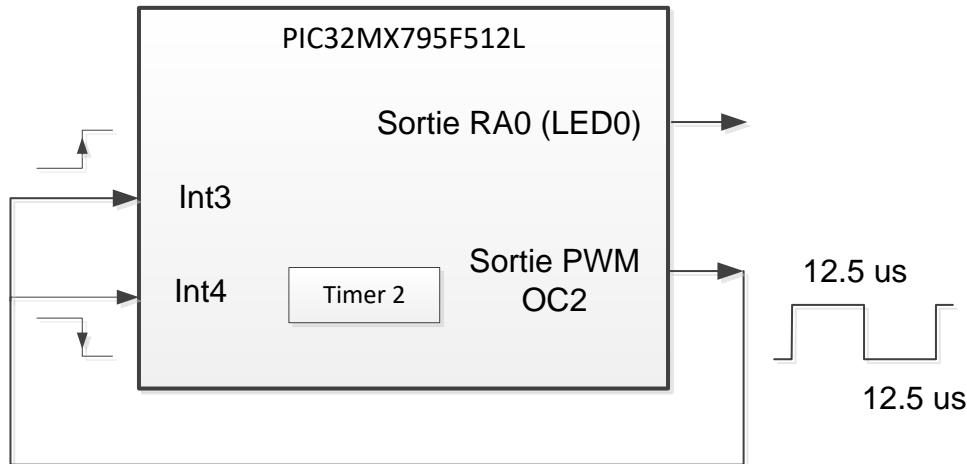
#### Parameters

Parameters	Description
index	Identifier for the module instance to be configured (it should be INT_ID_0 for all of the devices that have only one Interrupt module).
source	One or more of the values from INT_EXTERNAL_SOURCES. Values can be combined using a bitwise "OR" operation.

## 5.6. TEMPS DE RÉACTION D'UNE INTERRUPTION EXTERNE

Nous allons compléter l'exemple précédent afin de mettre en œuvre 2 interruptions externes, pour disposer d'un exemple de configuration et d'un timing à l'oscilloscope.

### 5.6.1. PRINCIPE DE L'EXEMPLE



Utilisation de la sortie OC2 pour fournir un signal. Int3 (priorité 7) doit réagir au flanc montant, int4 (priorité 6) au flanc descendant. Dans la réponse à l'int3 LED0 est mis à 1, alors que dans la réponse à int4 LED0 est mis à 0.

#### 5.6.1.1. CONTRAINE KIT POUR LA RÉALISATION

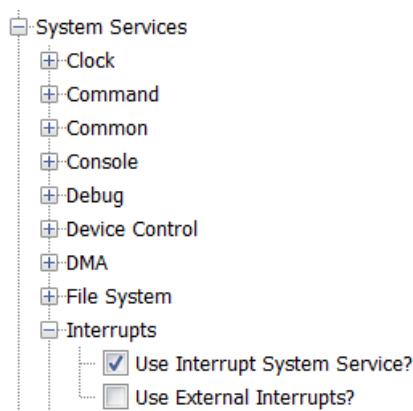


Il est nécessaire de configurer les pins d'interruption externes RA14 et RA15 en entrées pour ne pas avoir de problème avec l'introduction du signal. Il faut câbler la broche 76 (OC2) à 66 et 67.

### 5.6.2. CONFIGURATION DES INT. EXT. AVEC LE MHC

Voici comment mettre en œuvre les interruptions externes et les configurer avec le Microchip Harmony Configurator.

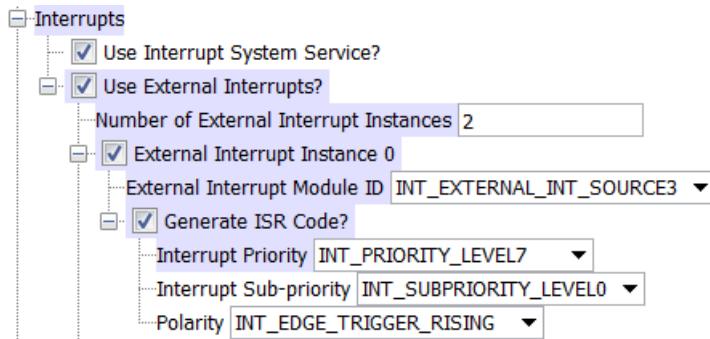
¶ Il s'agit d'utiliser un **System Services** et non pas un driver !



Si on coche Use External Interrupts, il est possible de configurer nos 2 interruptions externes.

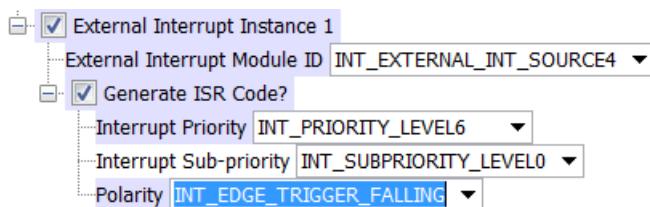
### 5.6.2.1. CONFIGURATION INT3 (INSTANCE 0)

Priorité 7 et flanc montant :



### 5.6.2.2. CONFIGURATION INT4 (INSTANCE 1)

Priorité 6 et flanc descendant :



### 5.6.2.3. MODIFICATION PRIORITÉ INTERRUPTION TIMER 4

Pour éviter d'influencer sur les interruptions externes, il faut réduire la priorité du timer 4 au niveau 5.

### 5.6.3. CODE GÉNÉRÉ POUR LES INTERRUPTIONS EXT. PAR LE MHC

La configuration des interruptions externes est directement effectuée dans le fichier system\_init.c. Utilisation des macros SYS\_INT\_xxxx.

Le code est placé après :

```
/*** Interrupt Service Initialization Code ***/
SYS_INT_Initialize();
```

Et avant :

```
/* Initialize Middleware */

/* Enable Global Interrupts */
SYS_INT_Enable();
```

### 5.6.3.1. CONFIGURATION INT3

Voici le code généré :

```
/*Setup the INT_SOURCE_EXTERNAL_3 and Enable it*/
SYS_INT_VectorPrioritySet(INT_VECTOR_INT3,
                           INT_PRIORITY_LEVEL7);
SYS_INT_VectorSubprioritySet(INT_VECTOR_INT3,
                             INT_SUBPRIORITY_LEVEL0);
SYS_INT_ExternalInterruptTriggerSet
    (INT_EXTERNAL_INT_SOURCE3, INT_EDGE_TRIGGER_RISING);
SYS_INT_SourceEnable(INT_SOURCE_EXTERNAL_3);
```

### 5.6.3.2. CONFIGURATION INT4

Voici le code généré :

```
/*Setup the INT_SOURCE_EXTERNAL_4 and Enable it*/
SYS_INT_VectorPrioritySet(INT_VECTOR_INT4,
                           INT_PRIORITY_LEVEL6);
SYS_INT_VectorSubprioritySet(INT_VECTOR_INT4,
                             INT_SUBPRIORITY_LEVEL0);
SYS_INT_ExternalInterruptTriggerSet
    (INT_EXTERNAL_INT_SOURCE4, INT_EDGE_TRIGGER_FALLING);
SYS_INT_SourceEnable(INT_SOURCE_EXTERNAL_4);
```

En plus des 3 actions de base valables pour n'importe quelle interruption, on trouve les fonctions de sélection du flanc.

## 5.6.4. LES FONCTIONS SYS\_INT

Ce sont des macros définies dans le fichier sys\_int\_mapping.h !

### 5.6.4.1. LA MACRO SYS\_INT\_VECTORPRIORITYSET

La macro SYS\_INT\_VectorPrioritySet utilise **PLIB\_INT\_VectorPrioritySet**.

```
#define SYS_INT_VectorPrioritySet( vector, priority ) \
    PLIB_INT_VectorPrioritySet( INT_ID_0, vector, priority)
```

### 5.6.4.2. LA MACRO SYS\_INT\_VECTORSUBPRIORITYSET

La macro SYS\_INT\_VectorSubprioritySet utilise **PLIB\_INT\_VectorSubPrioritySet**.

```
#define SYS_INT_VectorSubprioritySet( vector, subpriority ) \
    PLIB_INT_VectorSubPrioritySet( INT_ID_0, vector, subpriority)
```

### 5.6.4.3. LA MACRO SYS\_INT\_SOURCEENABLE

La macro SYS\_INT\_SourceEnable utilise **PLIB\_INT\_SourceEnable**.

```
#define SYS_INT_SourceEnable( source ) \
    PLIB_INT_SourceEnable( INT_ID_0, source )
```

#### 5.6.4.4. LA MACRO SYS\_INT\_EXTERNALINTERRUPTTRIGGERSET

N'est pas une macro, c'est une fonction implémentée dans sys\_int\_pic32.c.

```
void SYS_INT_ExternalInterruptTriggerSet
        ( INT_EXTERNAL_SOURCES source,
          INT_EXTERNAL_EDGE_TRIGGER edgeTrigger )
{
    if ( edgeTrigger == INT_EDGE_TRIGGER_RISING )
    {
        PLIB_INT_ExternalRisingEdgeSelect ( INT_ID_0,
                                             source );
    } else {
        PLIB_INT_ExternalFallingEdgeSelect ( INT_ID_0,
                                              source );
    }
}
```

#### 5.6.5. RÉPONSES AUX INTERRUPTIONS EXTERNES

Les réponses aux interruptions externes se trouvent dans le fichier system\_interrupt.c.

##### 5.6.5.1. RÉPONSE INTERRUPTION INT3

La priorité de la routine d'interruption est mise à **IPL7SRS**. L'action sur la Led0 a été placée avant la mise à 0 du Flag d'interruption.

```
void __ISR( _EXTERNAL_3_VECTOR, IPL7SRS )
        _IntHandlerExternalInterruptInstance0(void)
{
    LED0_W = 1;
    PLIB_INT_SourceFlagClear(INT_ID_0,
                               INT_SOURCE_EXTERNAL_3);
}
```

##### 5.6.5.2. RÉPONSE INTERRUPTION INT4

Voici la routine de réponse à l'interruption externe 4 tel que généré avec uniquement l'ajout de l'action sur la Led0.

```
void __ISR( _EXTERNAL_4_VECTOR, IPL6AUTO )
        _IntHandlerExternalInterruptInstance1(void)
{
    LED0_W = 0;
    PLIB_INT_SourceFlagClear(INT_ID_0,
                               INT_SOURCE_EXTERNAL_4);
}
```

### 5.6.6. ACTION POUR FAIRE FONCTIONNER LE TEST

Il suffit de placer les actions suivantes dans APP\_Initialize :

```
void APP_Initialize ( void )
{
    /* Place the App state machine in its initial state. */
    appData.state = APP_STATE_INIT;

    lcd_init();
    lcd_b1_on();
    printf_lcd("Chap5 TestInt      ");
    lcd_gotoxy(1,2);
    printf_lcd("C. Huber 29.11.2016");
    DRV_TMR0_Start();
    DRV_TMR1_Start();
    DRV_TMR2_Start();
    DRV_OC0_Start();
}
```

#### 5.6.6.1. MODIFICATION POUR LE TIMER2

Comme le Timer 2 sert de base de temps, il n'a pas besoin de générer une interruption.

Pour cela il faut mettre en commentaire l'autorisation de l'interruption dans la fonction \_DRV\_TMR2\_Resume :

```
static void _DRV_TMR2_Resume(bool resume)
{
    if (resume)
    {
        PLIB_INT_SourceFlagClear(INT_ID_0,
                                  INT_SOURCE_TIMER_2);
        // PLIB_INT_SourceEnable(INT_ID_0,
        //                        INT_SOURCE_TIMER_2);
        PLIB_TMR_Start(TMR_ID_2);
    }
}
```

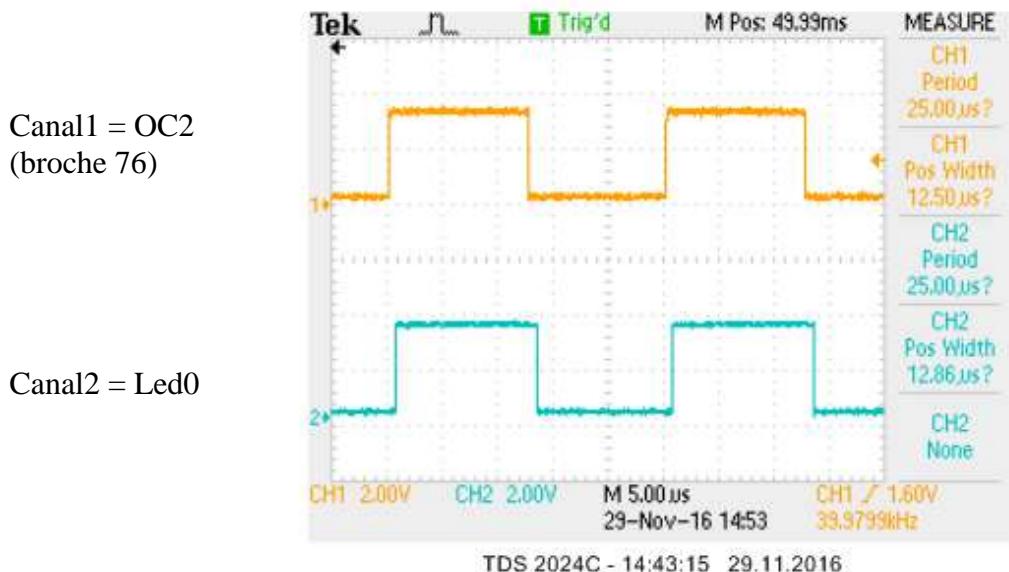
### 5.6.7. MESURE DES TEMPS DE RÉACTION

Il faut câbler la broche 76 (OC2) à 66 (int3) et 67 (int4).

Au niveau du Pin Settings, on constate que le système a ajouté Int3 et Int4 sans configuration de la direction.

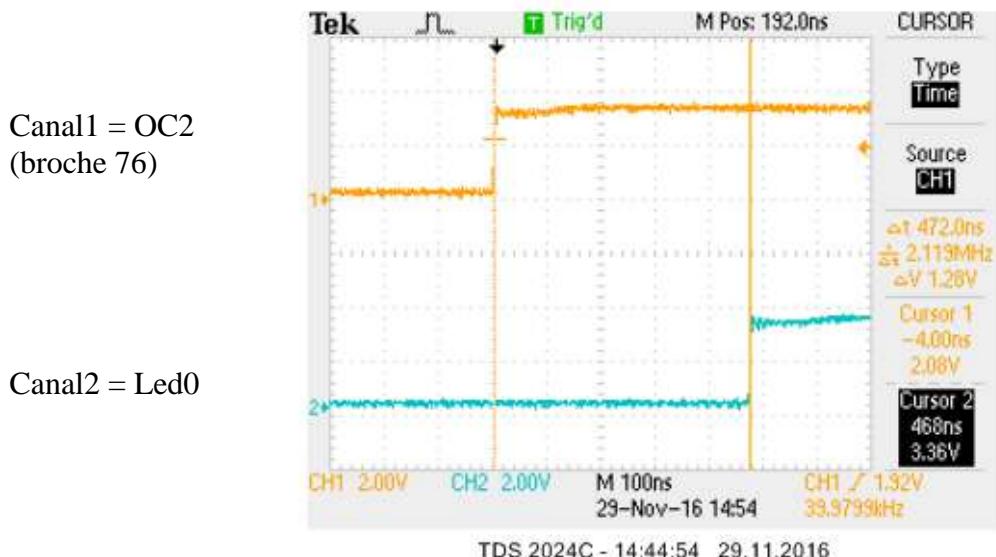
66	RA14	5V	INT3	n/a	n/a	<input type="checkbox"/>	Digital
67	RA15	5V	INT4	n/a	n/a	<input type="checkbox"/>	Digital

#### 5.6.7.1. VUE D'ENSEMBLE



La vue d'ensemble permet de vérifier que le système fonctionne et que la période du signal est bien de 25 us.

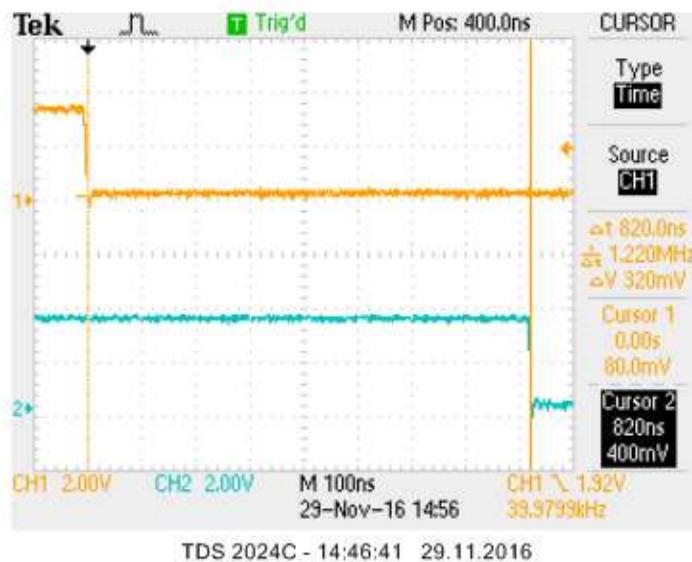
#### 5.6.7.2. TEMPS RÉACTION INT3



On obtient un décalage de 472 ns entre le flanc montant du signal d'interruption (canal 1) et le flanc montant du signal sur la led (canal 2).

### 5.6.7.3. TEMPS RÉACTION INT4

Canal1 = OC2  
(broche 76)  
Canal2 = Led0



On obtient un décalage de 820 ns entre le flanc montant du signal d'interruption (canal 1) et le flanc montant du signal sur la led (canal 2).

## 5.6.8. DIFFÉRENCE DANS LES ROUTINES DE RÉPONSE

Pour expliquer la modeste différence, il faut observer le code des routines de réponse en assembleur.

### 5.6.8.1. ISR INT3 EN ASSEMBLEUR

Voici la réponse à l'interruption externe 3 en assembleur :

```

72:           void __ISR(_EXTERNAL_3_VECTOR, IPL7SRS)
              _IntHandlerExternalInterruptInstance0(void)
73:           {
9D005740  415DE800  RDPGPR SP, SP
9D005744  401A7000  MFC0 K0, EPC
9D005748  401B6000  MFC0 K1, Status
9D00574C  27BDFFD8  ADDIU SP, SP, -40
9D005750  AFBB0024  SW K1, 36(SP)
9D005754  7C1B7844  INS K1, ZERO, 1, 15
9D005758  377B1C00  ORI K1, K1, 7168
9D00575C  409B6000  MTC0 K1, Status
9D005760  00001012  MFLO V0
9D005764  AFA20014  SW V0, 20(SP)
9D005768  00001810  MFHI V1
9D00576C  AFA30010  SW V1, 16(SP)
9D005770  03A0F021  ADDU S8, SP, ZERO
74:           LED0_W = 1;
9D005774  3C03BF88  LUI V1, -16504
9D005778  94626020  LHU V0, 24608(V1)
9D00577C  24040001  ADDIU A0, ZERO, 1
9D005780  7C820004  INS V0, A0, 0, 1
9D005784  A4626020  SH V0, 24608(V1)
75:           PLIB_INT_SourceFlagClear(INT_ID_0,
                           INT_SOURCE_EXTERNAL_3);
9D005788  00002021  ADDU A0, ZERO, ZERO
9D00578C  2405000F  ADDIU A1, ZERO, 15
9D005790  0F40192A  JAL 0x9D0064A8
9D005794  00000000  NOP
76:           }
9D005798  03C0E821  ADDU SP, S8, ZERO
9D00579C  8FA20014  LW V0, 20(SP)
9D0057A0  00400013  MTLO V0
9D0057A4  8FA30010  LW V1, 16(SP)
9D0057A8  00600011  MTHI V1
9D0057AC  8FBB0024  LW K1, 36(SP)
9D0057B0  27BD0028  ADDIU SP, SP, 40
9D0057B4  41DDE800  WRPGPR SP, SP
9D0057B8  409B6000  MTC0 K1, Status
9D0057BC  42000018  ERET
```

### 5.6.8.2. ISR INT4 EN ASSEMBLEUR

Voici la réponse à l'interruption externe 3 en assembleur :

```

78:           void __ISR(_EXTERNAL_4_VECTOR, IPL6AUTO)
                _IntHandlerExternalInterruptInstance1(void)
79:           {
9D003800  415DE800  RDPGPR SP, SP
9D003804  401B7000  MFC0 K1, EPC
9D003808  401A6002  MFC0 K0, SRSCtl
9D00380C  27BDFF88  ADDIU SP, SP, -120
9D003810  AFBB0074  SW K1, 116(SP)
9D003814  401B6000  MFC0 K1, Status
9D003818  AFBA006C  SW K0, 108(SP)
9D00381C  AFBB0070  SW K1, 112(SP)
9D003820  7C1B7844  INS K1, ZERO, 1, 15
9D003824  377B1800  ORI K1, K1, 6144
9D003828  409B6000  MTC0 K1, Status
9D00382C  AFA3001C  SW V1, 28(SP)
9D003830  AFA20018  SW V0, 24(SP)
9D003834  8FA3006C  LW V1, 108(SP)
9D003838  3063000F  ANDI V1, V1, 15
9D00383C  14600012  BNE V1, ZERO, 0x9D003888
9D003840  00000000  NOP
9D003844  AFBF005C  SW RA, 92(SP)
9D003848  AFBE0058  SW S8, 88(SP)
9D00384C  AFB90054  SW T9, 84(SP)
9D003850  AFB80050  SW T8, 80(SP)
9D003854  AFAF004C  SW T7, 76(SP)
9D003858  AFAE0048  SW T6, 72(SP)
9D00385C  AFAD0044  SW T5, 68(SP)
9D003860  AFAC0040  SW T4, 64(SP)
9D003864  AFAB003C  SW T3, 60(SP)
9D003868  AFAA0038  SW T2, 56(SP)
9D00386C  AFA90034  SW T1, 52(SP)
9D003870  AFA80030  SW T0, 48(SP)
9D003874  AFA7002C  SW A3, 44(SP)
9D003878  AFA60028  SW A2, 40(SP)
9D00387C  AFA50024  SW A1, 36(SP)
9D003880  AFA40020  SW A0, 32(SP)
9D003884  AFA10014  SW AT, 20(SP)
9D003888  00000000  NOP
9D00388C  00001012  MFLO V0
9D003890  AFA20064  SW V0, 100(SP)
9D003894  00001810  MFHI V1
9D003898  AFA30060  SW V1, 96(SP)
9D00389C  03A0F021  ADDU S8, SP, ZERO
80:           LED0_W = 0;
9D0038A0  3C03BF88  LUI V1, -16504
9D0038A4  94626020  LHU V0, 24608(V1)
9D0038A8  7C020004  INS V0, ZERO, 0, 1
9D0038AC  A4626020  SH V0, 24608(V1)
81:           PLIB_INT_SourceFlagClear(INT_ID_0,
                           INT_SOURCE_EXTERNAL_4);
9D0038B0  00002021  ADDU A0, ZERO, ZERO

```

```

9D0038B4 24050013 ADDIU A1, ZERO, 19
9D0038B8 0F40192A JAL 0x9D0064A8
9D0038BC 00000000 NOP
82:
9D0038C0 03C0E821 ADDU SP, S8, ZERO
9D0038C4 8FA20064 LW V0, 100(SP)
9D0038C8 00400013 MTLO V0
9D0038CC 8FA30060 LW V1, 96(SP)
9D0038D0 00600011 MTHI V1
9D0038D4 8FA2006C LW V0, 108(SP)
9D0038D8 3042000F ANDI V0, V0, 15
9D0038DC 14400014 BNE V0, ZERO, 0x9D003930
9D0038E0 00000000 NOP
9D0038E4 8FBF005C LW RA, 92(SP)
9D0038E8 8FBE0058 LW S8, 88(SP)
9D0038EC 8FB90054 LW T9, 84(SP)
9D0038F0 8FB80050 LW T8, 80(SP)
9D0038F4 8FAF004C LW T7, 76(SP)
9D0038F8 8FAE0048 LW T6, 72(SP)
9D0038FC 8FAD0044 LW T5, 68(SP)
9D003900 8FAC0040 LW T4, 64(SP)
9D003904 8FAB003C LW T3, 60(SP)
9D003908 8FAA0038 LW T2, 56(SP)
9D00390C 8FA90034 LW T1, 52(SP)
9D003910 8FA80030 LW T0, 48(SP)
9D003914 8FA7002C LW A3, 44(SP)
9D003918 8FA60028 LW A2, 40(SP)
9D00391C 8FA50024 LW A1, 36(SP)
9D003920 8FA40020 LW A0, 32(SP)
9D003924 8FA3001C LW V1, 28(SP)
9D003928 8FA20018 LW V0, 24(SP)
9D00392C 8FA10014 LW AT, 20(SP)
9D003930 00000000 NOP
9D003934 41606000 DI ZERO
9D003938 000000C0 EHB
9D00393C 8FBA0074 LW K0, 116(SP)
9D003940 8FBB0070 LW K1, 112(SP)
9D003944 409A7000 MTC0 K0, EPC
9D003948 8FBA006C LW K0, 108(SP)
9D00394C 27BD0078 ADDIU SP, SP, 120
9D003950 409A6002 MTC0 K0, SRSCtl
9D003954 41DDE800 WRPGPR SP, SP
9D003958 409B6000 MTC0 K1, Status
9D00395C 42000018 ERET

```

On remarque que la réponse Int3 avec le SRS effectue un minimum de sauvegarde, tandis que pour l'int4 en mode AUTO il y a des sauvegardes assez complètes.

### 5.6.9. CONCLUSION SUR LES INTERRUPTIONS EXTERNES

On constate que le temps de réaction entre le flanc qui déclenche l'interruption et le basculement de la sortie est inférieur à 500 ns dans le cas des routines de réponse utilisant le SRS et de l'ordre de 800 ns en mode AUTO.

#### 5.6.9.1. FRÉQUENCE LIMITE

En supposant que le temps pourachever la réponse à l'interruption est aussi de l'ordre de 500 ns, on peut en conclure que si l'on applique un signal de 1 MHz le système sera entièrement occupé à entrer et sortir de l'interruption. Si on veut obtenir un traitement pour d'autres éléments, il est ainsi raisonnable de ne pas dépasser les 200 kHz.

#### 5.6.9.2. SITUATION FAVORABLE

Les deux routines de réponse aux interruptions externes représentent un cas favorable, il est difficile d'avoir moins d'actions.

A partir du moment où on appelle une fonction dans la routine, les temps de traitement vont s'allonger à cause de sauvegarde plus nombreuses sur la pile.

## 5.7. CONCLUSION

Ce chapitre offre un aperçu du mécanisme de traitement des interruptions du PIC32MX. Seul le mode multi-vecteur a été présenté au niveau du code assembleur.

La présentation succincte des fonctions de la plib\_int et les quelques exemples devraient apporter aux étudiants la capacité à configurer et réaliser une routine de réponse à une interruption externe ou autre.

L'exemple pratique avec la mesure du temps de réaction permet d'avoir une idée des performances du PIC32MX dans ce domaine.

## 5.8. HISTORIQUE DES VERSIONS

### 5.8.1. VERSION 1.0 FÉVRIER 2014

Création du document et découverte du mécanisme des interruptions du PIC32MX.

### 5.8.2. VERSION 1.5 DÉCEMBRE 2014

Complément avec des aspects notions générales. Adaptation à la PLIB\_INT de Harmony. Cas des interruptions externes et exemple pratique.

### 5.8.3. VERSION 1.7 DÉCEMBRE 2015

Adaptation à la PLIB\_INT de Harmony 1.06 et a l'évolution du MHC en particulier pour la configuration des interruptions externes.

### 5.8.4. VERSION 1.8 NOVEMBRE 2016

Maj info documentation. Adaptation à la PLIB\_INT de Harmony 1.08 et à l'évolution du MHC. Eléments pratiques sur la base de la même application qui évolue pour la gestion des interruptions externes.

### 5.8.5. VERSION 1.9 NOVEMBRE 2017

Reprise et relecture par SCA.

**MINF**  
**Programmation des PIC32MX**

# **Chapitre 6**

## **Timers, PWM & capture**



### **Théorie PIC32MX**

**Christian HUBER (CHR)**  
**Serge CASTOLDI (SCA)**  
**Version 1.92 décembre 2019**



## CONTENU DU CHAPITRE 6

<b>6. Timers, PWM et capture</b>	<b>6-1</b>
<b>6.1. Le core timer du PIC32MX</b>	<b>6-2</b>
6.1.1. Principe du core timer	6-2
6.1.2. Utilisation via macros du compilateur	6-3
6.1.3. Utilisation via librairie Mc32CoreTimer	6-4
6.1.3.1. Contenu du fichier Mc32CoreTimer.h	6-5
6.1.3.2. Réalisation de la fonction OpenCoreTimer	6-5
6.1.3.3. Réalisation de la fonction ReadCoreTimer	6-5
6.1.3.4. Réalisation de la fonction UpdateCoreTimer	6-6
6.1.3.5. Réalisation de la fonction WriteCoreTimer	6-6
6.1.4. Interruption périodique avec le core timer	6-6
6.1.4.1. Configuration Interruption du core timer	6-6
6.1.4.2. Interruption du core timer	6-7
6.1.5. Réalisation de délais avec le core timer	6-7
<b>6.2. Les timers (peripheral) du PIC32MX</b>	<b>6-9</b>
6.2.1. Documentation et librairie à disposition	6-9
6.2.2. Principe fonctionnement des timers	6-9
6.2.2.1. Illustration du mécanisme de période match	6-10
6.2.3. Le timer 1	6-10
6.2.4. Les timers 2, 3, 4 et 5	6-11
6.2.5. Principe de configuration d'un timer	6-11
6.2.5.1. Le type énuméré TMR_MODULE_ID	6-12
6.2.5.2. Sélection du "Clock source"	6-12
6.2.5.3. Sélection du "Prescaler"	6-12
6.2.5.4. Sélection du mode	6-13
6.2.5.5. Mise à zéro d'un timer	6-13
6.2.5.6. Etablissement de la période d'un timer	6-13
6.2.5.7. Lecture de la période d'un timer	6-14
6.2.5.8. La fonction PLIB_TMR_Stop	6-14
6.2.5.9. La fonction PLIB_TMR_Start	6-14
6.2.6. Exemple de configuration du timer 1	6-14
6.2.6.1. Configuration timer 1 au niveau MHC	6-14
6.2.6.2. Fonction de configuration du timer 1 obtenue	6-15
6.2.6.3. Lancement du timer 1	6-15
6.2.7. Exemple configuration du timer 2	6-16
6.2.7.1. Configuration timer 2 au niveau MHC	6-16
6.2.7.2. Fonction de configuration du timer 2 obtenue	6-16
6.2.7.3. Lancement du timer 2	6-17
6.2.8. Exemple configuration du timer 3	6-17
6.2.8.1. Configuration timer 3 au niveau MHC	6-18
6.2.8.2. Fonction de configuration du timer 3 obtenue	6-18
6.2.8.3. Lancement du timer 3	6-19
6.2.9. Les timers 32 bits	6-20
6.2.10. Timer 32 bits, exemple	6-20
6.2.10.1. Configuration au niveau MHC de la paire timers 4 & 5	6-21
6.2.10.2. Fonction de configuration Obtenue pour la paire 4 & 5	6-21

6.2.10.3. Lancement des timers 4 & 5	6-22
6.2.10.4. Réponse interruption pour la paire timers 4 & 5	6-22
6.2.11. Les timers en comptage externe	6-23
6.2.11.1. Liste des entrées de comptage TxCK	6-23
6.2.11.2. Sélection source externe, exemple	6-23
6.2.11.3. La fonction PLIB_TMR_ClockSourceExternalSyncEnable	6-23
6.2.11.4. La fonction PLIB_TMR_ClockSourceExternalSyncDisable	6-23
6.2.11.5. Fonctions d'accès au compteur du timer	6-24
6.2.11.6. La fonction PLIB_TMR_Counter16BitGet	6-24
6.2.11.7. La fonction DRV_TMRx_CounterValueGet	6-24
<b>6.3. Les Modules "Output Compare"</b>	<b>6-25</b>
6.3.1. Schéma bloc du module "Output Compare"	6-25
6.3.2. Liste des sorties de comparaisons	6-25
6.3.1. Principe fonctionnement des OC	6-26
6.3.2. Fonctions de la PLIB_OC	6-26
6.3.3. Actions possibles	6-27
6.3.4. Fonctions pour configurer les modules OC	6-27
6.3.4.1. Le type énuméré OC_MODULE_ID	6-27
6.3.4.2. La fonction PLIB_OC_ModeSelect	6-27
6.3.4.3. La fonction PLIB_OC_BufferSizeSelect	6-29
6.3.4.4. La fonction PLIB_OC_TimerSelect	6-30
6.3.4.5. La fonction PLIB_OC_FaultInputSelect	6-30
6.3.4.6. La fonction PLIB_OC_Buffer16BitSet	6-30
6.3.4.7. La fonction PLIB_OC_Buffer32BitSet	6-30
6.3.4.8. La fonction PLIB_OC_PulseWidth16BitSet	6-30
6.3.4.9. La fonction PLIB_OC_PulseWidth32BitSet	6-31
6.3.5. Exemple génération d'un signal PWM	6-31
6.3.5.1. Configuration du timer 2	6-31
6.3.5.2. Configuration du OC2	6-31
6.3.5.3. Activation de l'OC2	6-32
6.3.5.4. Modulation du signal PWM	6-33
6.3.6. Exemple génération d'une impulsion	6-33
6.3.6.1. Configuration du timer 3	6-33
6.3.6.2. Configuration de OC3	6-34
6.3.6.3. Modulation de la largeur d'impulsion	6-35
6.3.6.4. Décalage du flanc montant de l'impulsion	6-35
6.3.7. Application pour contrôle des résultats	6-36
6.3.7.1. Observation des résultats	6-36
6.3.8. Application pour contrôle des résultats suite	6-37
6.3.8.1. Observation des résultats	6-38
<b>6.4. Les Inputs Capture du PIC32MX</b>	<b>6-39</b>
6.4.1. Evénements de capture	6-39
6.4.1.1. Evénements simples	6-39
6.4.1.2. Evénements doubles	6-39
6.4.1.3. Evénements multiples	6-39
6.4.2. Liste des entrées de captures	6-39
6.4.3. Schéma de principe mécanisme de capture	6-40
6.4.4. Principe de configuration de la capture	6-40
6.4.4.1. Séquence de lancement de la capture	6-41
6.4.5. Fonctions de configuration de la capture	6-41
6.4.5.1. Le type énuméré IC_MODULE_ID	6-41
6.4.5.2. La fonction PLIB_IC_ModeSelect	6-41

6.4.5.3.	La fonction PLIB_IC_FirstCaptureEdgeSelect	6-42
6.4.5.4.	La fonction PLIB_IC_TimerSelect	6-42
6.4.5.5.	La fonction PLIB_IC_BufferSizeSelect	6-43
6.4.5.6.	La fonction PLIB_IC_EventsPerInterruptSelect	6-43
6.4.5.7.	La fonction PLIB_IC_Disable	6-43
6.4.5.8.	La fonction PLIB_IC_Enable	6-43
6.4.6.	Exemple de configuration de la capture	6-44
6.4.7.	Fonctions d'utilisation de la capture	6-45
6.4.7.1.	Lecture résultat capture : PLIB_IC_Buffer16BitGet	6-45
6.4.7.2.	Lecture résultat capture : PLIB_IC_Buffer32BitGet	6-45
6.4.7.3.	La fonction PLIB_IC_BufferIsEmpty	6-45
6.4.7.4.	La fonction PLIB_IC_BufferOverflowHasOccurred	6-45
6.4.8.	Fonctions fournies par le DRV_ICx	6-46
6.4.8.1.	DRV_IC0_Start	6-46
6.4.8.2.	DRV_IC0_Stop	6-46
6.4.8.3.	DRV_IC0_Open	6-46
6.4.8.4.	DRV_IC0_Close	6-46
6.4.8.5.	DRV_IC0_Capture32BitDataRead	6-46
6.4.8.6.	DRV_IC0_Capture16BitDataRead	6-46
6.4.8.7.	DRV_IC0_BufferIsEmpty	6-47
6.4.9.	Lancement d'un IC	6-47
6.4.10.	Capture, exemple complet	6-48
6.4.10.1.	Configuration du timer 2 et OC2	6-49
6.4.10.2.	Configuration du timer 3	6-49
6.4.10.3.	Configuration de la capture (MHC)	6-50
6.4.10.4.	Include pour action IC	6-50
6.4.10.5.	Réponse à l'interruption de capture	6-50
6.4.10.6.	Vérification du fonctionnement de l'interruption de capture	6-52
6.4.10.7.	Lecture du tampon de capture	6-53
6.4.10.8.	Gestion du reboulement du timer 3	6-53
6.4.10.9.	Détail calcul période et Thaut	6-54
6.4.10.10.	Include pour action OC au niveau application	6-54
6.4.10.11.	Encclenchement capture au niveau init de l'application	6-54
6.4.10.12.	Détail des datas de l'application	6-55
6.4.10.13.	Contenu de de la section case APP_STATE_SERVICE_TASKS	6-55
<b>6.5.</b>	<b>Conclusion</b>	<b>6-56</b>
<b>6.6.</b>	<b>Historique des versions</b>	<b>6-56</b>
6.6.1.	V1.0 mars 2014	6-56
6.6.2.	V1.1 mars 2014	6-56
6.6.3.	V1.5 janvier 2015	6-56
6.6.4.	V1.6 janvier 2015	6-56
6.6.5.	V1.7 janvier 2016	6-56
6.6.6.	V1.8 janvier 2017	6-56
6.6.7.	V1.8.1 janvier 2017	6-56
6.6.8.	V1.9 novembre 2017	6-56
6.6.9.	V1.91 janvier 2019	6-56
6.6.10.	V1.92 décembre 2019	6-56

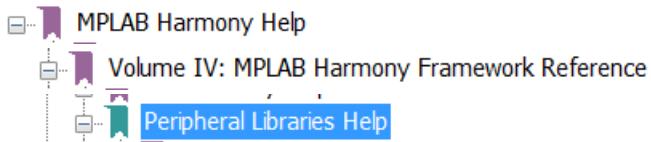


## 6. TIMERS, PWM ET CAPTURE

Dans ce chapitre nous allons étudier les timers du PIC32MX ainsi que leur combinaison avec les entrées de capture et les sorties de comparaison pour obtenir des signaux PWM.

Les documents de référence sont :

- La documentation "PIC32 Family Reference Manual" :  
Section 14 : Timers
- Dans le même document :
  - Section 16 : Output Compare
  - Section 15 : Input Capture
- Pour les détails spécifiques au PIC32MX795F512L, il faut se référer au document "PIC32MX5XX/6XX/7XX Family Data Sheet" :  
Section 13 pour le timer 1 et section 14 pour les timers 2/3 et 4/5
- Dans le même document :
  - Section 17 : Output Compare
  - Section 16 : Input Capture
- La documentation d'Harmony, qui se trouve dans <Répertoire Harmony>\v<n>\doc :  
Section MPLAB Harmony Framework Reference > Peripheral Libraries Help,  
sous-sections Timer Peripheral Library, Output Compare Peripheral Library et Input  
Capture Peripheral Library



Ce document a été établi sur la base de Harmony v1.08, puis modifié par rapport à Hamony 2.05.

## 6.1. LE CORE TIMER DU PIC32MX

Etant donné certaines fonctionnalités du CPU, comme par exemple le prefetch cache, il est difficile d'obtenir des résultats déterministes en termes de nombre de cycles d'exécution correspondant à une portion de code assembleur ou C. Le core timer et typiquement là pour résoudre le problème en apportant une base de temps de référence.

Le core timer peut par exemple être utilisé :

- lors de la réalisation d'attentes passives (les plus courtes possibles !) ne dépendant pas du temps d'exécution des instructions.
- par les OS temps réels pour générer leur tick interne, les timer périphériques restant alors tous libres pour le programme utilisateur.

¶ Le core timer fait partie de l'architecture MIPS; ce n'est pas un timer périphérique ajouté par Microchip. Ainsi, il n'est pas pris en charge par les bibliothèques de Harmony. Il était supporté par le compilateur XC32 (version  $\leq$  1.34, via le fichier #include <peripheral\timer.h>), mais ce n'est plus le cas au moment de l'écriture de cette documentation (novembre 2017, xc32 v1.43).

Les documents de référence pour le core timer sont :

- La documentation "PIC32 Family Reference Manual" :  
Section 2 : CPU for Devices with M4K Core
- La documentation MIPS, qui parle surtout du core timer en faisant référence au coprocesseur 0. Le core timer est une des fonctionnalités apportée par le coprocesseur 0.

### 6.1.1. PRINCIPE DU CORE TIMER

Le PIC32 utilise un jeu de registres spécial pour communiquer des informations de statut et de contrôle entre le CPU et le logiciel : le coprocesseur 0, abrégé CP0.

On accède aux fonctionnalités du core timer via 2 registres du coprocesseur 0 :

- Registre "Count" (CP0 registre 9) : C'est le registre de comptage.  
Une valeur 32 bits qui est incrémentée à la moitié de la fréquence du CPU (SYSCLK / 2). Le core timer compte toujours (incrémantation du registre count), la seule exception étant une suspension de l'incrémantation en mode debug.
- Registre "Compare" (CP0 registre 11) : C'est le registre de comparaison.  
Lorsque la valeur du core timer (registre count) est égale à la valeur de comparaison, cela permet de générer une interruption.  
Cette interruption possède un bit d'activation traditionnel.  
Le core timer n'est pas remis à 0 lors de la comparaison. Pour une interruption cyclique, il faut ajouter à la valeur de comparaison un offset fixe lors de l'interruption.

### 6.1.2. UTILISATION VIA MACROS DU COMPILATEUR

Le compilateur offre tout de même des macros simples permettant d'accéder de manière basique aux 2 registres du core timer.

Pour les utiliser, il faut inclure le fichier standard du compilateur :

```
#include <xc.h>
```

Ce fichier en inclut lui-même un autre, qui permet d'accéder aux registres du coprocessor 0 :

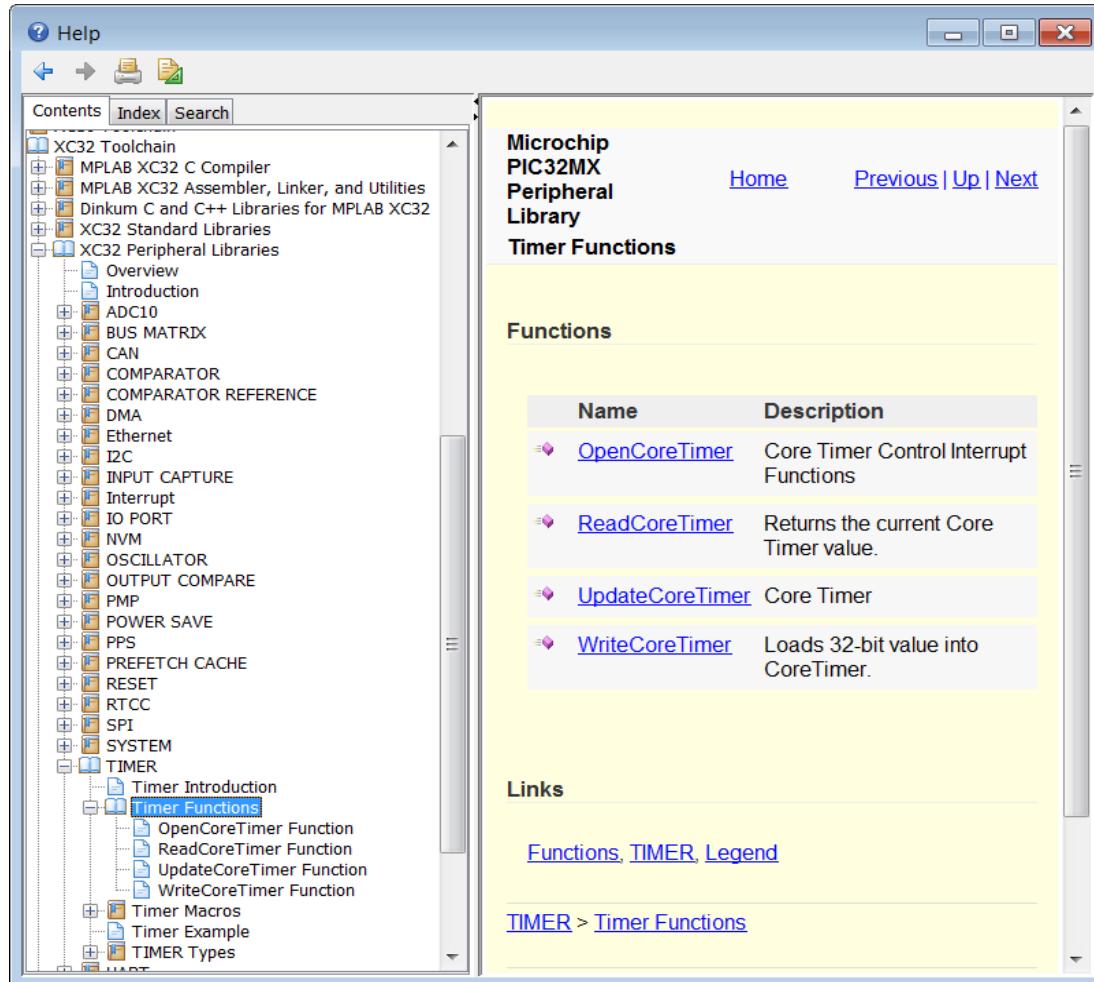
```
#include <cp0defs.h>
```

Les fonctions sont :

- `_CP0_GET_COUNT()` Lecture du registre 9 (count)
- `_CP0_SET_COUNT(val)` Ecriture dans registre 9 (count)
- `_CP0_GET_COMPARE` Lecture du registre 11 (compare)
- `_CP0_SET_COMPARE(val)` Ecriture dans registre 11 (compare)

### 6.1.3. UTILISATION VIA LIBRAIRIE Mc32CORETIMER

L'aide en ligne dans MPLABX Help > Help Contents > XC32 Toolchain > XC32 Peripheral Libraries > TIMER fournit une information sommaire au sujet de fonctions d'utilisation du core timer..



Cela semble être un reliquat de documentation, ces fonctions étant introuvable. Les fonctions ont donc été réécrites et sont fournies avec le BSP dans la librairie Mc32CoreTimer.

### 6.1.3.1. CONTENU DU FICHIER Mc32CORETIMER.H

Voici le contenu du fichier Mc32CoreTimer.h

```
#include <stdint.h>
#include <cp0defs.h>

/*-----
// Définition des fonctions prototypes
-----*/
void OpenCoreTimer( uint32_t compare);
uint32_t ReadCoreTimer(void);
void UpdateCoreTimer( uint32_t period);
void WriteCoreTimer( uint32_t val);
```

☞ Remarque : Dans le fichier Mc32CoreTimer.c il est nécessaire d'inclure :

```
#include <xc.h>
```

### 6.1.3.2. REALISATION DE LA FONCTION OPENCORETIMER

Cette fonction charge la valeur de comparaison et met à 0 le compteur.

```
void OpenCoreTimer( uint32_t compare)
{
    _CP0_SET_COMPARE(compare);
    _CP0_SET_COUNT(0);
}
```

Exemple :

Réaliser une période de 10 ms.

Le CoreTimer utilise SYS\_CLK / 2 comme horloge.  $f_{CT} = f_{SYSCLK}/2 = 80\text{MHz} / 2 = 40\text{ MHz}$ .

Il faut effectuer le OpenCoreTimer avec :

$$N_{MAX,CT} = T_{VOULU} / T_{CT} = 10'000 / (0,0125 * 2) = 400'000$$

```
OpenCoreTimer(400000);
```

### 6.1.3.3. REALISATION DE LA FONCTION READCORETIMER

Fournit la valeur du core timer (CP0 registre COUNT).

```
uint32_t ReadCoreTimer(void)
{
    return ( _CP0_GET_COUNT() );
}
```

#### 6.1.3.4. REALISATION DE LA FONCTION UPDATECORETIMER

Cette fonction ajoute l'argument *period* à la valeur du registre COMPARE du CP0.

```
void UpdateCoreTimer( uint32_t period)
{
    uint32_t NewCompare = _CP0_GET_COMPARE() + period;
    _CP0_SET_COMPARE(NewCompare);
}
```

#### 6.1.3.5. REALISATION DE LA FONCTION WRITECORETIMER

Cette fonction impose une valeur au registre COUNT de CP0.

```
void WriteCoreTimer( uint32_t val)
{
    _CP0_SET_COUNT(val);
}
```

### 6.1.4. INTERRUPTION PÉRIODIQUE AVEC LE CORE TIMER

Pour réaliser une interruption périodique avec le core timer, nous réalisons une fonction qui comporte l'ouverture du core timer et la configuration de l'interruption.

#### 6.1.4.1. CONFIGURATION INTERRUPTION DU CORE TIMER

Voici le contenu de la fonction de configuration du core timer qui a été créée. Son appel a été réalisé dans la fonction *SYS\_Initialize()*.

```
void Init_CoreTimer(void)
{
    OpenCoreTimer(400000);

    PLIB_INT_SourceEnable(INT_ID_0, INT_SOURCE_TIMER_CORE);
    PLIB_INT_VectorPrioritySet(INT_ID_0, INT_VECTOR_CT,
                               INT_PRIORITY_LEVEL5);
    PLIB_INT_VectorSubPrioritySet(INT_ID_0, INT_VECTOR_CT,
                                 INT_SUBPRIORITY_LEVEL0);
}
```

☝ Le core timer est présent dans les types énumérés de la **PLIB\_INT**. Il est nécessaire d'inclure `#include "peripheral/int/plib_int.h"`

#### 6.1.4.2. INTERRUPTION DU CORE TIMER

Voici un exemple montrant la réalisation de la routine de réponse à l'interruption du core timer. La période prévue est de 10 ms.

```
// Réponse à l'interruption du core timer (cycle 10 ms)
void __ISR(_CORE_TIMER_VECTOR, ip15AUTO)
                                CoreTimerHandler(void)
{
    // clear the interrupt flag
    PLIB_INT_SourceFlagClear(INT_ID_0,
                             INT_SOURCE_TIMER_CORE);

    // Update the core timer
    UpdateCoreTimer( 400000 );
    // Inverse la Led 5
    BSP_LEDToggle(BSP_LED_5);
}
```

La particularité de la réponse à l'interruption du core timer est d'appeler la fonction UpdateCoreTimer qui ajouter la période à la valeur de comparaison. Ceci afin de ne pas modifier ou reseter la valeur de comptage du core timer. On obtient ainsi une durée exacte entre les interruptions successives.

#### 6.1.5. RÉALISATION DE DÉLAIS AVEC LE CORE TIMER

Le core timer peut être utilisé pour réaliser des délais qui ne sont pas basés sur le temps d'exécution des instructions. Ceci présente l'avantage de supporter d'être interrompu sans que le délai s'allonge.

Voici la réalisation de la fonction **delay\_msCt**, qui utilise le core timer :

```
#ifndef SYS_FREQ
    #define SYS_FREQ (80000000L)      //80 MHz
#endif

//le core timer est incrémenté tous les 2 SYSCLK
#define TICK_CT_MS (SYS_FREQ / 2000L)
#define TICK_CT_US (SYS_FREQ / 2000000L)
#define TICK_OVERHEAD 15      //pour ajustement.

/*-----
// Fonction delay_msCt core timer
-----*/
//attente passive n * ms
//utilise le core timer
void __attribute__((optimize("-O0")))
                                delay_msCt(uint32_t NbMs)
{
    uint32_t time_to_wait;

    _CP0_SET_COUNT(0);
    time_to_wait = (TICK_CT_MS * NbMs) - TICK_OVERHEAD;
    while( _CP0_GET_COUNT() < time_to_wait) {
        // Waiting
    }
}
```

La réalisation a l'inconvénient de mettre à 0 le core timer. On ne pourra donc pas utiliser simultanément les interruptions du core timer.

Pour ce faire, il faudrait modifier la fonction **delay\_msCt** de manière à ce que le core timer ne soit pas remis à zéro. Cette modification est laissée à la discréption du lecteur.

La fonction ci-dessus est présente dans la **librairie Mc32Delays du BSP**. Cette librairie ayant évolué avec le temps, elle contient également des fonctions de délais sans utilisation du core timer, avec l'imprécision et les désavantages que cela comporte.

Les fonctions utilisant le core timer ont le suffixe -Ct, et sont :

- void delay\_msCt (unsigned int NbMs)
- void delay\_usCt (unsigned int NbUs)
- void delay500nsCt (void)

## 6.2. LES TIMERS (PERIPHERAL) DU PIC32MX

Le PIC32MX795F512L possède 5 timers 16 bits. Il est possible d'utiliser par paire T2 & T3 ou T4 & T5, pour former des timers 32 bits.

### 6.2.1. DOCUMENTATION ET LIBRAIRIE À DISPOSITION

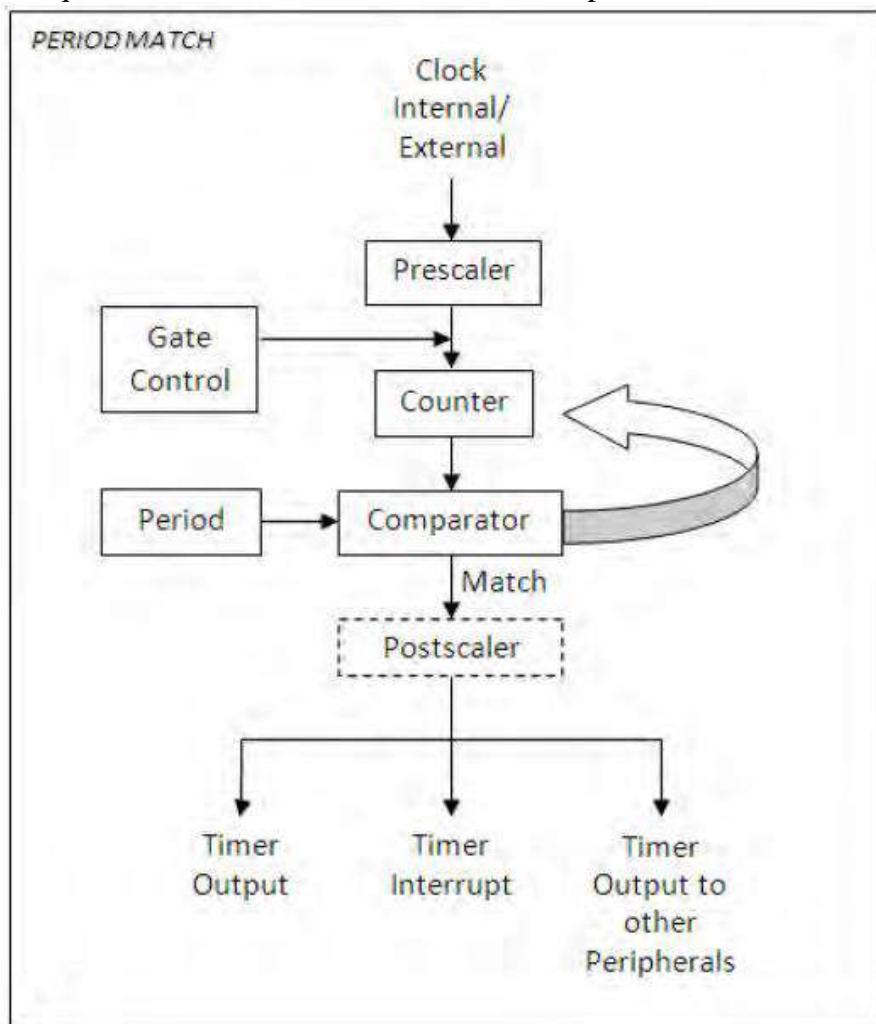
Au niveau d'Harmony, pour gérer les timers périphériques, il est nécessaire d'inclure `plib_tmr.h`

La section **Timer Peripheral Library** de la documentation décrit l'ensemble des fonctions et fournit quelques exemples.

¶ Cette librairie est très générale. Toutes les fonctions ne s'appliquent pas forcément à un modèle de processeur donné. C'est pour cela qu'il existe de nombreuses fonctions permettant de vérifier l'existence de certaines particularités.

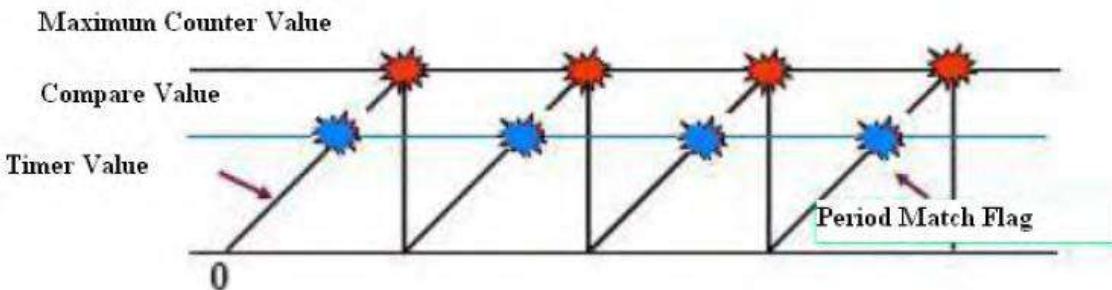
### 6.2.2. PRINCIPE FONCTIONNEMENT DES TIMERS

La documentation décrit les timers comme des "period match timer", le timer étant remis à 0 lorsque la valeur du timer atteint la valeur de période.



### 6.2.2.1. ILLUSTRATION DU MECANISME DE PERIODE MATCH

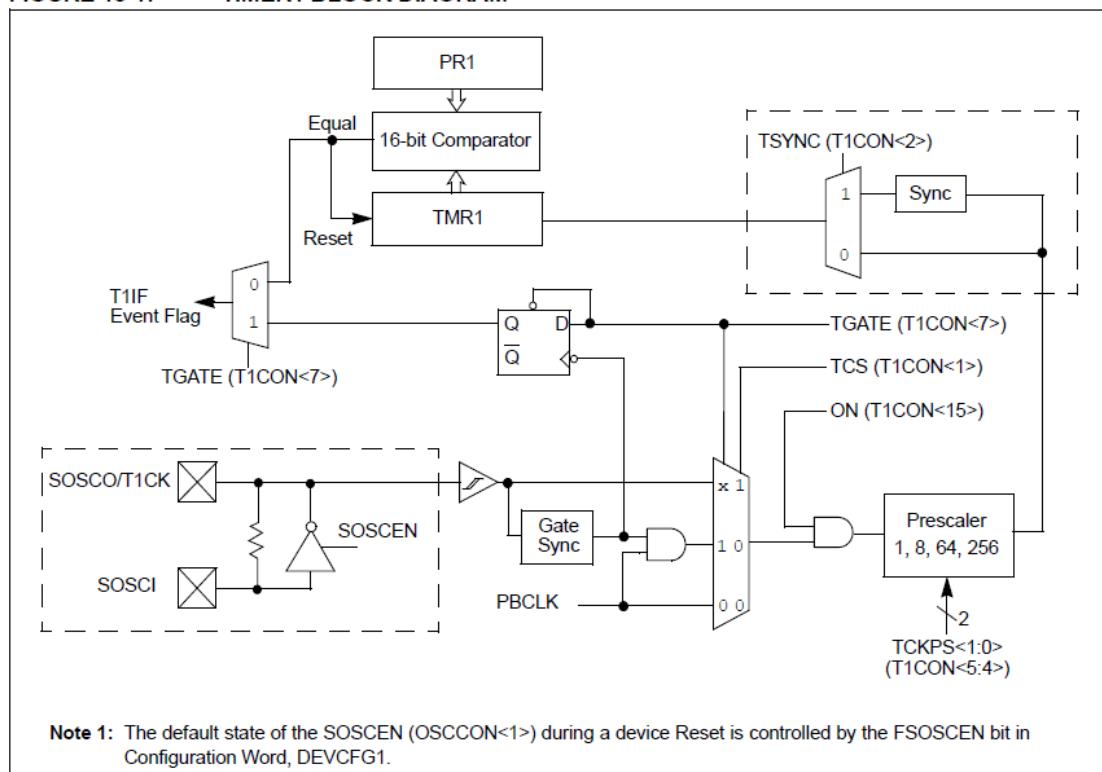
Le diagramme ci-dessous, que l'on trouve dans la documentation Harmony, illustre le principe de fonctionnement des timers :.



### 6.2.3. LE TIMER 1

Voici le schéma bloc du timer 1 :

**FIGURE 13-1: TIMER1 BLOCK DIAGRAM<sup>(1)</sup>**



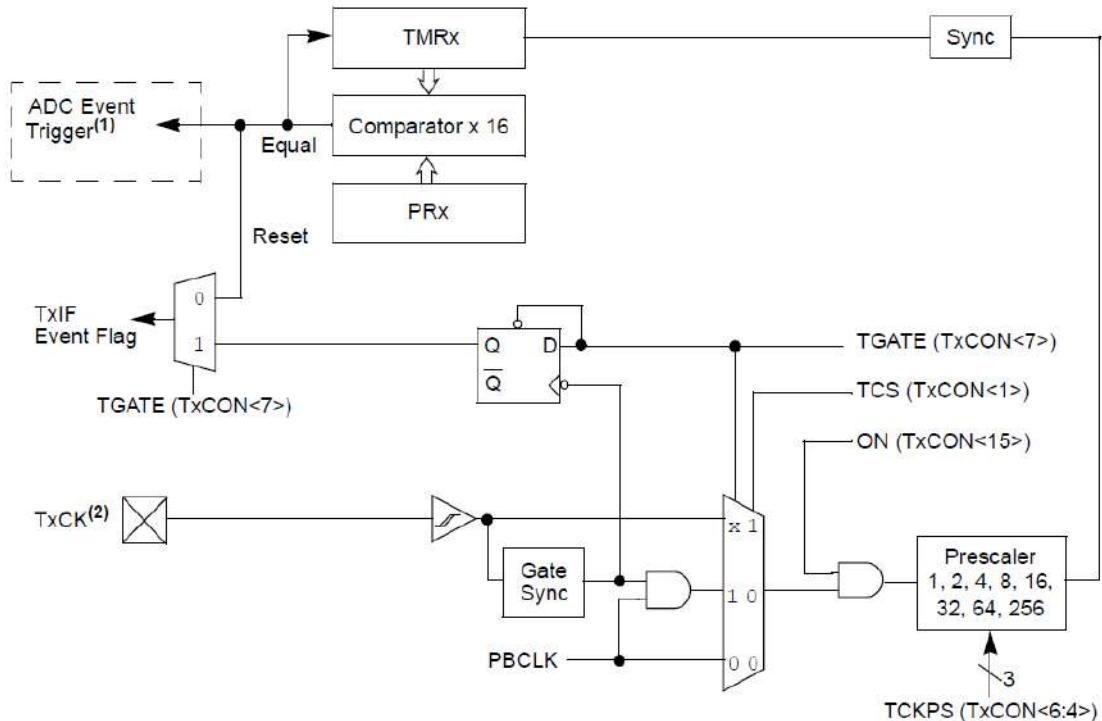
Comme on peut l'observer, le timer 1 est remis à 0 lorsque sa valeur correspond à celle de PR1. Le timer 1 est un timer 16 bits

Il dispose d'un Prescaler 1, 8, 64 ou 256.

¶ Les valeurs possibles diffèrent des autres timers !

### 6.2.4. LES TIMERS 2, 3, 4 ET 5

Voici le schéma valable pour les timers 2, 3, 4 et 5. Ce sont comme le timer 1 des timers 16 bits avec mécanisme de reset.



Note 1: ADC event trigger is available on Timer3 only.

2: TxCK pins are not available on 64-pin devices.

Comme on peut l'observer, les timers 2, 3, 4 et 5 sont remis à 0 lorsque la valeur de TMRx correspond à celle de PRx.

Ces timers disposent d'un Prescaler avec 8 valeurs possibles :

1, 2, 4, 8, 16, 32, 64 ou 256.

⚠ La valeur 128 n'est pas prévue !

### 6.2.5. PRINCIPE DE CONFIGURATION D'UN TIMER

Le principe fourni est général. Dans le cas du timer 1, il faut prendre garde aux valeurs du prescaler. Il n'est pas possible de combiner le timer 1 avec un autre timer pour former un timer 32 bits.

La configuration d'un timer consiste, en relation avec son schéma, à établir :

- La source de l'horloge
- La valeur du prescaler
- Le mode (en général 16 bits)
- Mettre à 0 le compteur
- Etablir la valeur de la période (comparateur)

Dans les exemples de configuration fournis par le MHC, le timer est stoppé avant la configuration, ce qui implique de le démarrer lorsque l'on en a besoin.

### 6.2.5.1. LE TYPE ENUMERE TMR\_MODULE\_ID

Le type énuméré TMR\_MODULE\_ID est utilisé dans toutes les fonctions pour identifier le timer que l'on manipule.

```
typedef enum {
    TMR_ID_1 = 0,
    TMR_ID_2,
    TMR_ID_4,
    TMR_ID_3,
    TMR_ID_5,
    TMR_NUMBER_OF_MODULES
} TMR_MODULE_ID;
```

Les valeurs ci-dessus correspondent au PIC32MX. Certaines autres familles de PIC32, comme les PIC32MZ, peuvent avoir plus ou moins de timers.

### 6.2.5.2. SELECTION DU "CLOCK SOURCE"

La fonction PLIB\_TMR\_ClockSourceSelect permet d'établir la source de l'horloge.

```
void PLIB_TMR_ClockSourceSelect(TMR_MODULE_ID index, TMR_CLOCK_SOURCE source);
```

#### 6.2.5.2.1. Le type énuméré TMR\_CLOCK\_SOURCE

Le type énuméré TMR\_CLOCK\_SOURCE permet d'établir l'utilisation de l'horloge interne (PB\_CLOCK) ou d'une horloge externe qui doit être connectée à la broche correspondante.

```
typedef enum {
    TMR_CLOCK_SOURCE_PERIPHERAL_CLOCK = 0,
    TMR_CLOCK_SOURCE_EXTERNAL_INPUT_PIN = 1
} TMR_CLOCK_SOURCE;
```

### 6.2.5.3. SELECTION DU "PRESCALER"

La fonction PLIB\_TMR\_PrescaleSelect permet d'établir la valeur du diviseur de la fréquence de l'horloge.

```
void PLIB_TMR_PrescaleSelect(TMR_MODULE_ID index, TMR_PRESCALE prescale);
```

#### 6.2.5.3.1. Le type énuméré TMR\_PRESCALE

Le type énuméré TMR\_PRESCALE permet de sélectionner une des valeurs de division possible.

```
typedef enum {
    TMR_PRESCALE_VALUE_1      = 0x00,
    TMR_PRESCALE_VALUE_2      = 0x01,
    TMR_PRESCALE_VALUE_4      = 0x02,
    TMR_PRESCALE_VALUE_8      = 0x03,
    TMR_PRESCALE_VALUE_16     = 0x04,
    TMR_PRESCALE_VALUE_32     = 0x05,
    TMR_PRESCALE_VALUE_64     = 0x06,
    TMR_PRESCALE_VALUE_256    = 0x07
} TMR_PRESCALE;
```

**⚠** Attention : Avec le timer 1, seulement 1, 8, 64, 256. Le MHC le signale. Si on insiste, pas d'erreur de compilation, mais problème de fonctionnement pour les valeurs qui n'existent pas.

#### 6.2.5.4. SELECTION DU MODE

Le Framework de Harmony met à disposition deux fonctions pour la sélection du mode :

Name	Description
<a href="#">PLIB_TMR_Mode16BitEnable</a>	Enables the Timer module for 16-bit operation and disables all other modes.
<a href="#">PLIB_TMR_Mode32BitEnable</a>	Enables 32-bit operation on the Timer module combination.

Pour le PIC32MX les modes 16 bits et 32 bits sont possibles sauf pour le timer 1 qui n'est que 16 bits.

##### 6.2.5.4.1. Exemple configuration 16 bits

Configuration du timer 2 en mode 16 bits :

```
/* Enable 16 bit mode */
PLIB\_TMR\_Mode16BitEnable(TMR_ID_2);
```

#### 6.2.5.5. MISE A ZERO D'UN TIMER

Cette opération n'est pas indispensable à la configuration, mais elle garantit que la première période soit correcte. On dispose de deux fonctions correspondant aux modes 16 bits et 32 bits.

```
void PLIB\_TMR\_Counter16BitClear(TMR_MODULE_ID index);
void PLIB\_TMR\_Counter32BitClear(TMR_MODULE_ID index);
```

##### 6.2.5.5.1. Exemple mise à zéro timer 16 bits

Comme le timer 2 a été configuré en mode 16 bits, on utilise la fonction suivante :

```
/* Clear counter */
PLIB\_TMR\_Counter16BitClear(TMR_ID_2);
```

#### 6.2.5.6. ETABLISSEMENT DE LA PERIODE D'UN TIMER

L'établissement de la période consiste à attribuer la valeur au registre de comparaison (PRx). On dispose de deux fonctions correspondant aux modes 16 bits et 32 bits.

```
void PLIB\_TMR\_Period16BitSet(TMR_MODULE_ID index, uint16_t period);
void PLIB\_TMR\_Period32BitSet(TMR_MODULE_ID index, uint32_t period);
```

A noter : les types entiers standards uint16\_t et uint32\_t.

##### 6.2.5.6.1. Exemple établissement période d'un timer 16 bits

Comme le timer 2 a été configuré en mode 16 bits, on utilise la fonction Period16BitSet.

```
/*Set period */
PLIB\_TMR\_Period16BitSet(TMR_ID_2, 7999);
```

⌚ On configure la valeur finale de comparaison. Dans l'exemple ci-dessus le timer comptera de 0 à 7'999 avant de reboucler. Donc 8'000 coups d'horloge par cycle.

Pour un timer 16 bits la valeur ne doit pas dépasser  $2^{16}-1 = 65'535$ .

### 6.2.5.7. LECTURE DE LA PERIODE D'UN TIMER

Il est possible d'obtenir la période d'un timer, c'est-à-dire la valeur de comparaison. Cela peut être utile en relation avec un module OC dont le timer sert de base de temps.

On dispose de deux fonctions correspondant aux modes 16 bits et 32 bits.

```
uint16_t PLIB_TMR_Period16BitGet(TMR_MODULE_ID index);  
uint32_t PLIB_TMR_Period32BitGet(TMR_MODULE_ID index);
```

### 6.2.5.8. LA FONCTION PLIB\_TMR\_STOP

Cette fonction stop/disable le timer sélectionné.

```
void PLIB_TMR_Stop(TMR_MODULE_ID index);
```

☞ Il est recommandé d'utiliser cette fonction avant une configuration/reconfiguration d'un timer.

### 6.2.5.9. LA FONCTION PLIB\_TMR\_START

Cette fonction start/enable le timer sélectionné.

```
void PLIB_TMR_Start(TMR_MODULE_ID index);
```

☞ Il est nécessaire d'utiliser cette fonction après une configuration/reconfiguration d'un timer.

## 6.2.6. EXEMPLE DE CONFIGURATION DU TIMER 1

Dans cet exemple, on configure le timer 1 pour une période de 50 ms. L'horloge avant division correspond au PB\_CLOCK de 80 MHz. Soit une période  $0,0125 \mu\text{s} = 12,5 \text{ ns}$ .

Pour obtenir une période de 50 ms, il faut compter  $50'000 / 0,0125 = 4'000'000$  ce qui est beaucoup trop grand pour un compteur 16 bits.

Remarque : au lieu de diviser par  $0,0125 [\mu\text{s}]$  il est plus simple de multiplier par  $80 [\text{MHz}]$  !

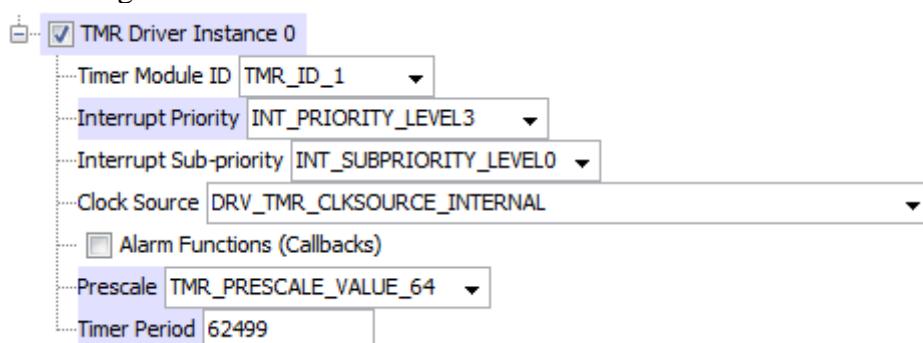
Il faut au minimum une division de  $4'000'000 / 65'536 = 61,03$ , D'où l'utilisation du prescaler de 64.

La valeur de comparaison pour obtenir la période de 50 ms sera donc :

$$N_{MAX} = (T_{VOULU} / T_{TIMER1}) - 1 = (T_{VOULU} * f_{TIMER1}) - 1 = (50'000 * (80 / 64)) - 1 = 62'499$$

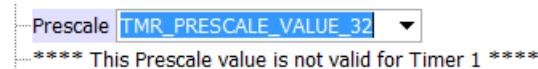
### 6.2.6.1. CONFIGURATION TIMER 1 AU NIVEAU MHC

Voici la configuration au niveau du MHC.



### 6.2.6.1.1. Configuration timer 1 au niveau MHC, signalisation erreur

Si on utilise un prescaler non supporté par le timer 1, on obtient :



### 6.2.6.2. FONCTION DE CONFIGURATION DU TIMER 1 OBTENUE

Voici la fonction de configuration obtenue du MHC. Cette fonction prépare partiellement l'interruption du timer 1.

```
void DRV_TMR0_Initialize(void)
{
    PLIB_TMR_Stop(TMR_ID_1); /* Disable Timer */
    /* Select clock source */
    PLIB_TMR_ClockSourceSelect(TMR_ID_1,
                               TMR_CLOCK_SOURCE_PERIPHERAL_CLOCK);
    /* Select prescalar value */
    PLIB_TMR_PrescaleSelect(TMR_ID_1, TMR_PRESCALE_VALUE_64);
    PLIB_TMR_Mode16BitEnable(TMR_ID_1); // 16 bit mode
    PLIB_TMR_Counter16BitClear(TMR_ID_1); // Clear counter
    PLIB_TMR_Period16BitSet(TMR_ID_1, 62499); // Set period

    /* Setup Interrupt */
    PLIB_INT_VectorPrioritySet(INT_ID_0, INT_VECTOR_T1,
                               INT_PRIORITY_LEVEL3);
    PLIB_INT_VectorSubPrioritySet(INT_ID_0, INT_VECTOR_T1,
                               INT_SUBPRIORITY_LEVEL0);
}
```

☝ L'interruption n'est pas activée ici. Cela sera réalisé au démarrage du timer (voir ci-dessous).

### 6.2.6.3. LANCEMENT DU TIMER 1

Le timer est stoppé lors de son initialisation. Pour le lancer, il sera nécessaire dans l'initialisation de l'application d'effectuer un appel à la fonction **DRV\_TMR0\_Start**, qui elle-même appelle la fonction **\_DRV\_TMR0\_Resume**.

```
static void _DRV_TMR0_Resume(bool resume)
{
    if (resume)
    {
        PLIB_INT_SourceFlagClear(INT_ID_0,
                               INT_SOURCE_TIMER_1);
        PLIB_INT_SourceEnable(INT_ID_0,
                               INT_SOURCE_TIMER_1);
        PLIB_TMR_Start(TMR_ID_1);
    }
}
```

La fonction **\_DRV\_TMR0\_Resume** autorise l'interruption du timer 1 avant d'effectuer le start.

```

bool DRV_TMR0_Start(void)
{
    /* Start Timer*/
    DRV_TMR0_Resume(true);
    DRV_TMR0_Running = true;

    return true;
}

```

### 6.2.7. EXEMPLE CONFIGURATION DU TIMER 2

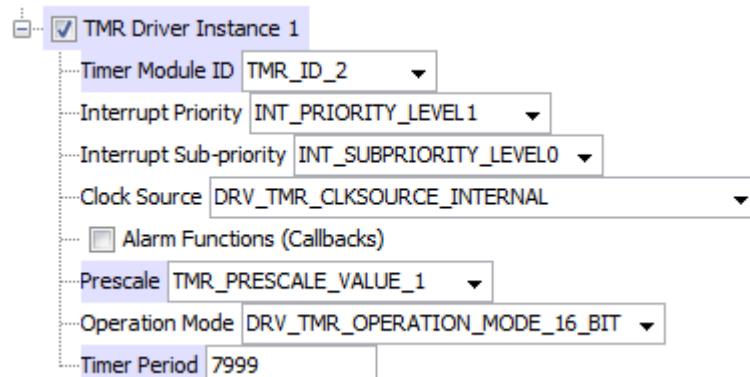
Dans cet exemple, on configure le timer 2 pour une période de 100 µs. L'horloge avant division correspond au PB\_CLOCK de 80 MHz. Soit une période 0,0125 µs.

Pour obtenir une période de 100 µs, il faut paramétriser :

$N_{MAX} = (T_{VOLU} / T_{TIMER2}) - 1 = (100 / 0,0125) - 1 = 7'999$  ou  $(100 \mu s * 80 \text{ MHz}) - 1 = 7'999$ , ce qui est correct pour un compteur 16 bits. Un prescaler de 1 convient donc.

#### 6.2.7.1. CONFIGURATION TIMER 2 AU NIVEAU MHC

Voici la configuration au niveau du MHC.



#### 6.2.7.2. FONCTION DE CONFIGURATION DU TIMER 2 OBTENUE

Voici la fonction de configuration obtenue du MHC. Cette fonction prépare aussi partiellement l'interruption du timer 2.

```

void DRV_TMR1_Initialize(void)
{
    /* Initialize Timer Instance1 */
    /* Disable Timer */
    PLIB_TMR_Stop(TMR_ID_2);
    /* Select clock source */
    PLIB_TMR_ClockSourceSelect(TMR_ID_2,
                                TMR_CLOCK_SOURCE_PERIPHERAL_CLOCK);
    /* Select prescalar value */
    PLIB_TMR_PrescaleSelect(TMR_ID_2,
                            TMR_PRESCALE_VALUE_1);
    /* Enable 16 bit mode */
    PLIB_TMR_Mode16BitEnable(TMR_ID_2);
    /* Clear counter */
    PLIB_TMR_Counter16BitClear(TMR_ID_2);
}

```

```

/*Set period */
PLIB_TMR_Period16BitSet(TMR_ID_2, 7999);

/* Setup Interrupt */
PLIB_INT_VectorPrioritySet(INT_ID_0, INT_VECTOR_T2,
                            INT_PRIORITY_LEVEL1);
PLIB_INT_VectorSubPrioritySet(INT_ID_0,
                               INT_VECTOR_T2, INT_SUBPRIORITY_LEVEL0);
}

§ Comme nous n'avons pas besoin de l'interruption du timer 2, l'autorisation de la
source d'interruption doit être mise en commentaire dans la fonction
DRV_TMR1_Resume.

static void _DRV_TMR1_Resume(bool resume)
{
    if (resume)
    {
        PLIB_INT_SourceFlagClear(INT_ID_0,
                                  INT_SOURCE_TIMER_2);
        // PLIB_INT_SourceEnable(INT_ID_0,
        //                         INT_SOURCE_TIMER_2);
        PLIB_TMR_Start(TMR_ID_2);
    }
}

bool DRV_TMR1_Start(void)
{
    /* Start Timer*/
    _DRV_TMR1_Resume(true);
    DRV_TMR1_Running = true;

    return true;
}

```

### 6.2.7.3. LANCEMENT DU TIMER 2

Comme le timer 2 est stoppé au début de la configuration, il est nécessaire de le démarrer en utilisant la fonction DRV\_TMR1\_Start().

### 6.2.8. EXEMPLE CONFIGURATION DU TIMER 3

Dans cet exemple, on configure le timer 3 pour une période de 10 ms. L'horloge avant division correspond au PB\_CLOCK de 80 MHz. Soit une période 0,0125 µs.

Pour obtenir une période de 10 ms, il faut compter  $(10'000 * 80)-1 = 799'999$ , ce qui est beaucoup trop grand pour un compteur 16 bits.

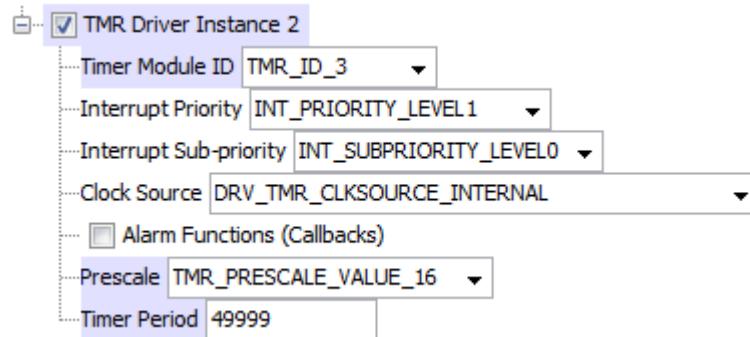
Il faut au minimum une division de  $800'000 / 65536 = 12,2$ , ce qui nous conduit à utiliser un diviseur de 16 qui existe pour le timer 3.

La valeur de comparaison pour obtenir la période de 10 ms sera donc :

$$N_{MAX} = (T_{VOULU} / T_{TIMER3}) - 1 = (T_{VOULU} * f_{TIMER3}) - 1 = (10'000 * (80 / 16)) - 1 = 49'999.$$

### 6.2.8.1. CONFIGURATION TIMER 3 AU NIVEAU MHC

Voici la configuration au niveau du MHC :



### 6.2.8.2. FONCTION DE CONFIGURATION DU TIMER 3 OBTENUE

Voici la fonction de configuration obtenue du MHC. Cette fonction prépare aussi l'interruption du timer 3 (Mise en commentaire).

```
void DRV_TMR2_Initialize(void)
{
    PLIB_TMR_Stop(TMR_ID_3); /* Disable Timer */
    /* Select clock source */
    PLIB_TMR_ClockSourceSelect(TMR_ID_3,
                               TMR_CLOCK_SOURCE_PERIPHERAL_CLOCK);
    /* Select prescalar value */
    PLIB_TMR_PrescaleSelect(TMR_ID_3,
                           TMR_PRESCALE_VALUE_16);
    /* Enable 16 bit mode */
    PLIB_TMR_Mode16BitEnable(TMR_ID_3);
    /* Clear counter */
    PLIB_TMR_Counter16BitClear(TMR_ID_3);
    // Periode 7 ms
    PLIB_TMR_Period16BitSet(TMR_ID_3, 49999);

    /* Setup Interrupt */
    PLIB_INT_VectorPrioritySet(INT_ID_0, INT_VECTOR_T3,
                             INT_PRIORITY_LEVEL1);
    PLIB_INT_VectorSubPrioritySet(INT_ID_0,
                               INT_VECTOR_T3, INT_SUBPRIORITY_LEVEL0);
}
```

✍ Comme nous n'avons pas besoin de l'interruption du timer 3, l'autorisation de la source d'interruption doit être mise en commentaire dans la fonction **DRV\_TMR2\_Resume**. On aurait également pu fixer le niveau de priorité d'interruption à 0.

```
static void _DRV_TMR2_Resume(bool resume)
{
    if (resume)
    {
        PLIB_INT_SourceFlagClear(INT_ID_0,
                                  INT_SOURCE_TIMER_3);
        // PLIB_INT_SourceEnable(INT_ID_0,
        //                         INT_SOURCE_TIMER_3);
        PLIB_TMR_Start(TMR_ID_3);
    }
}

bool DRV_TMR2_Start(void)
{
    /* Start Timer*/
    _DRV_TMR2_Resume(true);
    DRV_TMR2_Running = true;

    return true;
}
```

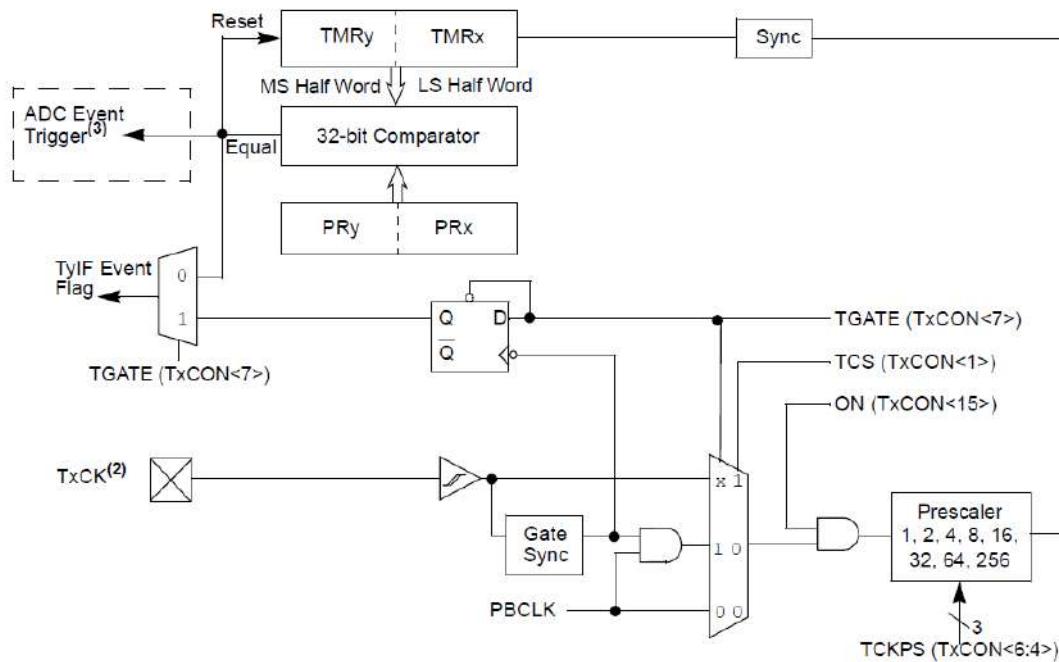
#### 6.2.8.3. LANCEMENT DU TIMER 3

Comme le timer 3 est stoppé au début de la configuration, il est nécessaire de le démarrer en utilisant la fonction DRV\_TMR2\_Start() qui elle-même appelle finalement PLIB\_TMR\_Start().

### 6.2.9. LES TIMERS 32 BITS

On dispose de deux timers 32 bits en utilisant la paire 2, 3 ainsi que la paire 4, 5. Par rapport à un timer 16 bits auquel on devrait adjoindre un prescaler, un timer 32 bits présente l'avantage d'avoir une finesse de réglage du cycle égale au clock du timer. Avec un SYSCLK à 80 MHz, on peut donc régler une période maximale de  $2^{32} * 12,5 \text{ ns} = 53,7 \text{ s}$  à 12,5 ns près (dans le cas où un comptage sur 32 bits est suffisant).

Voici le schéma de principe valable pour les 2 paires.



Note 1: In this diagram, the use of 'x' in registers, TxCON, TMRx, PRx and TxCK, refers to either Timer2 or Timer4; the use of 'y' in registers, TyCON, TMRy, PRy, TyIF, refers to either Timer3 or Timer5.

2: TxCK pins are not available on 64-pin devices.

3: ADC event trigger is available only on the Timer2/3 pair.

Pour la configuration le principe est le même sauf que l'on utilise le mode 32 bits.

- ⌚ Pour la paire des timers 2 & 3 on configure le timer 2 avec interruption sur timer 3.
- ⌚ Pour la paire des timers 4 & 5 on configure le timer 4 avec interruption sur timer 5.

### 6.2.10. TIMER 32 BITS, EXEMPLE

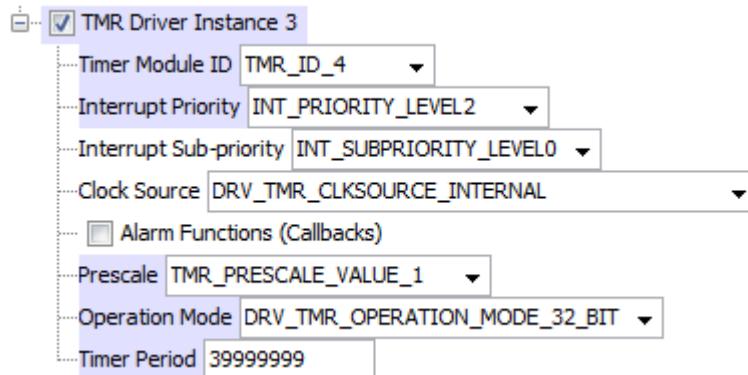
Dans cet exemple, configuration de la paire 4 & 5, pour une période de 500 ms. L'horloge avant division correspond au PB\_CLOCK de 80 MHz. Soit une période 0,0125 µs. Pour obtenir une période de 500 ms, il faut compter

$N_{MAX} = (T_{VOULU} / T_{TIMER45}) - 1 = (T_{VOULU} * f_{TIMER45}) - 1 = (500'000 * 80) - 1 = 39'999'999$   
ce qui est supportable pour un compteur 32 bits.

(Valeur max =  $2^{32} - 1 = 4'294'967'295$ ).

### 6.2.10.1. CONFIGURATION AU NIVEAU MHC DE LA PAIRE TIMERS 4 & 5

Voici la configuration de la paire de timer 4 & 5.



### 6.2.10.2. FONCTION DE CONFIGURATION OBTENUE POUR LA PAIRE 4 & 5

Voici la fonction de configuration obtenue du MHC (configuration du **timer 4**) avec l'ajout du Start. La fonction comporte la préparation de l'interruption pour le **timer 5**.

```
void DRV_TMR3_Initialize(void)
{
    PLIB_TMR_Stop(TMR_ID_4); /* Disable Timer */
    PLIB_TMR_ClockSourceSelect(TMR_ID_4,
                               TMR_CLOCK_SOURCE_PERIPHERAL_CLOCK);
    /* Select prescalar value */
    PLIB_TMR_PrescaleSelect(TMR_ID_4, TMR_PRESCALE_VALUE_1);

    /* Enable 32 bit mode */
    PLIB_TMR_Mode32BitEnable(TMR_ID_4);
    PLIB_TMR_Counter32BitClear(TMR_ID_4); // Clear counter
    /*Set 32 bits period */
    PLIB_TMR_Period32BitSet(TMR_ID_4, 39999999);

    /* Setup Interrupt */
    PLIB_INT_VectorPrioritySet(INT_ID_0, INT_VECTOR_T5,
                               INT_PRIORITY_LEVEL2);
    PLIB_INT_VectorSubPrioritySet(INT_ID_0, INT_VECTOR_T5,
                               INT_SUBPRIORITY_LEVEL0);
}
```

Cette configuration utilise les fonctions 32 bits pour la configuration du timer. Il est à remarquer qu'au niveau des interruptions, c'est le **timer 5 (poids fort)** qui est source de l'interruption. La source est autorisée dans la fonction **\_DRV\_TMR3\_Resume**.

```

static void _DRV_TMR3_Resume(bool resume)
{
    if (resume)
    {
        PLIB_INT_SourceFlagClear(INT_ID_0,
                                  INT_SOURCE_TIMER_5);
        PLIB_INT_SourceEnable(INT_ID_0,
                              INT_SOURCE_TIMER_5);
        PLIB_TMR_Start(TMR_ID_4);
    }
}

bool DRV_TMR3_Start(void)
{
    /* Start Timer*/
    _DRV_TMR3_Resume(true);
    DRV_TMR3_Running = true;
    return true;
}

```

#### 6.2.10.3. LANCEMENT DES TIMERS 4 & 5

Comme le timer 4 est stoppé au début de la configuration, il est nécessaire de le démarrer en utilisant la fonction DRV\_TMR2\_Start() qui elle-même appelle finalement PLIB\_TMR\_Start().

Remarque : la fonction démarre le timer 4 qui a le rôle de poids faible dans la paire.

#### 6.2.10.4. REPONSE INTERRUPTION POUR LA PAIRE TIMERS 4 & 5

Voici la routine de réponse à l'interruption de la paire timers 4 &5. L'interruption est liée au timer 5, qui correspond au poids fort.

```

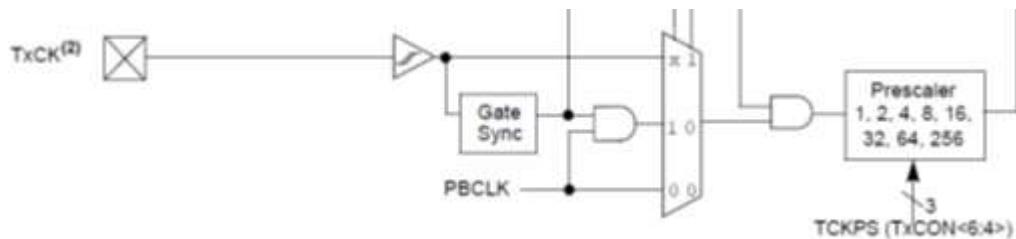
// Réponse à l'interruption du Timer5
void __ISR(_TIMER_5_VECTOR, ipl2AUTO)
{
    _IntHandlerDrvTmrInstance2(void)
{
    PLIB_INT_SourceFlagClear(INT_ID_0, INT_SOURCE_TIMER_5);
    BSP_LEDToggle(BSP_LED_4);
}

```

On obtient bien une inversion de la sortie toutes les 0,5 secondes.

### 6.2.11. LES TIMERS EN COMPTAGE EXTERNE

La fonction **PLIB\_TMR\_ClockSourceSelect** permet d'établir la source de l'horloge.



On a le choix entre le PBCLK (horloge interne) et un signal d'horloge externe qui vient sur une broche TxCK.

Le comptage externe permet le comptage d'impulsions en provenance d'un capteur incrémental ou autre. Cela permet, associé à une base de temps, de réaliser une mesure de fréquence.

Il faut établir séparément la broche TxCK en entrée !

#### 6.2.11.1. LISTE DES ENTRÉES DE COMPTAGE TxCK

Pour le PIC32MX795F512L 100 pins TQFP :

TxCK	Nom de la broche	No de la broche
T1CK	SOSC0/T1CK/CN0/RC14	74
T2CK	T2CK/RC1	6
T3CK	T3CK/AC2TX/RC2	7
T4CK	T4CK/AC2RX/RC3	8
T5CK	T5CK/SDI1/RC4	9

#### 6.2.11.2. SELECTION SOURCE EXTERNE, EXEMPLE

Voici un exemple de sélection de l'horloge externe avec le timer 3 en y ajoutant la synchronisation du signal.

```
PLIB_TMR_ClockSourceSelect(TMR_ID_3,
                           TMR_CLOCK_SOURCE_EXTERNAL_INPUT_PIN);
PLIB_TMR_ClockSourceExternalSyncEnable(TMR_ID_3);
```

#### 6.2.11.3. LA FONCTION PLIB\_TMR\_CLOCKSOURCEEXTERNALSYNCENABLE

La fonction **PLIB\_TMR\_ClockSourceExternalSyncEnable** permet d'introduire le passage du signal externe par un mécanisme de synchronisation. Cette fonction n'a d'effet que si la source externe a été sélectionnée.

```
void PLIB_TMR_ClockSourceExternalSyncEnable(TMR_MODULE_ID index);
```

#### 6.2.11.4. LA FONCTION PLIB\_TMR\_CLOCKSOURCEEXTERNALSYNCDISABLE

La fonction **PLIB\_TMR\_ClockSourceExternalSyncDisable** permet de revenir à la situation sans synchronisation.

```
void PLIB_TMR_ClockSourceExternalSyncDisable(TMR_MODULE_ID index);
```

### 6.2.11.5.FONCTIONS D'ACCES AU COMPTEUR DU TIMER

On dispose des fonctions Clear, Set et Get pour modifier ou lire la valeur du compteur.

Name	Description
<a href="#">PLIB_TMR_Counter16BitClear</a>	Clears the 16-bit timer value.
<a href="#">PLIB_TMR_Counter16BitGet</a>	Gets the 16-bit timer value.
<a href="#">PLIB_TMR_Counter16BitSet</a>	Sets the 16-bit timer value.
<a href="#">PLIB_TMR_Counter32BitClear</a>	Clears the 32-bit timer value.
<a href="#">PLIB_TMR_Counter32BitGet</a>	Gets the 32-bit timer value.
<a href="#">PLIB_TMR_Counter32BitSet</a>	Sets the 32-bit timer value.

### 6.2.11.6.LA FONCTION PLIB\_TMR\_COUNTER16BITGET

Cette fonction permet de lire la valeur du timer (compteur).

```
uint16_t PLIB_TMR_Counter16BitGet(TMR_MODULE_ID index);
```

Cette fonction effectue la lecture sans précaution particulière par rapport à un incrément en cours.

Exemple :

```
uint16_t Timer3Value = PLIB_TMR_Counter16BitGet(TMR_ID_3);
```

### 6.2.11.7.LA FONCTION DRV\_TMRX\_COUNTERVALUEGET

Le fichier `drv_tmr_static.c` fournit pour chaque instance une fonction `DRV_TMRx.CounterValueGet`. Cette fonction permet de lire la valeur du timer (compteur) en utilisant la fonction `PLIB_TMR_Counter16BitGet`.

```
uint32_t DRV_TMR2_CounterValueGet(void)
{
    /* Get 16-bit counter value*/
    return (uint32_t) PLIB_TMR_Counter16BitGet(TMR_ID_3);
}
```

Exemple :

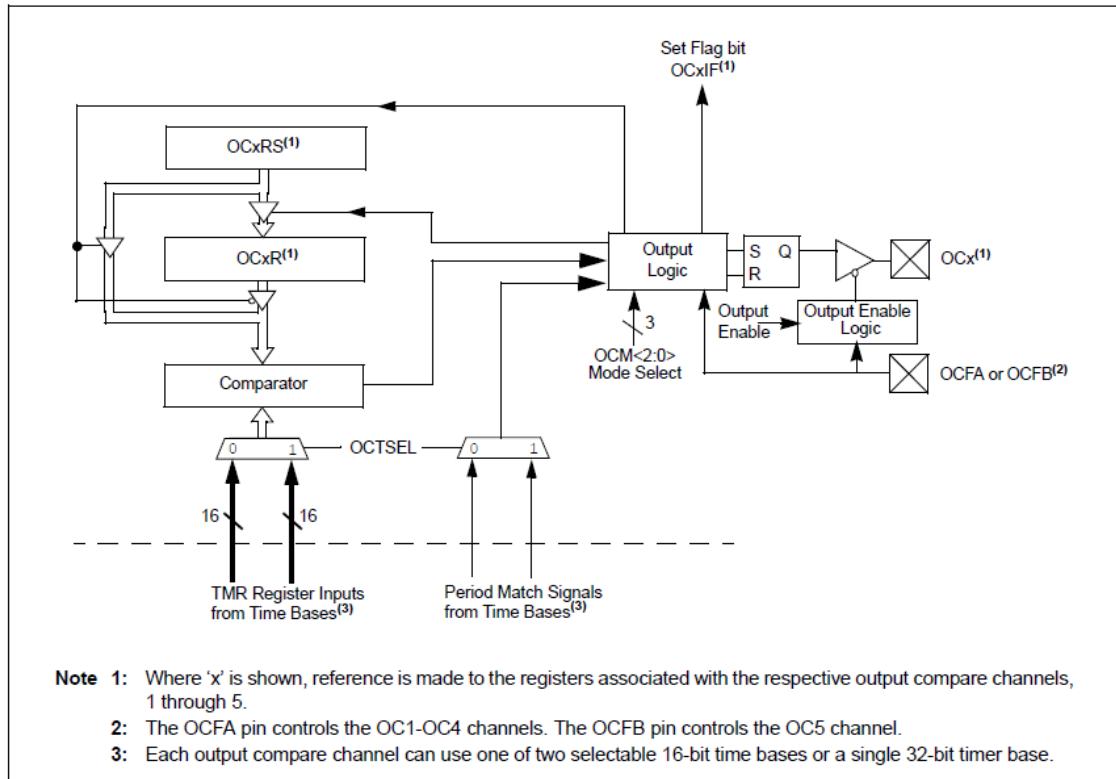
```
uint16_t Timer3Value = DRV_TMR2_CounterValueGet();
```

## 6.3. LES MODULES "OUTPUT COMPARE"

Le PIC32MX795F512L possède 5 modules "Output Compare", nommés OC1 à OC5. Ces modules nécessitent un timer et permettent de générer des impulsions, ou par exemple un signal PWM.

### 6.3.1. SCHÉMA BLOC DU MODULE "OUTPUT COMPARE"

FIGURE 16-1: OUTPUT COMPARE MODULE BLOCK DIAGRAM



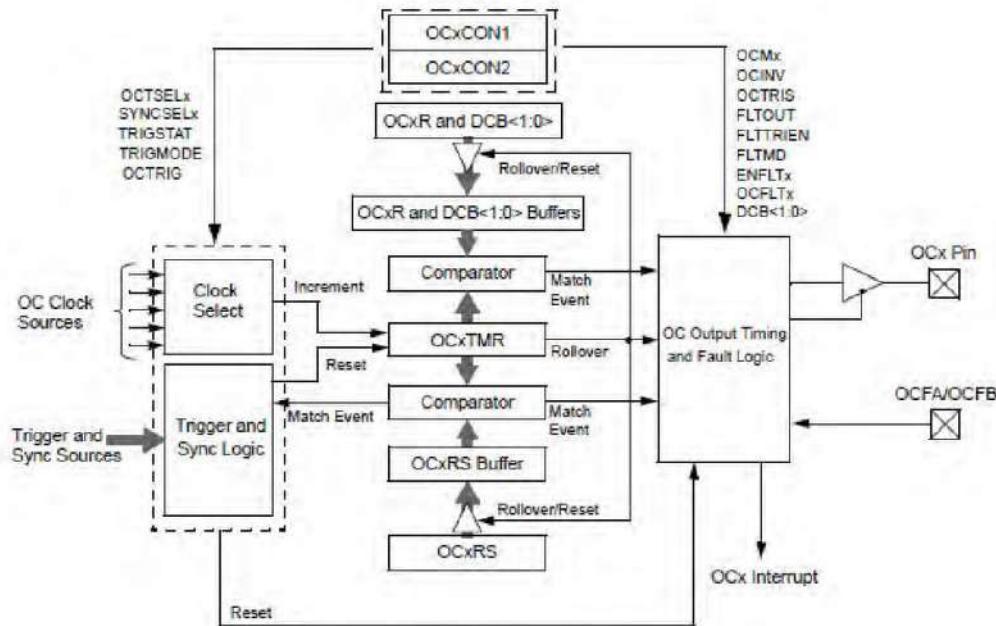
:( Seulement 2 timers 16 bits possibles comme source : timer 2 et timer 3 !

### 6.3.2. LISTE DES SORTIES DE COMPARAISONS

Pour le PIC32MX795FL512L 100 pin TQFP :

OCx	Nom de la broche	No de la broche
OC1	SDO1/OC1/INT0/RD0	72
OC2	OC2/RD1	76
OC3	OC3/RD2	77
OC4	OC4/RD3	78
OC5	OC5/PMWR/CN13/RD4	81

### 6.3.1. PRINCIPE FONCTIONNEMENT DES OC



Le schéma de principe ci-dessus nous montre qu'il y a **2 comparateurs** gérant la sortie. Dans les modes PWM et DUAL\_COMPARE, une des valeurs de référence est utilisée pour obtenir le flanc montant du signal et l'autre le flanc descendant.

Avec **PLIB\_OC\_Buffer16BitSet(OC\_ID\_2, 0)**, on établit la valeur de référence du comparateur pour le flanc montant. Dans cet exemple, 0 signifie au début de la période du timer.

Avec **PLIB\_OC\_PulseWidth16BitSet(OC\_ID\_2, 1000)**, on établit la valeur de référence du comparateur pour le flanc descendant. Dans cet exemple, 1'000 correspond à la moitié de la période du timer.

Pour modifier le rapport cyclique du signal, on modifie la valeur de référence pour le flanc descendant en utilisant la fonction **PLIB\_OC\_PulseWidth16BitSet**.

### 6.3.2. FONCTIONS DE LA PLIB\_OC

Il faut se référer à la section Output Compare Peripheral Library de la documentation Harmony.

Voici une sélection des fonctions utilisables avec le PIC32MX :

<a href="#">PLIB_OC_Disable</a>	Disable the OC module
<a href="#">PLIB_OC_Enable</a>	Enables the OC module.
<a href="#">PLIB_OC_ModeSelect</a>	Selects the compare mode for the OC module.
<a href="#">PLIB_OC_TimerSelect</a>	Selects a clock source for the OC module.
<a href="#">PLIB_OC_Buffer16BitSet</a>	Sets a 16-bit primary compare value for compare operations.
<a href="#">PLIB_OC_Buffer32BitSet</a>	Sets a 32-bit primary compare value for compare operations.
<a href="#">PLIB_OC_BufferSizeSelect</a>	Sets the buffer size and pulse width to 16-bits or 32-bits.
<a href="#">PLIB_OC_PulseWidth16BitSet</a>	Sets a 16-bit pulse width for OC module output.
<a href="#">PLIB_OC_PulseWidth32BitSet</a>	Sets a 32-bit pulse width for OC module output.

### 6.3.3. ACTIONS POSSIBLES

Les actions possibles sont les suivantes :

Each Output Compare module has the following modes of operation:

- Single Compare Match mode
  - With output drive high
  - With output drive low
  - With output drive toggles
- Dual Compare Match mode
  - With single output pulse
  - With continuous output pulses
- Simple Pulse-Width Modulation mode
  - Without fault protection input
  - With fault protection input

### 6.3.4. FONCTIONS POUR CONFIGURER LES MODULES OC

Pour réaliser la configuration d'une Output Compare (OC), il est nécessaire d'effectuer les opérations suivantes :

- Sélection du mode (ModeSelect)
- Sélection 16 bits ou 32 bits (BufferSizeSelect)
- Sélection du timer à comparer (TimerSelect)
- FaultInputSelect
- Initialisation du Buffer
- Initialisation la largeur d'impulsion (PulseWidth Set)

A la fin de la configuration, il faut enclencher le module OC avec la fonction Enable.

#### 6.3.4.1. LE TYPE ÉNUMÉRÉ OC\_MODULE\_ID

Le type énuméré OC\_MODULE\_ID définit les 5 modules OC :

```
typedef enum {
    OC_ID_1 = 0,
    OC_ID_2,
    OC_ID_3,
    OC_ID_4,
    OC_ID_5,
    OC_NUMBER_OF_MODULES
} OC_MODULE_ID;
```

#### 6.3.4.2. LA FONCTION PLIB\_OC\_MODESELECT

La fonction **PLIB\_OC\_ModeSelect** permet de sélectionner un des modes de fonctionnement de l'OC.

```
void PLIB_OC_ModeSelect(OC_MODULE_ID index, OC_COMPARE_MODES cmpMode);
```

### 6.3.4.2.1. Le type énuméré OC\_COMPARE\_MODES

Le type énuméré OC\_COMPARE\_MODES est défini ainsi :

```
typedef enum {
    OC_COMPARE_TURN_OFF_MODE = 0,
    OC_SET_HIGH_SINGLE_PULSE_MODE = 1,
    OC_SET_LOW_SINGLE_PULSE_MODE = 2,
    OC_TOGGLE_CONTINUOUS_PULSE_MODE = 3,
    OC_DUAL_COMPARE_SINGLE_PULSE_MODE = 4,
    OC_DUAL_COMPARE_CONTINUOUS_PULSE_MODE = 5,
    OC_COMPARE_PWM_EDGE_ALIGNED_MODE = 6,
    OC_COMPARE_PWM_MODE_WITHOUT_FAULT_PROTECTION = 6,
    OC_COMPARE_PWM_MODE_WITH_FAULT_PROTECTION = 7
} OC_COMPARE_MODES;
```

### 6.3.4.2.2. Description des modes

Members	Description
OC_COMPARE_PWM_MODE_WITH_FAULT_PROTECTION	<p>Output Compare module output is PWM signal and is fault protected if fault protection pin is enabled. Fault protection is valid if the fault pin is enabled in the hardware. Fault pin: OCFA for OC_ID_1 to OC_ID_4 , OCFB for OC_ID_5 in MX devices. OCFA for OC_ID_1 to OC_ID_3 and OC_ID_7 to OC_ID_9 , OCFB for OC_ID_4 to OC_ID_6 in PIC32MZ devices. If a logic ‘0’ is detected on the OCFA/OCFB pin, the selected PWM output pin(s) are placed in the tri-state. The user may elect to provide a pull-down or pull-up resistor on the PWM pin to provide for a desired state if a Fault condition occurs. The shutdown of the PWM output is immediate and is not tied to the device clock source. Fault occurrence can be detected by calling the function <a href="#">PLIB_OC_FaultHasOccurred</a>. The Output Compare will be disabled until the following conditions are met:</p> <ol style="list-style-type: none"> <li>1. The external Fault condition has been removed</li> <li>2. The PWM mode is re-enabled</li> </ol>
OC_COMPARE_PWM_MODE_WITHOUT_FAULT_PROTECTION	Output Compare module output is PWM signal and is not fault protected
OC_COMPARE_PWM_EDGE_ALIGNED_MODE	This element is obsolete and it will be removed from next release
OC_DUAL_COMPARE_CONTINUOUS_PULSE_MODE	Dual Compare, Continuous Pulse mode: Output Compare module output is driven high on compare match with primary compare value and driven low on compare match with secondary compare value. A continuous stream of pulses is generated unless the compare mode is changed or the module is disabled. If the secondary compare value is greater than time base period value, secondary compare match does not occur. As a consequence, Output Compare module output stays high.
OC_DUAL_COMPARE_SINGLE_PULSE_MODE	Dual Compare, Single Pulse mode: Output Compare module output is driven high on compare match with primary compare value and driven low on compare match with secondary compare value. If the secondary compare value is greater than time base period value, secondary compare match does not occur. As a consequence, Output Compare module output stays high until a mode change is made or module is disabled
OC_TOGGLE_CONTINUOUS_PULSE_MODE	Single Compare Toggle mode: Output Compare module output is initialized to Low. Output Compare module output toggles at every compare match event with primary compare value with a single peripheral bus clock cycle delay. This scheme generates a square wave with 50% duty cycle. An interrupt is generated each time the output toggles.
OC_SET_LOW_SINGLE_PULSE_MODE	Single Compare Set Low mode: A compare match event with primary compare value will set the output of Output Compare module 'Low' with a single peripheral bus clock cycle delay. Output stays Low unless Output Compare module is disabled or a new compare mode is chosen. An interrupt is generated at compare match event. Output Compare module output is initially forced High.
OC_SET_HIGH_SINGLE_PULSE_MODE	Single Compare Set High mode: A compare match event with primary compare value will set the output of Output Compare module 'High' with a single peripheral bus clock cycle delay. Output stays High unless Output Compare module is disabled or a new compare mode is chosen. An interrupt is generated at compare match event. Output Compare module output is initially forced Low.
OC_COMPARE_TURN_OFF_MODE	Turn OFF mode: Output Compare module is disabled but still draws current. This mode is used to temporarily turn OFF the Output Compare module before a new compare mode is selected

Les deux modes qui semblent le mieux convenir pour générer des signaux PWM simples sont les modes OC\_DUAL\_COMPARE\_CONTINUOUS\_PULSE\_MODE et OC\_COMPARE\_PWM\_MODE\_WITHOUT\_FAULT\_PROTECTION.

#### 6.3.4.2.3. Correspondance avec OCXCON

On peut essayer de vérifier la correspondance avec la configuration du registre OCXCON.

**Register 16-1: OCxCON: Output Compare 'x' Control Register (Continued)**

bit 2-0	OCM<2:0>: Output Compare Mode Select bits
	111 = PWM mode on OCx; Fault pin enabled
	110 = PWM mode on OCx; Fault pin disabled
	101 = Initialize OCx pin low; generate continuous output pulses on OCx pin
	100 = Initialize OCx pin low; generate single output pulse on OCx pin
	011 = Compare event toggles OCx pin
	010 = Initialize OCx pin high; compare event forces OCx pin low
	001 = Initialize OCx pin low; compare event forces OCx pin high
	000 = Output compare peripheral is disabled but continues to draw current

**Note 1:** Reads as '0' in modes other than PWM mode.

Le mode OC\_COMPARE\_PWM\_MODE\_WITHOUT\_FAULT\_PROTECTION = 6 correspond à 110 = PWM mode on OCx; Fault pin disabled

Le mode OC\_DUAL\_COMPARE\_CONTINUOUS\_PULSE\_MODE = 5 correspond à 101 = Initialize OCx pin low; generate continuous output pulses on OCx pin

#### 6.3.4.3. LA FONCTION PLIB\_OC\_BUFFERSIZESELECT

La fonction **PLIB\_OC\_BufferSizeSelect** permet d'indiquer si l'OC travaille en 16 bits ou en 32 bits. Dans le cas 32 bits, la comparaison est effectuée avec la paire de timers 2&3.

```
void PLIB_OC_BufferSizeSelect(OC_MODULE_ID index, OC_BUFFER_SIZE size);
```

**Returns**

None.

**Description**

This function sets the size of the buffer and pulse width to 16-bits or 32-bits. The choice is made based on whether a 16-bit timer or a 32-bit timer is selected.

Le type énuméré OC\_BUFFER\_SIZE est défini ainsi :

```
typedef enum {
    OC_BUFFER_SIZE_16BIT = 0,
    OC_BUFFER_SIZE_32BIT = 1
} OC_BUFFER_SIZE;
```

#### 6.3.4.4. LA FONCTION PLIB\_OC\_TIMERSELECT

La fonction **PLIB\_OC\_TimerSelect** permet de choisir le timer à comparer.

```
void PLIB_OC_TimerSelect(OC_MODULE_ID index, OC_16BIT_TIMERS tmr);
```

Le type énuméré OC\_16BIT\_TIMERS est défini ainsi :

```
typedef enum {
    OC_TIMER_16BIT_TMR2 = 0,
    OC_TIMER_16BIT_TMR3 = 1
} OC_16BIT_TIMERS;
```

¶ Nous sommes limités aux timers 2 & 3, ce qui implique aussi que dans le cas du fonctionnement en 32 bits, la sélection n'a plus de sens.

#### 6.3.4.5. LA FONCTION PLIB\_OC\_FAULTINPUTSELECT

La fonction **PLIB\_OC\_FaultInputSelect** permet de déterminer si on utilise ou non les entrées d'erreur OCFA et OCFB. OCFA est utilisée pour les OC1 à OC4 et OCFB avec OC5.

```
void PLIB_OC_FaultInputSelect(OC_MODULE_ID index, OC_FAULTS flt);
```

Le type énuméré OC\_FAULTS n'offre que 2 possibilités.

```
typedef enum {
    OC_FAULT_PRESET = 7,
    OC_FAULT_DISABLE = 6
} OC_FAULTS;
```

#### 6.3.4.6. LA FONCTION PLIB\_OC\_BUFFER16BITSET

La fonction **PLIB\_OC\_Buffer16BitSet** établit la valeur du "primary compare" dans tous les modes sauf en PWM, ceci pour une configuration 16 bits.

```
void PLIB_OC_Buffer16BitSet(OC_MODULE_ID index, uint16_t val16Bit);
```

#### 6.3.4.7. LA FONCTION PLIB\_OC\_BUFFER32BITSET

La fonction **PLIB\_OC\_Buffer32BitSet** établit la valeur du "primary compare" dans tous les modes sauf en PWM, ceci pour une configuration 32 bits.

```
void PLIB_OC_Buffer32BitSet(OC_MODULE_ID index, uint32_t val32Bit);
```

#### 6.3.4.8. LA FONCTION PLIB\_OC\_PULSEWIDTH16BITSET

La fonction **PLIB\_OC\_PulseWidth16BitSet** permet d'établir la largeur d'impulsion positive du signal PWM ou continuous pulse, ceci en 16 bits.

```
void PLIB_OC_PulseWidth16BitSet(OC_MODULE_ID index, uint16_t pulseWidth);
```

¶ La valeur fournie doit être comprise entre 0 et la valeur maximum prévue pour le timer associé.

### 6.3.4.9. LA FONCTION PLIB\_OC\_PULSEWIDTH32BITSET

La fonction **PLIB\_OC\_PulseWidth32BitSet** permet d'établir la largeur d'impulsion positive du signal PWM ou continuous pulse, ceci en 32 bits.

```
void PLIB_OC_PulseWidth32BitSet(OC_MODULE_ID index, uint32_t pulseWidth);
```

☝ La valeur fournie doit être comprise entre 0 et la valeur maximum prévue pour la paire de timers 2 & 3.

#### 6.3.4.9.1. La fonction PLIB\_OC\_Enable

La fonction **PLIB\_OC\_Enable** active le module OC. Elle ne doit être utilisée qu'à la fin de la séquence de configuration.

```
void PLIB_OC_Enable(OC_MODULE_ID index);
```

#### 6.3.4.9.2. La fonction PLIB\_OC\_Disable

La fonction **PLIB\_OC\_Disable** désactive le module OC.

```
void PLIB_OC_Disable(OC_MODULE_ID index);
```

☝ Il est recommandé de l'utiliser avant une séquence de reconfiguration.

## 6.3.5. EXEMPLE GÉNÉRATION D'UN SIGNAL PWM

Utilisation du timer 2 et de OC2 (signal PWMA\_HBridge du kit PIC32).

On souhaite générer un signal PWM d'une fréquence de 10 kHz. Pour cela, on effectuera la configuration ci-dessous.

### 6.3.5.1. CONFIGURATION DU TIMER 2

Le timer 2 a été configuré pour une période de 100 µs soit 10 kHz.

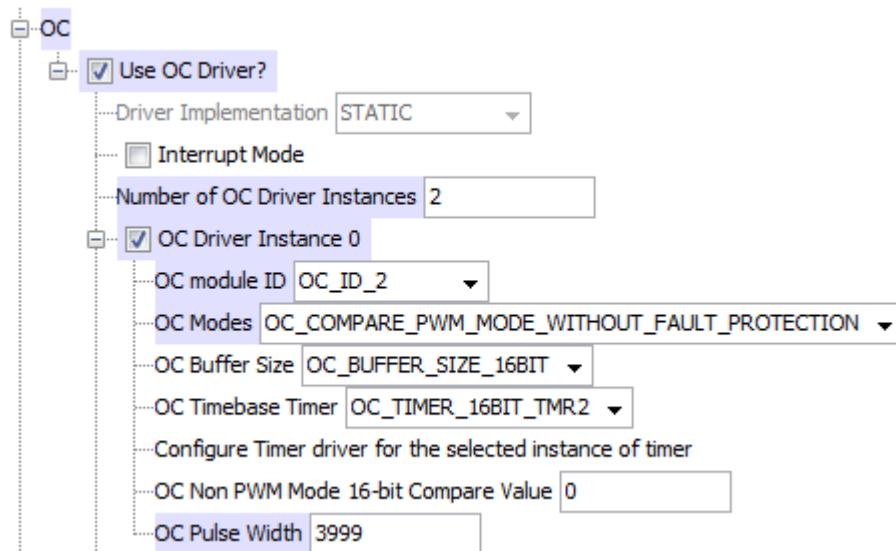
Un prescaler de 1 convient, la valeur de la période vaut (avec PB\_CLOCK = 80 MHz) :  
 $100 \mu\text{s} * 80 = 8'000$ . Donc valeur maximale de comptage  $8'000-1 = 7'999$ .

### 6.3.5.2. CONFIGURATION DU OC2

Pour un signal PWM, on utilise le mode **OC\_COMPARE\_PWM\_MODE\_WITHOUT\_FAULT\_PROTECTION**. Sélection du mode 16 bits. Il faut indiquer le timer 2 comme source pour la comparaison. Pour obtenir par défaut un PWM à 50% on fournit la moitié de la période du timer 2 au champ OC pulse width.

### 6.3.5.2.1. Configuration OC2 au niveau du MHC.

Voici la configuration réalisée pour l'OC2 en mode OC\_COMPARE\_PWM\_MODE\_WITHOUT\_FAULT\_PROTECTION.



### 6.3.5.2.2. Fonction d'initialisation de l'OC2

```
void DRV_OC0_Initialize(void)
{
    PLIB_OC_Disable(OC_ID_2); // ajout manuel
    /* Setup OC0 Instance */
    PLIB_OC_ModeSelect(OC_ID_2,
        OC_COMPARE_PWM_MODE_WITHOUT_FAULT_PROTECTION);
    PLIB_OC_BufferSizeSelect(OC_ID_2,
        OC_BUFFER_SIZE_16BIT);
    PLIB_OC_TimerSelect(OC_ID_2, OC_TIMER_16BIT_TMR2);
    PLIB_OC_Buffer16BitSet(OC_ID_2, 0);
    PLIB_OC_PulseWidth16BitSet(OC_ID_2, 3999); // 50%
}
```

### 6.3.5.3. ACTIVATION DE L'OC2

Même si on n'effectue pas l'ajout recommandé du Disable avant la configuration, il est nécessaire d'activer l'OC. Pour cela, on dispose de 2 fonctions qui appellent la fonction PLIB\_OC\_Enable().

```
void DRV_OC0_Enable(void)
{
    PLIB_OC_Enable(OC_ID_2);
}

void DRV_OC0_Start(void)
{
    PLIB_OC_Enable(OC_ID_2);
}
```

### 6.3.5.4. MODULATION DU SIGNAL PWM

Il suffit d'utiliser la fonction **PLIB\_OC\_PulseWidth16BitSet** en lui fournissant une valeur de comparaison correspondant à la valeur du duty cycle voulu.

Dans notre exemple, le 50% correspond à une valeur de 3'999 (4'000 cycles de comptage, donc 50 µs). On fournira donc une valeur variant de 0 à 7'999 à la fonction.

Exemple de modulation à 15%

```
PLIB_OC_PulseWidth16BitSet(OC_ID_2, (8000 * 0.15)-1);
```

¶ Pour éviter de prendre directement la valeur numérique de la période du timer, il est possible d'utiliser la fonction **PLIB\_TMR\_Period16BitGet** ou la fonction fournie par le driver du timer.

```
uint32_t DRV_TMR1_PeriodValueGet(void)
{
    /* Get 16-bit counter value*/
    return (uint32_t) PLIB_TMR_Period16BitGet(TMR_ID_2);
}
```

### 6.3.6. EXEMPLE GÉNÉRATION D'UNE IMPULSION

Utilisation du timer 3 configuré pour une période de 10 ms et de OC3 (signal PWMB\_HBrige du kit PIC32), que l'on utilisera pour piloter un servomoteur de modélisme.

#### 6.3.6.1. CONFIGURATION DU TIMER 3

Configuration du timer 3 pour une période de 10 ms (sans interruption).

L'horloge avant division correspond au PB\_CLOCK de 80 MHz. Soit une période 0,0125 µs.

Pour obtenir une période de 10 ms, il faut compter  $10'000 * 80 = 800'000$ , ce qui est beaucoup trop grand pour un compteur 16 bits.

Il faut au minimum une division de  $800'000 / 65536 = 12,2$  ce qui nous conduit à utiliser un diviseur de 16 qui existe pour le timer 3.

Pour obtenir la période de 10 ms, le timer 3 devra donc être paramétré comme suit :  
 $N_{MAX} = (T_{VOULU} / T_{TIMER3}) - 1 = (T_{VOULU} * f_{TIMER3}) - 1 = (10'000 * (80 / 16)) - 1 = 49'999$

### 6.3.6.2. CONFIGURATION DE OC3

On souhaite générer une impulsion variant de 0,8 ms à 2,2 ms, se répétant toutes les 10 ms. Ces valeurs correspondent typiquement à valeur de consigne pour un servomoteur de modélisme :

- Impulsion de 0,8 ms : positionnement à 0°
- Impulsion de 2,2 ms : positionnement à 360 °

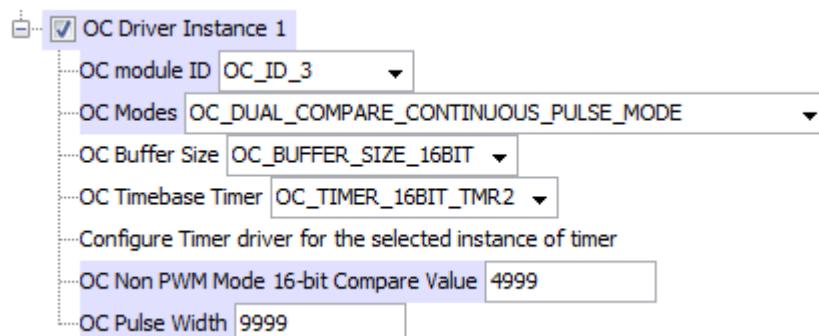
Pour la génération d'impulsions, on utilise le mode **OC\_DUAL\_COMPARE\_CONTINUOUS\_PULSE\_MODE**. Il faut indiquer le timer 3 comme source pour la comparaison.

☺ Dans ce mode, il est possible de décaler le flanc montant.

Pour vérifier le fonctionnement, on établit le flanc montant à 1 ms (4'999) et le flanc descendant à 2 ms (9'999). Le résultat doit être une impulsion d'une durée de 1 ms, se répétant à 100 Hz (rapport cyclique de 10%).

#### 6.3.6.2.1. Configuration OC3 au niveau du MHC

Voici la configuration au niveau du MHC :



#### 6.3.6.2.2. Fonction d'initialisation de l'OC3

```
void DRV_OC1_Initialize(void)
{
    PLIB_OC_Disable(OC_ID_3); // ajout manuel
    /* Setup OC1 Instance */
    PLIB_OC_ModeSelect(OC_ID_3,
                       OC_DUAL_COMPARE_CONTINUOUS_PULSE_MODE);
    PLIB_OC_BufferSizeSelect(OC_ID_3,
                           OC_BUFFER_SIZE_16BIT);
    PLIB_OC_TimerSelect(OC_ID_3, OC_TIMER_16BIT_TMR3);

    // Set Buffer (Primary compare) Value
    PLIB_OC_Buffer16BitSet(OC_ID_3, 4999); // ↑ à 1 ms
    // Set Pulse Width (Secondary compare) Value
    PLIB_OC_PulseWidth16BitSet(OC_ID_3, 9999); // ↓ à 2 ms
}
```

Avec la période du timer 3 à 50'000 cycles pour 10 ms (comptage de 0 à 49'999), une impulsion de 1,0 ms décalée de 1 ms est réalisée par Buffer = (50'000 / 10)-1 = 4'999 et Pulse Width = (2 \* 5'000)-1 = 9'999.

### 6.3.6.3. MODULATION DE LA LARGEUR D'IMPULSION

On utilise une entrée de l'AD comme valeur de consigne de largeur d'impulsion.

Il suffit d'utiliser la fonction **PLIB\_OC\_PulseWidth16BitSet** en lui donnant une valeur en unité du timer 3 correspondant à des largeurs d'impulsion de 0,8 ms à 2,2 ms.

En utilisant la plage du convertisseur AD et sachant que la valeur 5'000 représente 1 ms, on obtient :

```
ValPulseOC3 = (5000 * 0.8) -1 +  
              (AdcRes.Chan1 * 5000 * (2.2 - 0.8) / 1023);  
PLIB_OC_PulseWidth16BitSet(OC_ID_3, ValPulseOC3);
```

Remarque : Ceci est valable avec **PLIB\_OC\_Buffer16BitSet(OC\_ID\_3, 0);**

### 6.3.6.4. DÉCALAGE DU FLANC MONTANT DE L'IMPULSION

Il suffit d'utiliser la fonction **PLIB\_OC\_Buffer16BitSet** en lui donnant une valeur en unité du timer 3 correspondant à la durée du décalage souhaité.

Pour un décalage de 1 ms il faut une valeur de 4'999. On aura :

```
PLIB_OC_Buffer16BitSet(OC_ID_3, 4999);
```

Remarque : Le décalage aura une influence sur la largeur de l'impulsion, puisque le flanc montant apparaît en premier et que le flanc descendant est lié à l'autre valeur de comparaison (fixée par **PLIB\_OC\_PulseWidth16BitSet**).

### 6.3.7. APPLICATION POUR CONTRÔLE DES RÉSULTATS

Dans une 1<sup>ère</sup> phase, l'application ne modifie pas la valeur du rapport cyclique des signaux PWM, elle effectue uniquement l'initialisation et met l'application en état d'attente.

```
case APP_STATE_INIT:
{
    // Start les Timer
    DRV_TMR0_Start();
    DRV_TMR1_Start();
    DRV_TMR2_Start();
    DRV_TMR3_Start();
    // Start les OC
    DRV_OC0_Start();
    DRV_OC1_Start();
    appData.state = APP_STATE_WAIT;
    break;
}
```

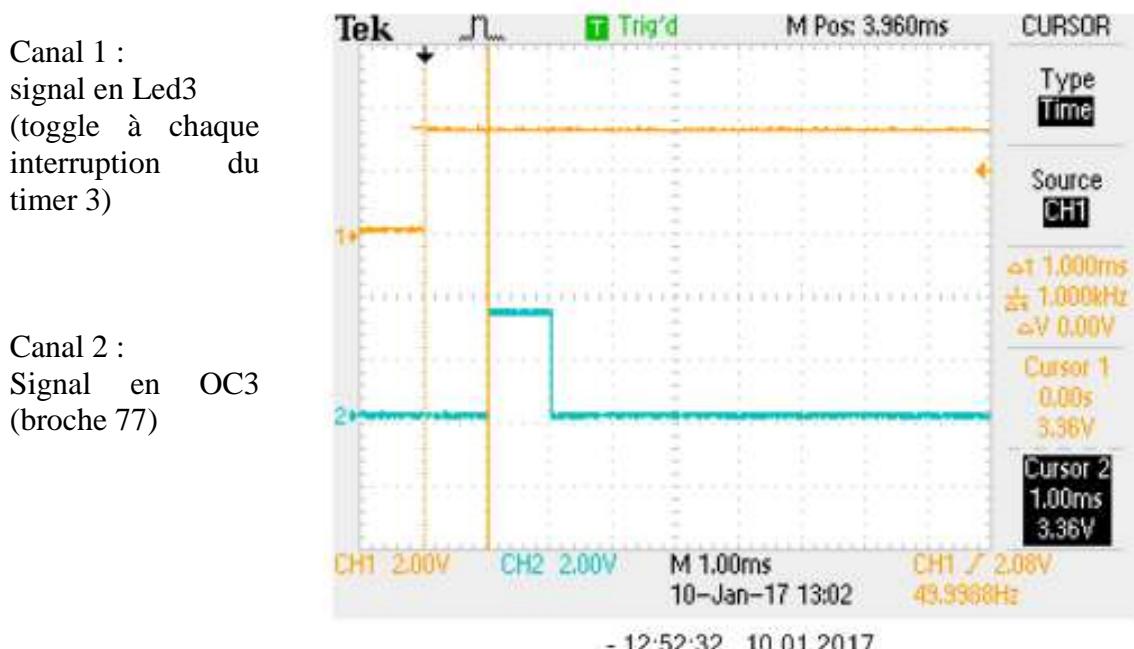
#### 6.3.7.1. OBSERVATION DES RESULTATS

Nous allons observer l'effet de :

```
PLIB_OC_Buffer16BitSet(OC_ID_3, 4999);
PLIB_OC_PulseWidth16BitSet(OC_ID_3, 9999);
```

Effectué dans la fonction **DRV\_OC1\_Initialize**.

Pour observer le décalage nous nous référons au toggle de la Led3 dans l'interruption du timer 3 (nécessaire d'activer l'interruption du timer 3) :



On observe bien le décalage de 1ms et une impulsion d'une durée de 1 ms.

### 6.3.8. APPLICATION POUR CONTRÔLE DES RÉSULTATS SUITE

Dans une 2<sup>ème</sup> phase, l'application modifie la valeur signaux PWM en utilisant les valeurs de l'AD.

```
case APP_STATE_SERVICE_TASKS:  
{  
    // Lecture des 2 pots  
    appData.AdcRes = BSP_ReadAllADC();  
    lcd_gotoxy(1, 3);  
    printf_lcd("Ch0 %4d Ch1 %4d ", appData.AdcRes.Chan0,  
              appData.AdcRes.Chan1);  
  
    // Modulation PWM OC2  
    appData.PulseWidthOC2 = (DRV_TMR1_PeriodValueGet() *  
                            appData.AdcRes.Chan0 / 1024);  
    PLIB_OC_PulseWidth16BitSet(OC_ID_2,  
                               appData.PulseWidthOC2);  
    // Modulation PWM OC3  
    // 1 ms correspond à 5000 cycles  
    // 0.8 ms => 3999 d'offset  
    // 2.2 - 0.8 = 1.4 => 7000  
    appData.PulseWidthOC3 = 3999 + ((7000 *  
                                    appData.AdcRes.Chan1) / 1023);  
    PLIB_OC_PulseWidth16BitSet(OC_ID_3,  
                               appData.PulseWidthOC3);  
  
    lcd_gotoxy(1, 4);  
    printf_lcd("OC2 %5d OC3 %5d", appData.PulseWidthOC2,  
              appData.PulseWidthOC3);  
    appData.state = APP_STATE_WAIT;  
    break;  
}
```

### 6.3.8.1. OBSERVATION DES RESULTATS

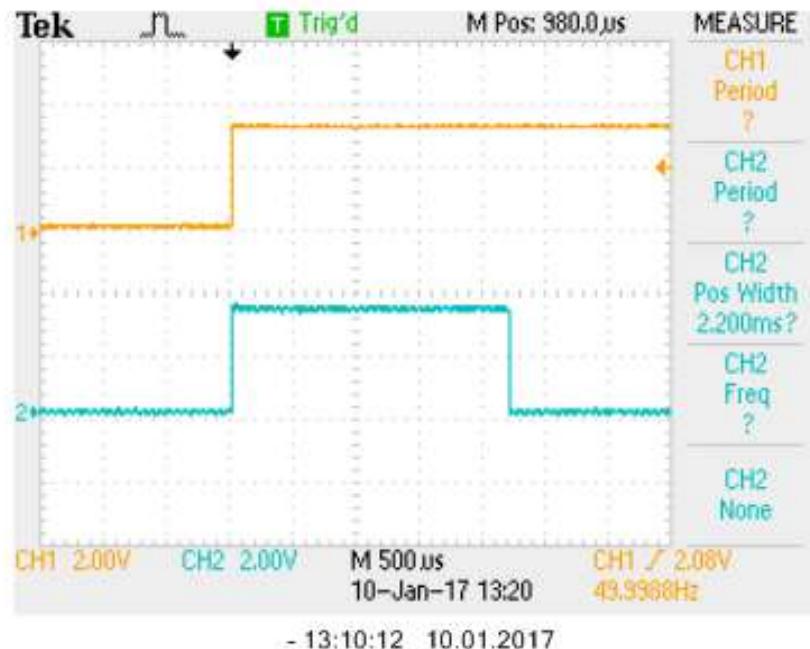
Canal 1 : signal en Led3 (toggle à chaque interruption du timer 3)

Canal 2 : signal en OC3 (broche 77)

Potentiomètre au minimum : ValOC3 = 3'999 => durée impulsion 0,8001 ms



Potentiomètre au maximum : ValOC3 = 10'999 => durée impulsion 2,200 ms



☺ On obtient bien la variation des largeurs d'impulsion comme prévu dans les 2 cas.  
Pour OC3, le décalage du flanc montant est supprimé par :

```
PLIB_OC_Buffer16BitSet(OC_ID_3, 0);
```

## 6.4. LES INPUTS CAPTURE DU PIC32MX

Les modules "input capture" (IC) sont utiles dans des applications demandant la mesure de périodes ou de largeurs d'impulsions.

Il est possible d'effectuer la capture de compteurs 16 bits ou 32 bits. On dispose uniquement du timer 2 et du timer 3. Ce qui permet 2 timers 16 bits ou un 32 bits. La capture a lieu lorsqu'un événement a lieu sur une des broches ICx. On dispose de IC1 à IC5.

Les événements de captures sont les suivants :

### 6.4.1. ÉVÉNEMENTS DE CAPTURE

#### 6.4.1.1. EVENEMENTS SIMPLES

- Capture lors du flanc montant sur l'entrée ICx
- Capture lors du flanc descendant sur l'entrée ICx

#### 6.4.1.2. EVENEMENTS DOUBLES

- Capture lors des deux flancs (montant et descendant) sur l'entrée ICx.
- Capture lors des deux flancs (montant et descendant) sur l'entrée ICx, en pouvant spécifier le 1<sup>er</sup> flanc.

#### 6.4.1.3. EVENEMENTS MULTIPLES

En utilisant le prescaler :

- Capture lors du 4<sup>ème</sup> flanc montant sur l'entrée ICx.
- Capture lors du 16<sup>ème</sup> flanc montant sur l'entrée ICx.

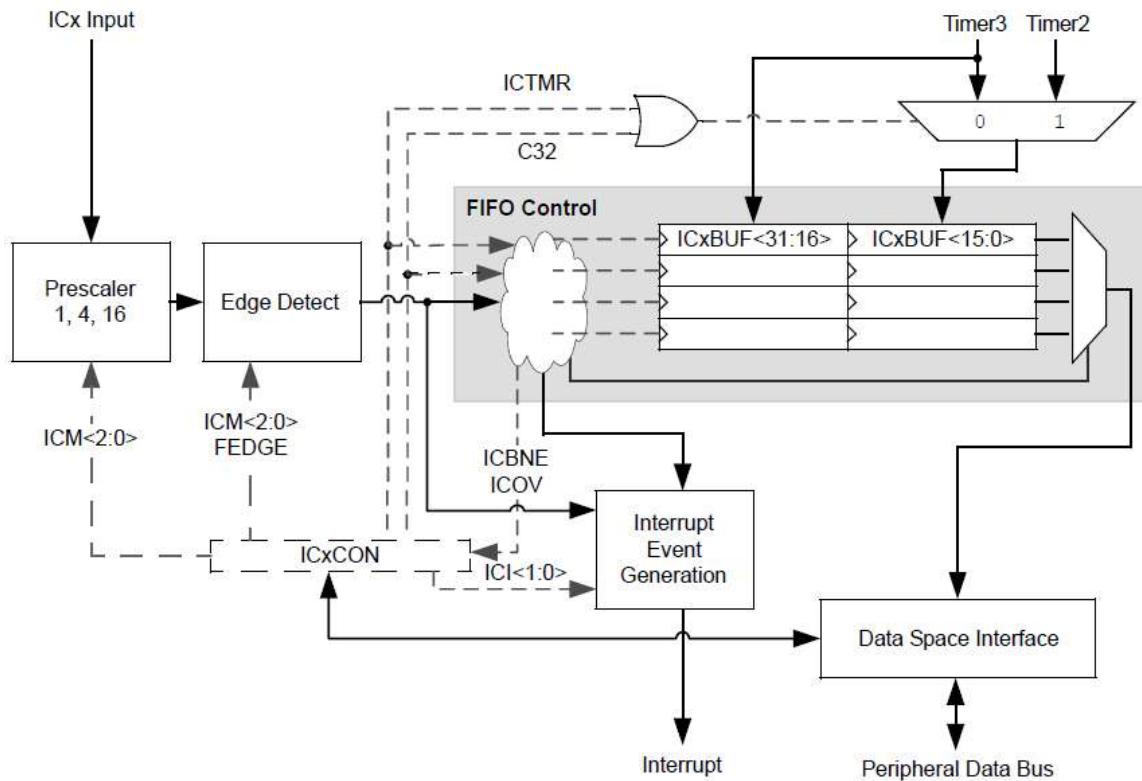
### 6.4.2. LISTE DES ENTRÉES DE CAPTURES

Pour le PIC32MX795FL512L 100 pin TQFP :

<b>ICx</b>	<b>Nom de la broche</b>	<b>No de la broche</b>
IC1	RTCC/EMDIO/AEMDIO/ <b>IC1</b> /RD8	<b>68</b>
IC2	_SS1/ <b>IC2</b> /RD9	<b>69</b>
IC3	SCK1/ <b>IC3</b> /PMCS2/PMA15/RD10	<b>70</b>
IC4	EMDC/AEMDC/ <b>IC4</b> /PMCS1/PMA14/RD11	<b>71</b>
IC5	ETXD2/ <b>IC5</b> /PMD12/RD12	<b>79</b>

### 6.4.3. SCHÉMA DE PRINCIPE MÉCANISME DE CAPTURE

Voici le schéma de principe du mécanisme de capture :



On constate que pour le stockage des valeurs capturées, on dispose d'un FIFO. Il sera nécessaire d'approfondir l'obtention des résultats dans le "Data Space Interface".

Remarque : Le diagramme montre qu'il y a un FIFO contenant 4 éléments pour chaque module de capture

### 6.4.4. PRINCIPE DE CONFIGURATION DE LA CAPTURE

Les étapes de configuration de la capture sont les suivantes :

- Disable de la capture (fonction **PLIB\_IC\_Disable**)
- Configuration en entrée de la broche ICx
- Sélection du mode (fonction **PLIB\_IC\_ModeSelect**)
- Sélection de la polarité du 1<sup>er</sup> flanc (fonction **PLIB\_IC\_FirstCaptureEdgeSelect**)
- Sélection du timer à capturer (fonction **PLIB\_IC\_TimerSelect**)
- Sélection capture 16 ou 32 bits (fonction **PLIB\_IC\_BufferSizeSelect**)
- Sélection du nombre d'événements de capture par interruption (fonction **PLIB\_IC\_EventsPerInterruptSelect**)
- Configuration de l'interruption de capture, mais sans autoriser l'interruption.

#### 6.4.4.1. SEQUENCE DE LANCEMENT DE LA CAPTURE

Pour assurer un bon démarrage de la capture, il est recommandé d'effectuer la séquence de lancement suivante dans la phase de démarrage de l'application :

- Autorisation de la capture (fonction **PLIB\_IC\_Enable** ou fonction **DRV\_ICx\_Start()**)
- Sélection de la polarité du 1<sup>er</sup> flanc (fonction **PLIB\_IC\_FirstCaptureEdgeSelect**)
- Vidange du tampon de capture s'il n'est pas vide (fonction **PLIB\_IC\_BufferIsEmpty** et fonctions **PLIB\_IC\_Buffer16BitGet** ou **PLIB\_IC\_Buffer32BitGet**).
- Mise à zéro du flag d'interruption
- Autorisation de l'interruption.

#### 6.4.5. FONCTIONS DE CONFIGURATION DE LA CAPTURE

Ces fonctions sont fournies par le fichier header **plib\_ic.h**. Toutes les fonctions ont le préfixe **PLIB\_IC\_** et elles utilisent le type énuméré **IC\_MODULE\_ID**.

##### 6.4.5.1. LE TYPE ENUMERE IC\_MODULE\_ID

Le type énuméré **IC\_MODULE\_ID** permet de sélectionner un des 5 modules de capture suivant dans le cas du PIC32MX :

```
typedef enum {
    IC_ID_1 = 0,
    IC_ID_2,
    IC_ID_3,
    IC_ID_4,
    IC_ID_5,
    IC_NUMBER_OF_MODULES
} IC_MODULE_ID;
```

##### 6.4.5.2. LA FONCTION PLIB\_IC\_MODESELECT

La fonction **PLIB\_IC\_ModeSelect** permet de choisir le mode de fonctionnement de la capture.

```
void PLIB_IC_ModeSelect(IC_MODULE_ID index, IC_INPUT_CAPTURE_MODES modeSel);
```

##### 6.4.5.2.1. Le type énuméré IC\_INPUT\_CAPTURE\_MODES

Le type énuméré **IC\_INPUT\_CAPTURE\_MODES** définit les différents modes:

```
typedef enum {
    IC_INPUT_CAPTURE_DISABLE_MODE = 0,
    IC_INPUT_CAPTURE_EDGE_DETECT_MODE = 1,
    IC_INPUT_CAPTURE_FALLING_EDGE_MODE = 2,
    IC_INPUT_CAPTURE_RISING_EDGE_MODE = 3,
    IC_INPUT_CAPTURE_EVERY_4TH_EDGE_MODE = 4,
    IC_INPUT_CAPTURE_EVERY_16TH_EDGE_MODE = 5,
    IC_INPUT_CAPTURE_EVERY_EDGE_MODE = 6,
    IC_INPUT_CAPTURE_INTERRUPT_MODE = 7
} IC_INPUT_CAPTURE_MODES;
```

#### 6.4.5.2.2. Comportement de chacun des modes.

Le tableau ci-dessous décrit les différents modes de captures :

Members	Description
IC_INPUT_CAPTURE_INTERRUPT_MODE	Interrupt only mode: Rising edge on input triggers an interrupt. This mode is used only when device is in Sleep/Idle mode
IC_INPUT_CAPTURE_EVERY_EDGE_MODE	Every Edge Capture mode: The first edge of the input signal is specified via PLIB_IC_RisingEdgeCaptureSet() or PLIB_IC_FallingEdgeCaptureSet() routines. Subsequently, timer count value is captured on every rising and falling of the input signal
IC_INPUT_CAPTURE_EVERY_16TH_EDGE_MODE	Prescaled Capture mode: Timer count value is captured every 16th rising edge of input signal
IC_INPUT_CAPTURE_EVERY_4TH_EDGE_MODE	Prescaled Capture mode: Timer count value is captured every 4th rising edge of input signal
IC_INPUT_CAPTURE_RISING_EDGE_MODE	Rising Edge mode: Timer count value is captured on every rising edge of input signal
IC_INPUT_CAPTURE_FALLING_EDGE_MODE	Falling Edge mode: Timer count value is captured on every falling edge of input signal
IC_INPUT_CAPTURE_EDGE_DETECT_MODE	Edge Detect mode: Timer count value is captured on every rising and falling of the input signal. Interrupt control bits are ignored and an interrupt event is generated for every capture. Overflow status is not updated.
IC_INPUT_CAPTURE_DISABLE_MODE	Capture module is disabled. Input signal is ignored and no capture events or interrupts are generated

Avec le mode IC\_INPUT\_CAPTURE\_EVERY\_EDGE\_MODE, il y a la possibilité de spécifier quel est le 1<sup>er</sup> flanc qui déclenche la capture.

#### 6.4.5.3. LA FONCTION PLIB\_IC\_FIRSTCAPTUREEDGESELECT

La fonction **PLIB\_IC\_FirstCaptureEdgeSelect** permet d'indiquer quel flanc déclenche la 1<sup>ère</sup> capture.

```
void PLIB_IC_FirstCaptureEdgeSelect(IC_MODULE_ID index, IC_EDGE_TYPES edgeType);
```

Le type énuméré IC\_EDGE\_TYPES présente deux valeurs possibles :

```
typedef enum {
    IC_EDGE_FALLING = 0,
    IC_EDGE_RISING = 1
} IC_EDGE_TYPES;
```

#### 6.4.5.4. LA FONCTION PLIB\_IC\_TIMERSELECT

La fonction **PLIB\_IC\_TimerSelect** permet de choisir le timer 16 bits qui est capturé. En capture 16 bits, il y a le choix entre le timer 2 et le timer 3.

```
void PLIB_IC_TimerSelect(IC_MODULE_ID index, IC_TIMERS tmr);
```

Le type énuméré IC\_TIMERS présente ces deux valeurs :

```
typedef enum {
    IC_TIMER_TMR3 = 0,
    IC_TIMER_TMR2 = 1
} IC_TIMERS;
```

#### 6.4.5.5. LA FONCTION PLIB\_IC\_BUFFERSIZESELECT

La fonction **PLIB\_IC\_BufferSizeSelect** permet de choisir entre le mode de capture 16 bits ou 32 bits.

```
void PLIB_IC_BufferSizeSelect(IC_MODULE_ID index, IC_BUFFER_SIZE bufSize);
```

Le type énuméré **IC\_BUFFER\_SIZE** présente deux valeurs : 16 bits ou 32 bits.

```
typedef enum {
    IC_BUFFER_SIZE_16BIT = 0,
    IC_BUFFER_SIZE_32BIT = 1
} IC_BUFFER_SIZE;
```

⌚ Il faut être cohérent avec le choix effectué. Si on choisit 16 bits, on a le choix entre 2 timers et on devra utiliser la fonction de lecture 16 bits. Si on choisit 32 bits on utilisera la paire de timers 2 & 3 et on devra utiliser la fonction de lecture 32 bits

#### 6.4.5.6. LA FONCTION PLIB\_IC\_EVENTSPERINTERRUPTSELECT

La fonction **PLIB\_IC\_EventsPerInterruptSelect** permet de choisir la relation entre les événements de captures et les interruptions levées.

```
void PLIB_IC_EventsPerInterruptSelect(IC_MODULE_ID index, IC_EVENTS_PER_INTERRUPT event);
```

Le type énuméré **IC\_EVENTS\_PER\_INTERRUPT** permet un choix parmi 4 possibilités.

```
typedef enum {
    IC_INTERRUPT_ON_EVERY_CAPTURE_EVENT = 0,
    IC_INTERRUPT_ON_EVERY_2ND_CAPTURE_EVENT = 1,
    IC_INTERRUPT_ON_EVERY_3RD_CAPTURE_EVENT = 2,
    IC_INTERRUPT_ON_EVERY_4TH_CAPTURE_EVENT = 3
} IC_EVENTS_PER_INTERRUPT;
```

Si on choisit par exemple **IC\_INTERRUPT\_ON\_EVERY\_2ND\_CAPTURE\_EVENT**, cela implique que lors de l'interruption on disposera de deux valeurs dans le tampon, et qu'il faudra lire ces deux valeurs pour vider le tampon de capture.

#### 6.4.5.7. LA FONCTION PLIB\_IC\_DISABLE

La fonction **PLIB\_IC\_Disable** désactive le module IC spécifié.

```
void PLIB_IC_Disable(IC_MODULE_ID index);
```

Cette fonction agit sur le bit 15 du registre ICxCON. La description du rôle du bit nous fournit une indication supplémentaire sur l'action.

bit 15	<b>ON:</b> Input Capture Module Enable bit
	1 = Module enabled
	0 = Disable and reset module, disable clocks, disable interrupt generation and allow SFR modifications

On comprend mieux l'importance de commencer la configuration par un Disable.

#### 6.4.5.8. LA FONCTION PLIB\_IC\_ENABLE

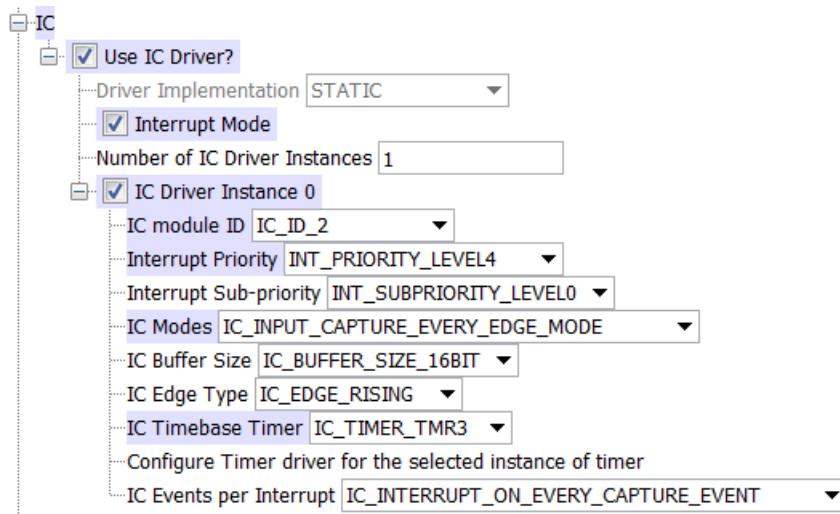
La fonction **PLIB\_IC\_Enable** active le module IC spécifié.

```
void PLIB_IC_Enable(IC_MODULE_ID index);
```

⌚ A n'exécuter qu'après avoir tout configuré.

#### 6.4.6. EXEMPLE DE CONFIGURATION DE LA CAPTURE

Voici la configuration réalisée au niveau du MHC :



Il faut compléter le code généré par 3 actions :

- Disable IC,
- Configuration broche ICx en entrée (cela n'est pas fait automatiquement),
- Mise en commentaire de l'autorisation de la source d'interruption.

```
void DRV_IC0_Initialize(void)
{
    // Ajout Disable et configuration IC2 en entrée.
    PLIB_IC_Disable(IC_ID_2);
    TRISDbits.TRISD9 = 1; // input

    /* Setup IC0 Instance */
    PLIB_IC_ModeSelect(IC_ID_2,
                        IC_INPUT_CAPTURE_EVERY_EDGE_MODE);
    PLIB_IC_FirstCaptureEdgeSelect(IC_ID_2,
                                   IC_EDGE_RISING);
    PLIB_IC_TimerSelect(IC_ID_2, IC_TIMER_TMR3);
    PLIB_IC_BufferSizeSelect(IC_ID_2,
                            IC_BUFFER_SIZE_16BIT);
    PLIB_IC_EventsPerInterruptSelect(IC_ID_2,
                                    IC_INTERRUPT_ON_EVERY_CAPTURE_EVENT);

    /* Setup Interrupt */
    // Source Enable à effectuer plus tard !
    // PLIB_INT_SourceEnable(INT_ID_0,
    //                      INT_SOURCE_INPUT_CAPTURE_2);
    PLIB_INT_VectorPrioritySet(INT_ID_0, INT_VECTOR_IC2,
                               INT_PRIORITY_LEVEL4);
    PLIB_INT_VectorSubPrioritySet(INT_ID_0, INT_VECTOR_IC2,
                                 INT_SUBPRIORITY_LEVEL0);
}
```

## 6.4.7. FONCTIONS D'UTILISATION DE LA CAPTURE

On dispose des fonctions suivantes pour la gestion de la lecture de la capture :

Name	Description
<code>PLIB_IC_Buffer16BitGet</code>	Obtains the 16-bit input capture buffer value.
<code>PLIB_IC_Buffer32BitGet</code>	Obtains the 32-bit input capture buffer value.
<code>PLIB_IC_BufferIsEmpty</code>	Checks whether the input capture buffer is empty.
<code>PLIB_IC_BufferOverflowHasOccurred</code>	Checks whether an input capture buffer overflow has occurred.

### 6.4.7.1. LECTURE RÉSULTAT CAPTURE : PLIB\_IC\_BUFFER16BITGET

La fonction `PLIB_IC_Buffer16BitGet` permet d'obtenir un élément du buffer de capture.

```
uint16_t PLIB_IC_Buffer16BitGet(IC_MODULE_ID index);
```

### 6.4.7.2. LECTURE RÉSULTAT CAPTURE : PLIB\_IC\_BUFFER32BITGET

La fonction `PLIB_IC_Buffer32BitGet` permet d'obtenir un élément du buffer de capture.

```
uint32_t PLIB_IC_Buffer32BitGet(IC_MODULE_ID index);
```

### 6.4.7.3. LA FONCTION PLIB\_IC\_BUFFERISEMPTY

La fonction `PLIB_IC_BufferIsEmpty` permet de savoir si le tampon de capture est vide ou non.

```
bool PLIB_IC_BufferIsEmpty(IC_MODULE_ID index);
```

### 6.4.7.4. LA FONCTION PLIB\_IC\_BUFFEROVERFLOWHASOCCURRED

La fonction `PLIB_IC_BufferOverflowHasOccurred` permet de savoir si le tampon de capture a débordé.

```
bool PLIB_IC_BufferOverflowHasOccurred(IC_MODULE_ID index);
```

Une situation de débordement indique que des captures se produisent et que l'on ne vient pas les lire suffisamment rapidement.

#### **6.4.8. FONCTIONS FOURNIES PAR LE DRV\_ICX**

En plus de la fonction de configuration DRV\_IC0\_Initialize, on trouve les fonctions suivantes (avec le DRV\_IC0) :

##### **6.4.8.1. DRV\_IC0\_START**

Utilise la fonction PLIB\_IC\_Enable.

```
void DRV_IC0_Start(void)
{
    PLIB_IC_Enable(IC_ID_2);
}
```

##### **6.4.8.2. DRV\_IC0\_STOP**

Utilise la fonction PLIB\_IC\_Disable.

```
void DRV_IC0_Stop(void)
{
    PLIB_IC_Disable(IC_ID_2);
}
```

##### **6.4.8.3. DRV\_IC0\_OPEN**

Prévue pour !?

```
void DRV_IC0_Open(void)
{
}
```

##### **6.4.8.4. DRV\_IC0\_CLOSE**

Prévue pour !?

```
void DRV_IC0_Close(void)
{
}
```

##### **6.4.8.5. DRV\_IC0\_CAPTURE32BITDATAREAD**

Permet de lire une capture 32 bits sans utiliser directement la fonction PLIB\_IC.

```
uint32_t DRV_IC0_Capture32BitDataRead(void)
{
    return PLIB_IC_Buffer32BitGet(IC_ID_2);
}
```

##### **6.4.8.6. DRV\_IC0\_CAPTURE16BITDATAREAD**

Permet de lire une capture 16 bits sans utiliser directement la fonction PLIB\_IC.

```
uint16_t DRV_IC0_Capture16BitDataRead(void)
{
    return PLIB_IC_Buffer16BitGet(IC_ID_2);
}
```

#### 6.4.8.7. DRV\_IC0\_BUFFERIsEmpty

Permet de savoir si le tampon est vide sans utiliser directement la fonction PLIB\_IC.

```
bool DRV_IC0_BufferIsEmpty(void)
{
    return PLIB_IC_BufferIsEmpty(IC_ID_2);
}
```

#### 6.4.9. LANCEMENT D'UN IC

Pour lancer l'IC, il faut établir la séquence d'action suivante :

- Enclenchement IC2.
- Purge du tampon de capture.
- Clear du flag d'interruption
- Autorisation de l'interruption de capture.

Pour rendre ceci plus systématique, nous exploitons la fonction vide du driver en la complétant comme ci-dessous :

```
void DRV_IC0_Open(void)
{
    PLIB_IC_Enable(IC_ID_2);
    while (!PLIB_IC_BufferIsEmpty(IC_ID_2))
    {
        PLIB_IC_Buffer16BitGet(IC_ID_2);
    }
    PLIB_INT_SourceFlagClear(INT_ID_0,
                             INT_SOURCE_INPUT_CAPTURE_2);
    PLIB_INT_SourceEnable(INT_ID_0,
                          INT_SOURCE_INPUT_CAPTURE_2);
}
```

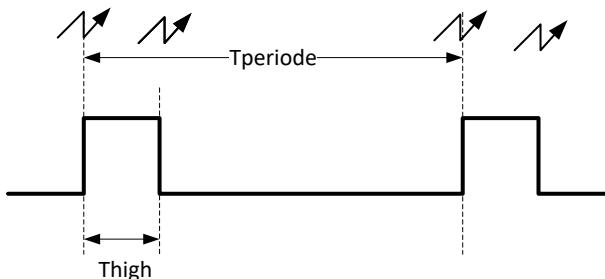
 Il est nécessaire d'effectuer un ajout dans le fichier drv\_ic\_static.h

```
// ****
// Section: Interface Headers for Instance 0 for the static
driver
// ****
// Ajout
void DRV_IC0_Open(void);

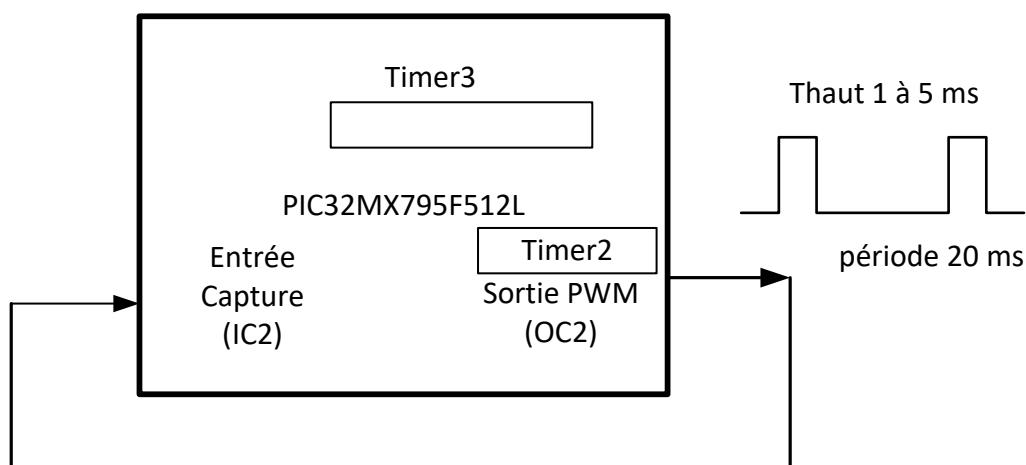
void DRV_IC0_Initialize(void);
```

#### 6.4.10. CAPTURE, EXEMPLE COMPLET

Dans cet exemple, nous allons configurer et utiliser le mécanisme de capture pour mesurer le Temps haut (Thigh) ainsi que la période d'un signal.



Le schéma ci-dessous illustre le mécanisme prévu dans cet exemple.

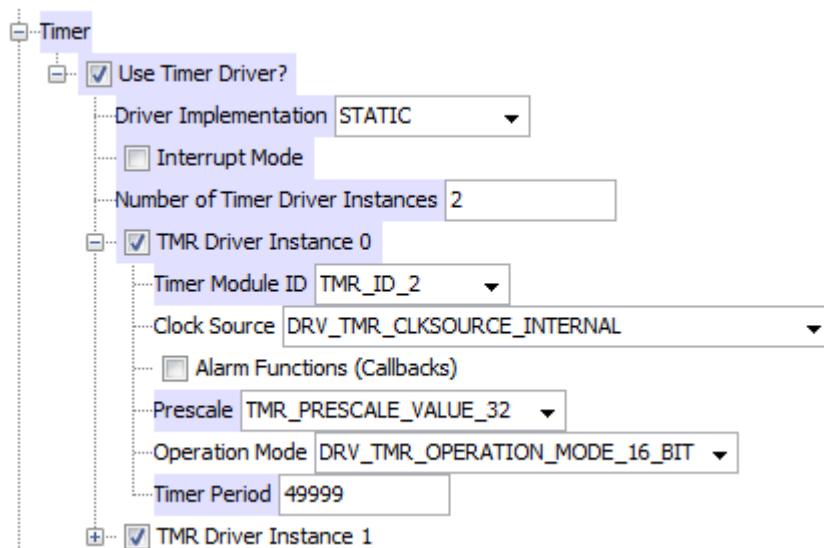


Le signal généré par OC2 (broche 76) est câblé sur IC2 (broche 69). Le timer 3 est le timer capturé.

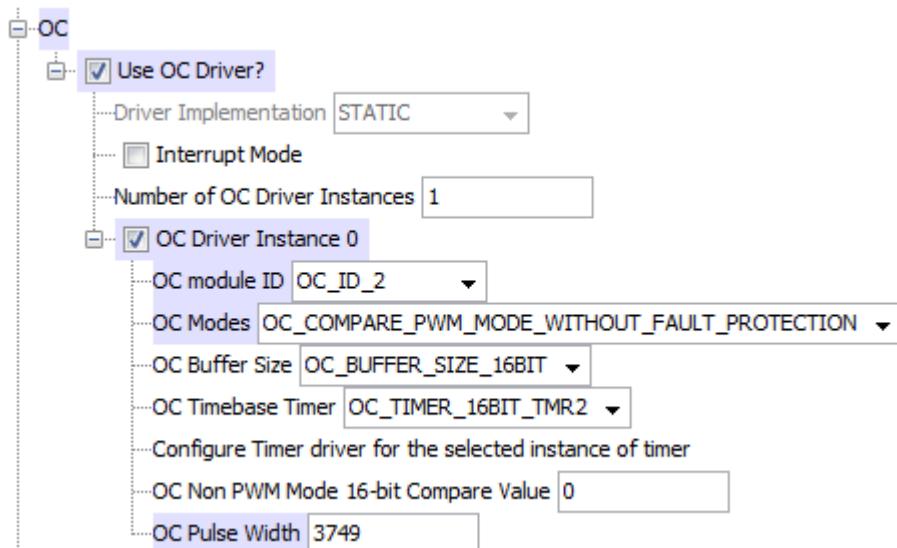
Le timer 2 est configuré pour une période de 20 ms. Le timer 3 travaille avec la période max soit 26.214 ms. Tous les 2 ont un prescaler de 32.

#### 6.4.10.1. CONFIGURATION DU TIMER 2 ET OC2

Le nombre de cycles du timer 2 pour 20 ms, avec un prescaler de 32 est de  $(20'000 * 80 / 32) = 50'000$ . Donc valeur maximale de comptage = 49'999.

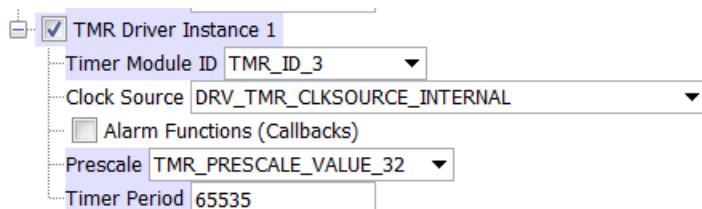


L'OC2 est configuré en PWM en générant une impulsion d'une largeur de 1,5 ms au niveau de la configuration. Valeur de OC\_Pulse\_Width de  $(50'000 / 20) * 1,5 = 3750$ .



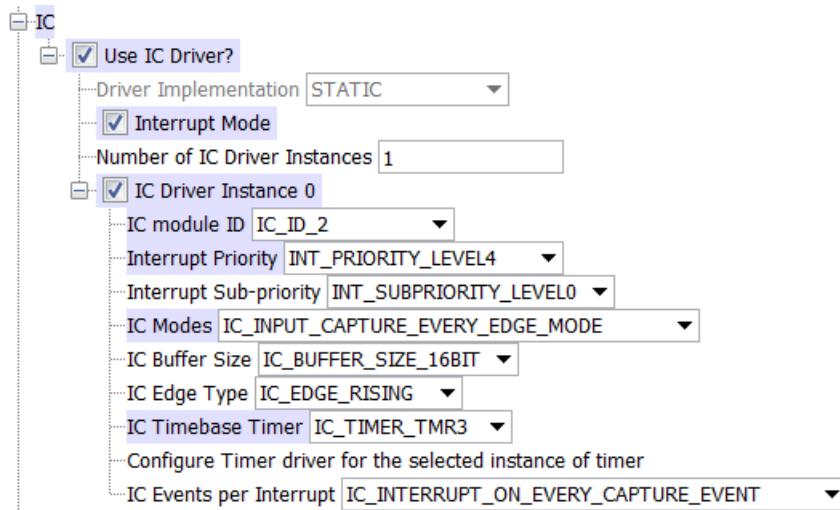
#### 6.4.10.2. CONFIGURATION DU TIMER 3

Etablissement d'un prescaler de 32 et attribution de 0xFFFF (65535) pour la période.



#### 6.4.10.3. CONFIGURATION DE LA CAPTURE (MHC)

La configuration de la capture est identique à l'exemple du paragraphe 6.4.6 avec l'IC2.



#### 6.4.10.4. INCLUDE POUR ACTION IC

Pour les différentes actions utilisant les fonctions de la plib\_ic, l'include suivant est nécessaire :

```
// Nécessaire pour les actions IC
#include "peripheral\ic\plib_ic.h"
```

#### 6.4.10.5. REPONSE A L'INTERRUPTION DE CAPTURE

La réponse à l'interruption de capture générée par le MHC ne comporte que la mise à 0 du flag d'interruption.

```
void __ISR(_INPUT_CAPTURE_2_VECTOR, ipl4AUTO)
           _IntHandlerDrvICInstance0(void)
{
    PLIB_INT_SourceFlagClear(INT_ID_0,
                           INT_SOURCE_INPUT_CAPTURE_2);
}
```

Voici ci-dessous la routine de réponse à l'interruption de capture 2 après complément. Nous allons y calculer la valeur de Thigh et de la période du signal sur la base des valeurs capturées.

Les principes sont les suivants :

- Les résultats sont mis à jour dans l'application en utilisant appData qui a été complété. Ils sont déposés dans des variables de type float afin d'exprimer les durées en ms.
- Le statut de l'application est établi à APP\_STATE\_SERVICE\_TASKS au moyen de la fonction APP\_UpdateState.
- Pour mesurer la période, il est nécessaire de mémoriser la valeur capturée d'un flanc montant à l'autre, d'où le besoin d'une variable static.
- Comme on a choisi une interruption à chaque capture, il est nécessaire de tester l'état de l'entrée de capture afin de déterminer s'il s'agit de l'interruption au flanc montant ou au flanc descendant.

```

void __ISR(_INPUT_CAPTURE_2_VECTOR, ipl4AUTO)
          _IntHandlerDrvICInstance0(void)
{
    uint16_t Capt2Falling;
    uint16_t Capt2Rising;
    uint16_t PeriodeTick, ThighTick;
    static uint16_t OldCaptRising;
    float PeriodeSignal;
    float Thigh ;

    // IC2 correspond à RD9
    if (PORTDbits.RD9 == 1) {
        LED0_W = 1; // flanc montant
        // Obtient capture du flanc montant
        Capt2Rising = PLIB_IC_Buffer16BitGet(IC_ID_2);
        // Calcul de la période
        PeriodeTick = Capt2Rising - OldCaptRising;
        // mise à jour memo capture au flanc montant
        OldCaptRising = Capt2Rising;
        PeriodeSignal = PeriodeTick * 0.4 / 1000 ; // en ms
        // 0.4 us par Tick
        // Fourni à l'application
        appData.Periode = PeriodeSignal;
        PLIB_INT_SourceFlagClear(INT_ID_0,
                                 INT_SOURCE_INPUT_CAPTURE_2);
        BSP_LEDToggle(BSP_LED_1);
    } else {
        LED0_W = 0; // flanc descendant
        // Obtient capture du flanc descendant
        Capt2Falling = PLIB_IC_Buffer16BitGet(IC_ID_2);
        // Calcul du Thigh
        ThighTick = Capt2Falling - OldCaptRising;
        Thigh = (ThighTick * 0.4) / 1000 ; // en ms
        // 0.4 us par Tick
        // Fourni à l'application
        appData.Thaut = Thigh;
        APP_UpdateState(APP_STATE_SERVICE_TASKS);
        PLIB_INT_SourceFlagClear(INT_ID_0,
                                 INT_SOURCE_INPUT_CAPTURE_2);
        BSP_LEDToggle(BSP_LED_1);
    }
}

```

**¶ Une interruption ne devrait jamais monopoliser le processeur plus longtemps que strictement nécessaire.**

Dans cet exemple, les calculs de durées sont effectués dans la routine d'interruption. Une bonne pratique serait de ne faire que de stocker les valeurs capturées dans l'interruption et réveiller l'application (APP\_UpdateState).

Les calculs de durées, gourmands en temps (il y a des floats !), peuvent être effectués dans l'application.

#### 6.4.10.6. VERIFICATION DU FONCTIONNEMENT DE L'INTERRUPTION DE CAPTURE

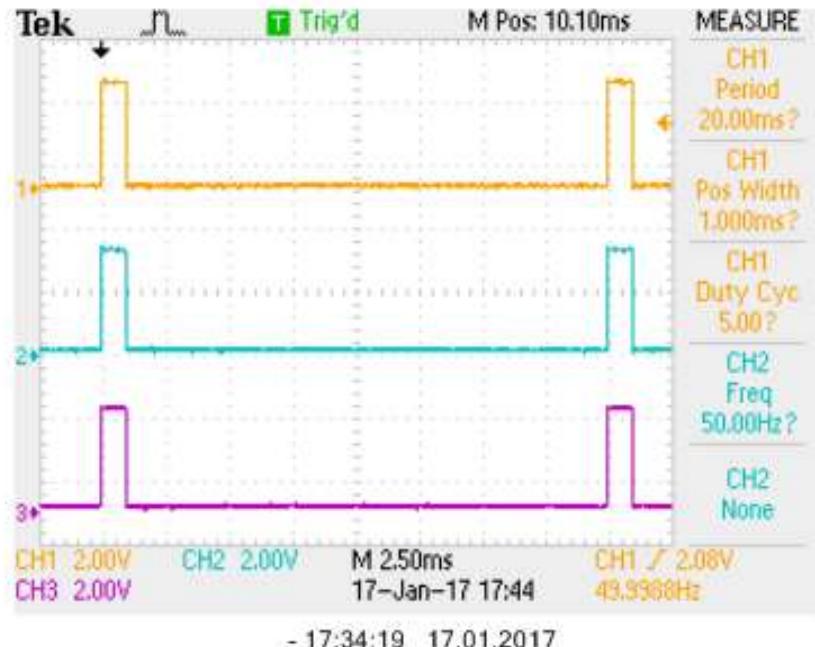
Les observations ci-dessous nous permettent de vérifier que l'on obtient bien une interruption à chaque capture.

✿ Il est important de vérifier cela car il suffit d'une erreur pour avoir l'interruption qui soit se produit tout le temps, soit ne se produit plus.

Canal 1 :  
signal en OC2  
(broche 76 câblée à  
69 IC2)

Canal 2 :  
Signal en Led1  
(toggle à chaque  
interruption)

Canal 3 :  
Signal en Led0  
(suit le flanc  
capture)

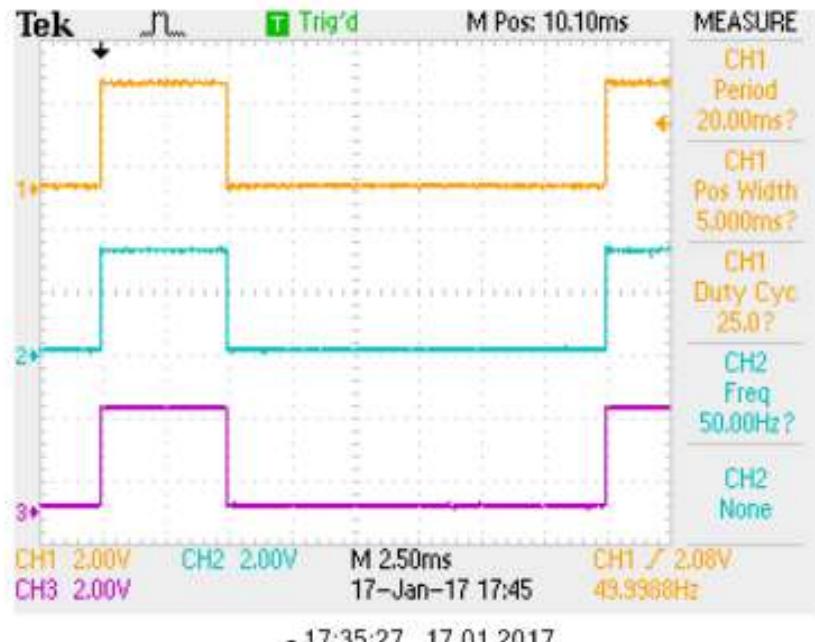


L'affichage sur le lcd nous fournit : Période = 20,000 ms et Thaut = 1,000 ms.

Canal 1 :  
signal en OC2  
(broche 76 câblée à  
69 IC2)

Canal 2 :  
Signal en Led1  
(toggle à chaque  
interruption)

Canal 3 :  
Signal en Led0  
(suit le flanc  
capture)



L'affichage sur le lcd nous fournit : Période = 20,000 ms et Thaut = 5,000 ms.

#### 6.4.10.7.LECTURE DU TAMPON DE CAPTURE

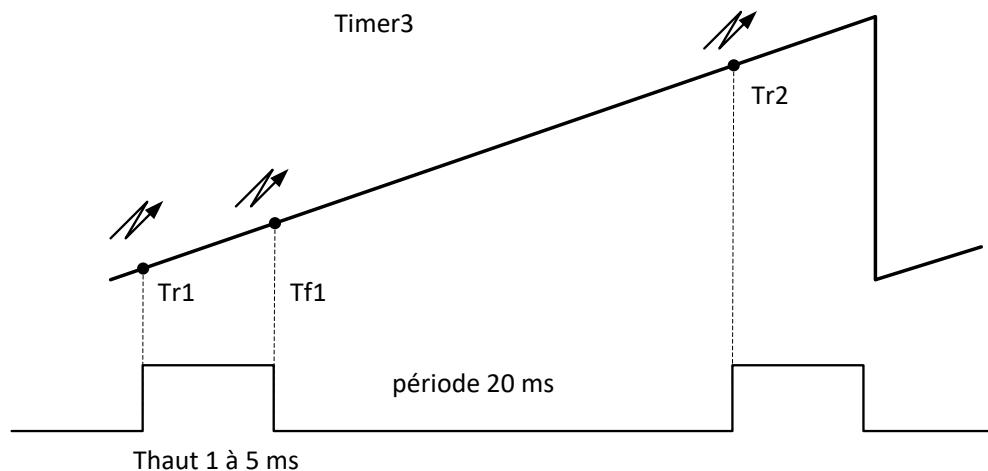
Comme on dispose d'une seule capture à chaque interruption il est possible de lire le FIFO de capture avec la fonction **PLIB\_IC\_Buffer16BitGet** sans se préoccuper de la situation du tampon de capture.

☞ Il ne faut appeler la fonction **PLIB\_INT\_SourceFlagClear** qu'après la lecture du FIFO de capture pour éviter une double interruption, car la condition d'interruption persiste tant que le FIFO n'est pas vide.

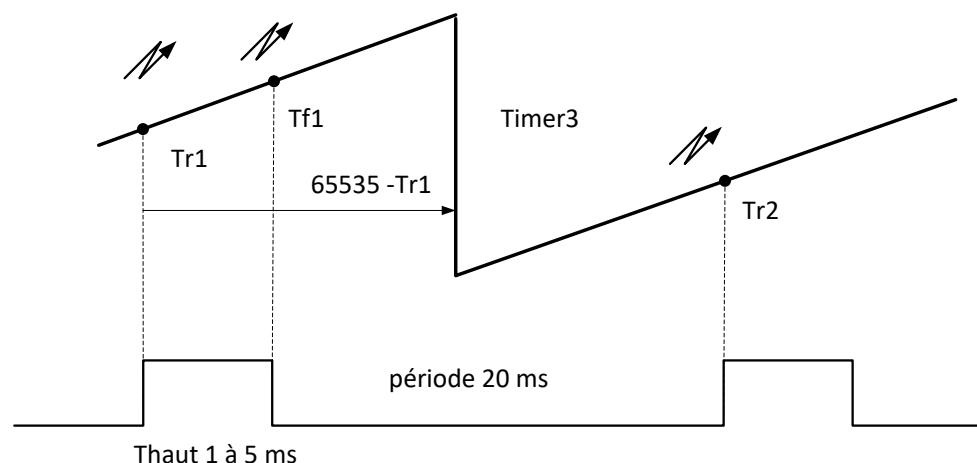
#### 6.4.10.8.GESTION DU REBOUCLEMENT DU TIMER 3

Le timer 3 est utilisé comme base de temps à une période supérieure à celle de timer 2 afin de pouvoir mesurer la période du signal généré par le timer 2 et OC2.

Le diagramme ci-dessous illustre la situation sans reboulement. La période du signal vaut  $Tr2 - Tr1$ .



Le diagramme ci-dessous illustre la situation de reboulement. Dans cette situation,  $Tr2 - Tr1$  donnerait une valeur négative selon toute logique. Toutefois, en déclarant soigneusement les variables du même type que les valeurs du timer, donc en 16 bits non signé (uint16\_t), le calcul s'effectue correctement.



La situation de reboulement du timer 3 peut aussi se produire entre  $Tr1$  et  $Tf1$ .

Si plusieurs reboulements ont lieu entre 2 captures, alors il faudrait gérer la situation.

#### 6.4.10.9. DÉTAIL CALCUL PÉRIODE ET THAUT

Le calcul de la période est basé sur la différence entre la valeur capturée au flanc montant avec l'ancienne valeur obtenue.

```
// Calcul de la periode
PeriodeTick = Capt2Rising - OldCaptRising;
// mise à jour memo capture au flanc montant
OldCaptRising = Capt2Rising;
PeriodeSignal = (PeriodeTick * 0.4) / 1000;
// en ms. 0.4 us par Tick
```

La période du signal en ms correspond à exprimer la période en  $\mu$ s, puis à diviser par 1'000. Avec un prescaler de 32, la durée du tick est de  $32 * 0,0125 = 0,4 \mu$ s. ( $f_{PBCLK} = 80$  MHz).

Le calcul du Thaut est basé sur la différence entre la valeur capturée au flanc descendant et la valeur mémorisée pour le flanc montant.

```
// Calcul du Thigh
ThighTick = Capt2Falling - OldCaptRising;
Thigh = ThighTick * 0.4 / 1000 ; // en ms
```

Même principe pour la transformation en ms.

#### 6.4.10.10. INCLUDE POUR ACTION OC AU NIVEAU APPLICATION

Pour les différentes actions utilisant les fonctions de la plib\_oc, les include suivants sont nécessaires :

```
#include "Mc32DriverAdc.h"
#include "Mc32DriverLcd.h"
// Nécessaire pour les actions OC
#include "peripheral\oc\plib_oc.h"
```

Remarque : l'utilisation de la fonction DRV\_IC0\_Open évite la nécessité d'inclure la plib\_ic

#### 6.4.10.11. ENCLENCHEMENT CAPTURE AU NIVEAU INIT DE L'APPLICATION

Dans la section case APP\_STATE\_INIT: de l'application on effectue la séquence d'actions suivante :

```
// Start les Timer
DRV_TMR0_Start();
DRV_TMR1_Start();
// Start l' OC
DRV_OC0_Start();
// Lancement IC
DRV_IC0_Open(); // fonction modifiée
```

La séquence d'action est la suivante (dans **DRV\_IC0\_Open**)

- Enclenchement IC2.
- Purge du tampon de capture.
- Clear du flag d'interruption.
- Autorisation de l'interruption de capture.

#### **6.4.10.12. DÉTAIL DES DATAS DE L'APPLICATION**

Voici dans app.h, le complément de la définition de la structure APP\_DATA :

```
typedef struct
{
    /* The application's current state */
    APP_STATES state;
    /* TODO: Define any additional data. */
    float Periode;
    float Thaut ;
    S_ADCResults AdcRes;
} APP_DATA;

// pour accéder au data dans system_interrupt.c
extern APP_DATA appData;
```

#### **6.4.10.13. CONTENU DE LA SECTION CASE APP\_STATE\_SERVICE\_TASKS**

Voici le contenu de la section case APP\_STATE\_SERVICE\_TASKS:

```
case APP_STATE_SERVICE_TASKS:
    // Lecture des 2 pots
    appData.AdcRes = BSP_ReadAllADC();
    // Modulation largeur impulsion 1 à 5 ms
    // Periode 20 ms ==> 50000 cycles
    // 1 ms correspond à 2500
    // 4 ms => 10000
    ValPulseOC2 = 2500-1 +
        ((10000 * (uint32_t) appData.AdcRes.Chan0) / 1023);
    PLIB_OC_PulseWidth16BitSet(OC_ID_2, ValPulseOC2);

    // Affichage des mesure de période et Thaut
    lcd_gotoxy(1,3);
    printf_lcd("Periode %6.3f [ms]", appData.Periode);
    lcd_gotoxy(1,4);
    printf_lcd("Thaut %6.3f [ms]", appData.Thaut);
    BSP_LEDToggle(BSP_LED_2);
    appData.state = APP_STATE_WAIT;
break;
```

## 6.5. CONCLUSION

Ce chapitre tente d'apporter à la fois des principes et des détails pratiques pour l'utilisation des timers, des modules OC (output compare) et des modules IC (input capture).

Au niveau des modules IC, il montre la gestion assez délicate de l'interruption de capture.

## 6.6. HISTORIQUE DES VERSIONS

### 6.6.1. V1.0 MARS 2014

Création du document en reprenant les chapitres 5 et 6 du cours laboratoire.

### 6.6.2. V1.1 MARS 2014

Mise au point de la partie capture et adaptation de la partie PWM à la situation PB\_FREQ = 80 MHz.

### 6.6.3. V1.5 JANVIER 2015

Adaptations à la nouvelle plib introduite par Harmony V1.00. Section IC incomplète

### 6.6.4. V1.6 JANVIER 2015

Compléments et corrections sections timers et OC, section IC complétée.

### 6.6.5. V1.7 JANVIER 2016

Adaptation à la version plib de Harmony V1.06.

### 6.6.6. V1.8 JANVIER 2017

Adaptation à la version plib de Harmony V1.08.

Pour IC modification de la fonction DRV\_IC0\_Open

### 6.6.7. V1.8.1 JANVIER 2017

Correction de quelques erreurs vues lors de la présentation.

### 6.6.8. V1.9 NOVEMBRE 2017

Reprise et relecture par SCA.

Retouché exemple et explications IC lors du reboulement.

### 6.6.9. V1.91 JANVIER 2019

Relecture.

Adaptation Harmony 2.05 : mode OC

OC\_COMPARE\_PWM\_EDGE\_ALIGNED\_MODE devenu obsolète, remplacé par  
OC\_COMPARE\_PWM\_MODE\_WITHOUT\_FAULT\_PROTECTION.

Correction valeurs maximales comptages dans exemples OC et IC (valeur maximale = nb cycles -1).

### 6.6.10. V1.92 DÉCEMBRE 2019

Clarifié calculs et formules exemples timers.

**MINF**  
**Programmation des PIC32MX**

# **Chapitre 7**

## **Gestion de la communication sérielle**



### **Théorie PIC32MX**

**Christian HUBER (CHR)**  
**Serge CASTOLDI (SCA)**  
**Version 1.92 septembre 2022**



## CONTENU DU CHAPITRE 7

<b>7. Gestion de la communication sérieelle</b>	<b>7-1</b>
<b>7.1. Principe de la liaison sérieelle</b>	<b>7-2</b>
7.1.1. Principe	7-2
7.1.2. Communication série asynchrone	7-3
7.1.2.1. Exemple de transmission asynchrone	7-4
7.1.2.2. Fonctionnement de la synchronisation	7-4
7.1.3. Communication série synchrone	7-5
<b>7.2. La liaison RS232</b>	<b>7-5</b>
7.2.1. Normes Electriques des communications série	7-5
7.2.1.1. Normes EIA (Electronic Industries Association)	7-6
7.2.1.2. Niveau logique norme V28	7-6
7.2.1.3. Niveau logique norme V11	7-6
7.2.2. Les signaux et la connectique	7-7
7.2.2.1. Connecteur 9 pôles du PC AT	7-7
7.2.2.2. Rôle des signaux	7-7
7.2.3. Câble Null Modem	7-8
7.2.4. Contrôle de flux hardware	7-9
7.2.5. Contrôle de flux software	7-9
7.2.6. Absence de contrôle de flux	7-9
<b>7.3. Les USART du PIC32MX</b>	<b>7-10</b>
7.3.1. Liste des USART à disposition (PIC32MX795F512L)	7-10
7.3.2. Schéma bloc de principe de l'USART	7-10
7.3.3. Schéma détaillé de la transmission	7-10
7.3.4. Schéma détaillé de la réception	7-12
7.3.5. Réalisation de la liaison RS232 sur le Kit	7-13
7.3.5.1. Câblage sur le PIC32MX795F512L	7-13
7.3.5.2. Adaptation pour les signaux RS232	7-13
7.3.6. Les registres des USART	7-14
7.3.6.1. Le registre UxMODE (UARTx Mode register)	7-14
7.3.6.2. Le registre UxSTA (UARTx Status and Control Register)	7-16
7.3.6.3. Le registre UxTXREG (UARTx Transmit Register)	7-18
7.3.6.4. Le registre UxRXREG (UARTx Receive Register)	7-18
7.3.6.5. Le registre UxBAUD (UARTx Baudrate Register)	7-18
7.3.7. Le générateur de baudrate	7-19
7.3.7.1. Exemple calcul du baudrate	7-20
<b>7.4. Configuration de l'USART avec la PLIB_USART</b>	<b>7-21</b>
7.4.1. Le type énuméré USART_MODULE_ID	7-21
7.4.2. La fonction PLIB_USART_BaudRateSet	7-21
7.4.3. La fonction PLIB_USART_HandshakeModeSelect	7-21
7.4.4. La fonction PLIB_USART_OperationModeSelect	7-22
7.4.5. La fonction PLIB_USART_LineControlModeSelect	7-22
7.4.5.1. Le type énuméré USART_LINECONTROL_MODE	7-23
7.4.5.2. USART_LINECONTROL_MODE, exemple	7-23
7.4.6. La fonction PLIB_USART_TransmitterEnable	7-23

7.4.7. PLIB_USART_TransmitterInterruptModeSelect _____	7-23
7.4.7.1. Le type énuméré USART_TRANSMIT_INTR_MODE _____	7-24
7.4.7.2. PLIB_USART_TransmitterInterruptModeSelect, exemple _____	7-24
7.4.8. La fonction PLIB_USART_ReceiverEnable _____	7-24
7.4.9. PLIB_USART_ReceiverInterruptModeSelect _____	7-24
7.4.9.1. Le type énuméré USART_RECEIVE_INTR_MODE _____	7-24
7.4.9.2. PLIB_USART_ReceiverInterruptModeSelect, exemple _____	7-24
7.4.10. Exemple de configuration _____	7-25
<b>7.5. Fonctions de l'émetteur</b>	<b>7-26</b>
7.5.1. La fonction PLIB_USART_TransmitterByteSend _____	7-26
7.5.2. La fonction PLIB_USART_TransmitterBufferIsFull _____	7-26
7.5.3. Exemple d'émission avec TransmitterByteSend _____	7-26
<b>7.6. Fonctions du récepteur</b>	<b>7-27</b>
7.6.1. La fonction PLIB_USART_ReceiverByteReceive _____	7-27
7.6.2. La fonction PLIB_USART_ReceiverDataIsAvailable _____	7-27
7.6.2.1. PLIB_USART_ReceiverDataIsAvailable, exemple _____	7-27
7.6.3. Fonction PLIB_USART_ReceiverOverrunErrorClear _____	7-27
<b>7.7. Fonctions générales de l'USART</b>	<b>7-28</b>
7.7.1. La fonction PLIB_USART_Disable _____	7-28
7.7.2. La fonction PLIB_USART_Enable _____	7-28
7.7.3. La fonction PLIB_USART_ErrorGet _____	7-29
7.7.4. Exemple gestion des erreurs _____	7-29
<b>7.8. Emission et réception sans interruption</b>	<b>7-30</b>
7.8.1. Configuration USART sans interruption _____	7-30
7.8.2. Emission en polling _____	7-30
7.8.3. Utilisation de la fonction d'émission et observations _____	7-31
7.8.3.1. Comportement de l'émission _____	7-31
7.8.3.2. Résultat émission avec Terminal _____	7-31
7.8.4. Réception en polling _____	7-33
7.8.4.1. Fonction de remplissage du FIFO _____	7-33
7.8.4.2. Appel de la fonction dans la réponse à l'interruption _____	7-33
7.8.4.3. La fonction APP_GetMessAD _____	7-34
7.8.4.4. Utilisation dans l'application _____	7-35
7.8.4.5. Exemple de résultat avec l'application _____	7-36
7.8.4.6. Comportement de la réception _____	7-36
7.8.4.7. Conclusion sur réception en polling _____	7-36
<b>7.9. Nécessité du recours à un FIFO</b>	<b>7-37</b>
7.9.1. Rôles du FIFO _____	7-37
7.9.2. Situation communication avec des FIFO _____	7-37
<b>7.10. Principe du FIFO</b>	<b>7-38</b>
7.10.1. Séparation des contextes _____	7-38
7.10.2. Principe de l'écriture dans le FIFO _____	7-38
7.10.3. Principe de la lecture dans le FIFO _____	7-40
<b>7.11. Analyse du fonctionnement d'un FIFO</b>	<b>7-41</b>
7.11.1. Situation de départ _____	7-41
7.11.2. Situation après écriture de 2 caractères _____	7-41
7.11.3. Situation après lecture des 2 caractères _____	7-41

7.11.4.	Situation de reboulement _____	7-42
7.11.5.	Situation FIFO plein _____	7-42
7.11.6.	FIFO plein conclusion _____	7-42
<b>7.12.</b>	<b>Réalisation FIFO avec des pointeurs _____</b>	<b>7-43</b>
7.12.1.	Structure décrivant un FIFO _____	7-43
7.12.2.	Déclaration d'un FIFO _____	7-43
7.12.3.	Initialisation d'un FIFO _____	7-43
7.12.4.	La fonction GetWriteSpace _____	7-44
7.12.5.	La fonction GetReadSize _____	7-44
7.12.6.	La fonction PutCharInFifo _____	7-45
7.12.7.	La fonction GetCharFromFifo _____	7-46
7.12.8.	Gestion des FIFO, librairie à disposition _____	7-46
<b>7.13.</b>	<b>Exemple utilisation des FIFOs et des interruptions _____</b>	<b>7-47</b>
7.13.1.	Configuration de l'USART _____	7-47
7.13.1.1.	Configuration d'un USART Driver avec Harmony _____	7-47
7.13.1.2.	Contenu de la fonction DRV_USART0_Initialize _____	7-48
7.13.2.	Contexte de réception _____	7-49
7.13.3.	Détails configuration interruption réception _____	7-49
7.13.3.1.	Choix d'un mode de déclenchement de Int RX _____	7-50
7.13.4.	Section réception de l'interruption de l'USART _____	7-51
7.13.5.	Principe traitement interruption de réception _____	7-52
7.13.6.	Comportement interruption RX avec HALF_FULL _____	7-53
7.13.7.	Comportement interruption RX avec 3B4FULL _____	7-54
7.13.8.	Comportement interruption RX avec ONE_CHAR _____	7-55
7.13.9.	Principe du traitement de la réception (GetMessage) _____	7-56
7.13.9.1.	Test disponibilité du message dans RxFifo _____	7-56
7.13.9.2.	Traitement du Contrôle de flux en réception _____	7-57
7.13.10.	Vérification contrôle de flux de réception _____	7-58
7.13.10.1.	Vue d'ensemble contrôle flux réception _____	7-58
7.13.10.1.	Vue de détail contrôle flux réception _____	7-58
7.13.11.	Contexte d'émission _____	7-59
7.13.12.	Détail de la configuration de l'interruption d'émission _____	7-59
7.13.13.	Réponse à l'interruption d'émission _____	7-61
7.13.14.	Principe de l'interruption d'émission _____	7-62
7.13.14.1.	Gestion du controle de flux dans interruption d'émission _____	7-63
7.13.15.	Principe du traitement de l'émission (SendMessage) _____	7-63
7.13.15.1.	Code partiel du mécanisme d'émission _____	7-64
7.13.16.	Observation de l'émission _____	7-64
7.13.16.1.	Vue d'ensemble _____	7-64
7.13.16.2.	Vue de détail _____	7-65
7.13.16.3.	Check des 6 caractères _____	7-65
7.13.17.	Vérification du contrôle de flux à l'émission _____	7-66
7.13.17.1.	Vue sans action du contrôle flux _____	7-66
7.13.17.2.	Vue d'ensemble contrôle flux d'émission _____	7-67
7.13.17.1.	Vue de détail du contrôle flux d'émission _____	7-67
7.13.17.2.	Vue de la reprise de l'émission _____	7-68
<b>7.14.</b>	<b>Structure d'un message _____</b>	<b>7-69</b>
7.14.1.	Données binaires ou ASCII _____	7-69
7.14.2.	Calcul d'un CRC 16 bits _____	7-69

<b>7.15. Principe du calcul d'un CRC</b>	<b>7-70</b>
7.15.1. Les polynômes utilisés pour le calcul du CRC	7-70
7.15.2. Variété d'algorithmes	7-71
7.15.3. Exemple de schéma logique	7-71
7.15.4. Exemple d'algorithme	7-72
7.15.4.1. Calcul du CRC bit à bit sur 1 byte	7-72
7.15.4.2. Calcul du CRC d'un message	7-72
7.15.4.3. Exemple de résultat	7-73
7.15.5. Calcul du CRC au moyen d'une table	7-74
7.15.5.1. Formule	7-74
7.15.5.2. Table calcul du CRC	7-74
7.15.5.3. Fonction pour calcul du CRC avec la table	7-75
7.15.5.1. Calcul du CRC d'un message	7-76
7.15.5.2. Exemple de résultat	7-76
7.15.6. Calcul du CRC au moyen de deux tables	7-76
7.15.6.1. Les 2 Tables de calcul du CRC	7-77
7.15.6.2. Sous-fonctions et fonctions	7-77
7.15.6.3. Calcul du CRC d'un message	7-78
7.15.6.4. Exemple de résultat	7-78
7.15.7. Exemple d'évolution du CRC sur 2 trames	7-79
7.15.8. Vérification du CRC à la réception	7-80
7.15.8.1. Algorithme de calcul du CRC lors de l'émission	7-80
7.15.8.2. Contrôle du CRC à la réception	7-80
7.15.8.3. Vérification du résultat	7-81
7.15.9. Calcul Crc16, librairie à disposition	7-81
<b>7.16. Exemple complet avec émission et réception</b>	<b>7-82</b>
7.16.1. Structure du message de l'exemple	7-82
7.16.2. Cycles de traitements	7-82
7.16.3. Définitions et éléments globaux	7-83
7.16.4. La fonction SendMessage	7-84
7.16.5. La fonction GetMessage	7-85
7.16.6. Réaction du système à des messages corrompus	7-87
7.16.6.1. Vue générale réaction à mauvais message	7-87
7.16.6.2. Vue réaction à mauvais message	7-87
7.16.6.3. Observation attente STX	7-88
7.16.6.1. Observation attente STX détail	7-88
7.16.6.2. Conclusion sur le traitement de mauvais message	7-88
7.16.7. Conclusion sur l'exemple	7-89
<b>7.17. Conclusion générale</b>	<b>7-89</b>
<b>7.18. Historique des Versions</b>	<b>7-90</b>
7.18.1. Version 1.5 janvier 2015	7-90
7.18.2. Version 1.6 mars 2015	7-90
7.18.3. Version 1.7 mars 2016	7-90
7.18.4. Version 1.7_1 mars 2016	7-90
7.18.5. Version 1.8 février 2017	7-90
7.18.1. Version 1.9 janvier 2018	7-90
7.18.1. Version 1.91 janvier 2019	7-90
7.18.2. Version 1.92 septembre 2022	7-90

## 7. GESTION DE LA COMMUNICATION SÉRIELLE

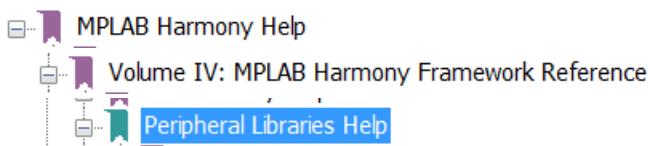
Ce chapitre traite de la communication serielle asynchrone en vue de réaliser une liaison RS232 avec une PIC32MX. La problématique de l'usage d'un FIFO est aussi abordée, ainsi que la sécurisation des trames par un CRC.

Les points suivants sont abordés :

- Principe de la liaison serielle
- La liaison RS232
- Les USART du PIC32MX
- Les fonctions pour la gestion de la communication
- Concept de FIFO
- Principe de la réception sous interruption
- Exemple d'application
- Validation des trames par CRC

Les documents de référence sont :

- La documentation "PIC32 Family Reference Manual" :  
Section 21 : UART
- Pour les détails spécifiques au PIC32MX795F512L, il faut se référer au document "PIC32MX5XX/6XX/7XX Family Data Sheet" :  
Section 20 : Universal Asynchronous Receiver Transmitter (UART)
- La documentation d'Harmony, qui se trouve dans <Répertoire Harmony>\v<n>\doc :  
Section MPLAB Harmony Framework Reference > Peripheral Libraries Help,  
sous-section USART Peripheral Library



Ce document a été établi sur la base de Harmony v1.08, sauf contre-indication.

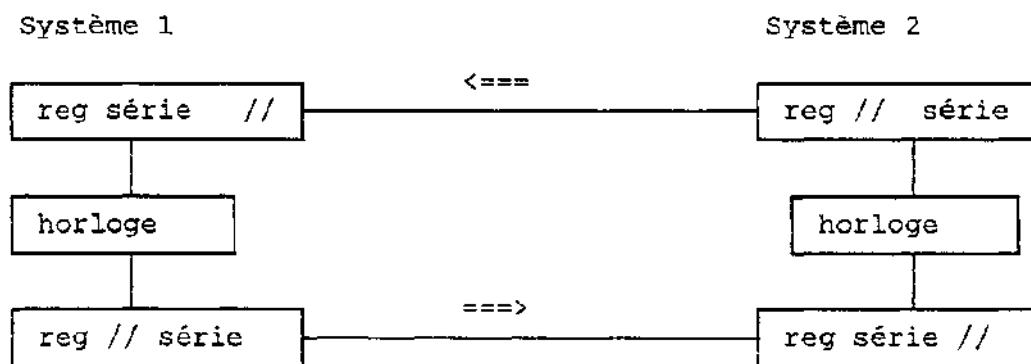
## 7.1. PRINCIPE DE LA LIAISON SÉRIELLE

Voici quelques rappels sur les concepts de la communication série.

### 7.1.1. PRINCIPE

Plutôt que de véhiculer l'information en parallèle, on transforme celle-ci en une information série à l'aide d'un registre à décalage. On effectue ainsi un multiplexage dans le temps de l'information à transmettre.

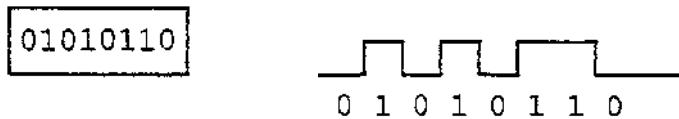
Cette méthode présente l'avantage de ne véhiculer des signaux électriques que sur une ligne par rapport aux 8 lignes d'un bus de 8 bits. Afin de pouvoir communiquer de façon bidirectionnelle entre deux systèmes, il est nécessaire d'utiliser deux fils : un fil sera utilisé pour communiquer du système 1 vers le système 2, l'autre fil sera utilisé pour communiquer du système 2 vers le système 1.



Le byte à transmettre est écrit dans le registre à décalage par le processeur.

Le registre à décalage, piloté par une horloge, sort les 8 bits du byte à transmettre les uns à la suite des autres à la fréquence d'un bit par période d'horloge.

Par exemple l'envoi de 0x56 correspond à (en commençant par le bit de poids fort) :



On constate que :

- A la réception, le registre à décalage ne sait pas quand le premier bit arrive sur la ligne.
- Si plusieurs bits identiques se suivent sur la ligne, le registre à décalage de la réception ne connaît pas ce nombre de bits identiques.
- Les bits sont émis sur la ligne à une certaine vitesse. Cette vitesse de transmission doit être identique dans les deux systèmes. Elle est définie en nombre de bits transmis par seconde, et est exprimée en bauds.

Il faut donc ajouter à ce schéma bloc un élément de synchronisation entre le récepteur et l'émetteur.

Cette synchronisation peut se faire de deux méthodes différentes.

### 7.1.2. COMMUNICATION SÉRIE ASYNCHRONE

Dans ce mode de transmission, on ajoute à la donnée devant être transmise une enveloppe de synchronisation, ce qui donne la structure suivante :

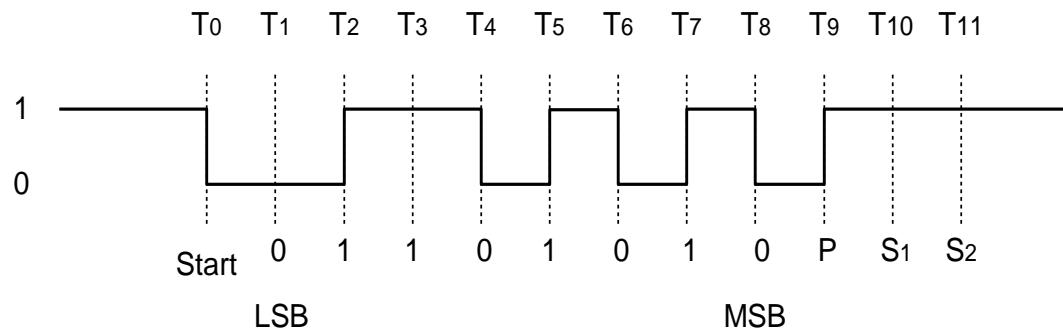
- Un bit de start de polarité inverse de l'état de repos de la ligne. Ce bit est utilisé pour synchroniser et pour indiquer au registre à décalage de réception qu'une nouvelle donnée est transmise sur la ligne.
- La donnée à transmettre, formée de 5, 6, 7 ou 8 bits.
- D'un bit de parité paire ou impaire (facultatif) utilisé pour contrôler la transmission.
- D'un ou de plusieurs bits de stop indiquant au récepteur la fin de la transmission de la donnée, ce qui permet à l'ordinateur de pouvoir lire la donnée.

### 7.1.2.1. EXEMPLE DE TRANSMISSION ASYNCHRONE

Paramètres de la communication :

- Vitesse 9'600 bauds.
- Nombre de bits de la donnée : 8.
- Parité : paire.
- Nombre de bits de stop : 2.

Transmission de 0x56 soit 0101'0110



- Avant le temps T<sub>0</sub>, la ligne est à l'état de repos (niveau 1).
- A la suite du bit de start, on a l'envoi des bits D<sub>0</sub> à D<sub>7</sub> (temps T<sub>1</sub> à T<sub>8</sub>).
- Au temps T<sub>9</sub>, on trouve le bit de parité paire, calculé sur les 8 bits de la donnée.
- Au temps T<sub>10</sub>, le premier bit de stop, et à T<sub>11</sub> le 2ème bit de stop puis fin de transmission de la donnée.
- Après le temps T<sub>11</sub>, la ligne peut soit rester au niveau de repos, soit émettre le bit de start de la prochaine donnée.

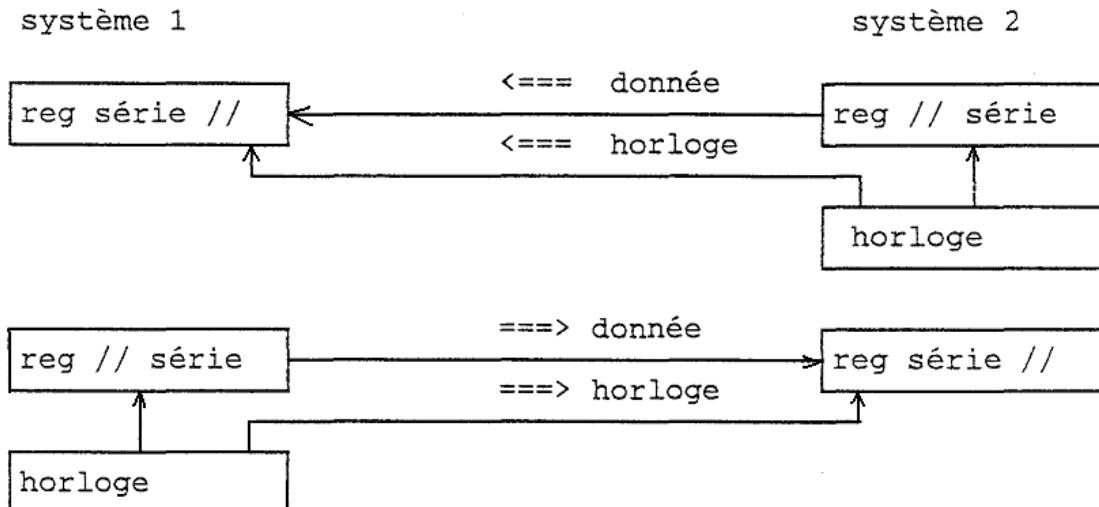
On constate que pour transmettre une donnée de 8 bits, il a fallu émettre 12 bits sur la ligne (start + 8 datas + parité + 2 stop). Le minimum est 10 bits sans parité et 1 seul stop.

### 7.1.2.2. FONCTIONNEMENT DE LA SYNCHRONISATION

Le récepteur possède une horloge qui pilote le registre à décalage. Cette horloge est resynchronisée par l'arrivée du bit de start. Le bit est mémorisé dans le registre à décalage à l'instant qui correspond à la moitié de la durée de ce bit.

### 7.1.3. COMMUNICATION SÉRIE SYNCHRONE

Dans ce mode de transmission, on ajoute une ligne supplémentaire transmettant une horloge dont la fréquence correspond à la vitesse de transmission. De ce fait, il n'est plus nécessaire de rajouter les bits de start et de stop utilisés en mode asynchrone. Le schéma bloc devient :

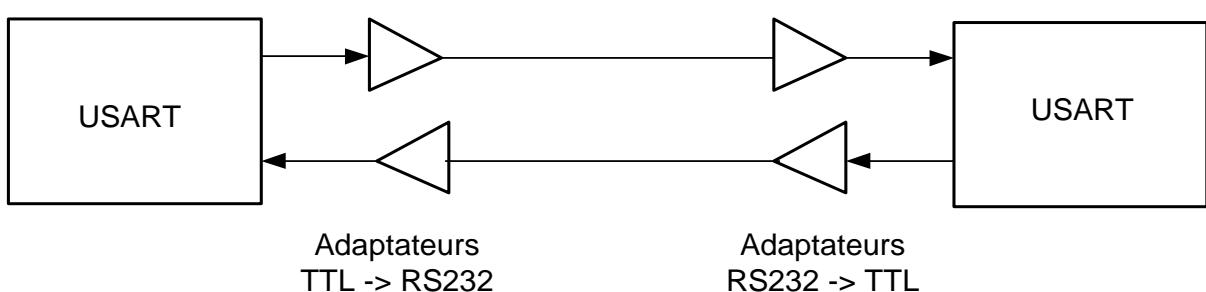


## 7.2. LA LIAISON RS232

Cette liaison serielle correspond à la norme EIA232 ou V28.

### 7.2.1. NORMES ELECTRIQUES DES COMMUNICATIONS SÉRIE

Le but des lignes de communication sérielles étant de pouvoir communiquer entre des systèmes informatiques distants, il est nécessaire de travailler avec des signaux électriques différents des signaux logiques de l'informatique, qui ne se prêtent pas à des liaisons sur de grandes distances.



Remarque : Au niveau de la ligne RS232, le '0' est représenté par le +12V et le '1' par le -12V.

De plus, il est nécessaire de normaliser ces lignes électriques afin que l'on puisse y brancher des systèmes de différents fournisseurs.

Il existe des normes fixant :

- Les niveaux électriques.
- La connectique.
- Le protocole de communication.

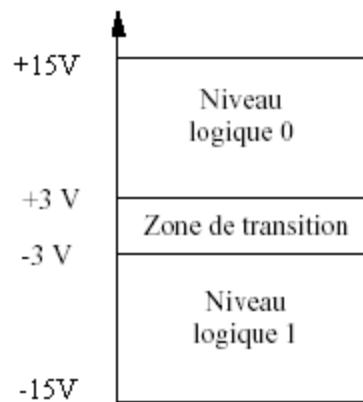
### 7.2.1.1. NORMES EIA (ELECTRONIC INDUSTRIES ASSOCIATION)

Les normes EIA définissent notamment les normes suivantes dont voici quelques caractéristiques :

Norme	RS232	RS423	RS422	RS485
Mode d'opération	Simple	Simple	Différentiel	Différentiel
Nombre d'émetteurs et de récepteurs	1 émetteur 1 récepteur	1 émetteur 10 récepteurs	1 émetteur 10 récepteurs	32 émetteurs 32 récepteurs
Longueur max (m)	15	1'200	1'200	1'200
Vitesse max (Bits/sec)	20k	100k	10M	10M
Signal de sortie min	$\pm 5V$	$\pm 3$	$\pm 2V$	$\pm 1,5V$
Signal de sortie max	$\pm 15V$	$\pm 6V$	$\pm 6V$	$\pm 6V$

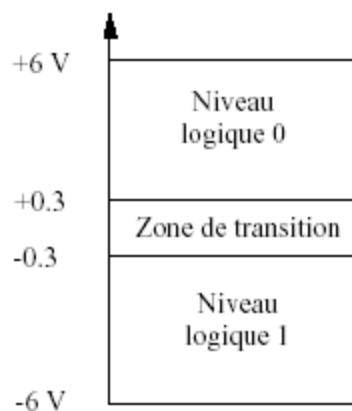
### 7.2.1.2. NIVEAU LOGIQUE NORME V28

La norme V28 correspond à la norme RS232 :



### 7.2.1.3. NIVEAU LOGIQUE NORME V11

La norme V11 correspond aux normes RS422 et RS485 :



## 7.2.2. LES SIGNAUX ET LA CONNECTIQUE

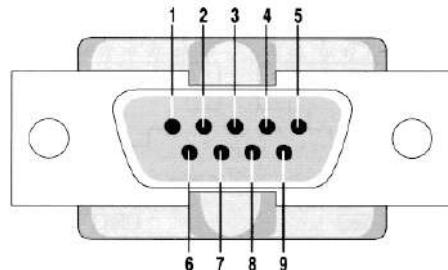
### 7.2.2.1. CONNECTEUR 9 PÔLES DU PC AT

Voici la représentation du connecteur série 9 pôles (port COM pour les OS Microsoft) comme on peut le voir sur un PC. Il s'agit d'un connecteur mâle de type SUB-D.

Connecteur SUB-D 9 pôles mâle



Numérotation (vue de devant)



### 7.2.2.2. RÔLE DES SIGNAUX

Broche	Signal	Description	E/S
1	DCD	Détection de porteuse	Entrée
2	RX	Réception de données	Entrée
3	TX	Emission de données	Sortie
4	DTR	Terminal de données prêt	Sortie
5	GND	Masse de signal	-
6	DSR	Données prêtes	Entrée
7	RTS	Requête d'émission	Sortie
8	CTS	Prêt pour l'émission	Entrée
9	RI	Indicateur d'appel	Entrée

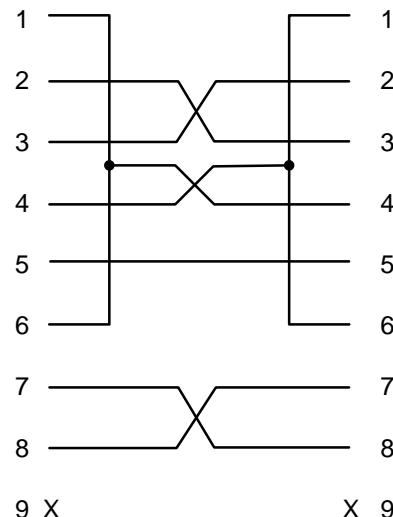
- **DCD** (Data Carrier Detect) : cette ligne est une entrée (active = logic '0', tension positive). Elle signale à l'ordinateur qu'une liaison a été établie avec un correspondant.
- **RX** (Receive Data) : cette ligne est une entrée. C'est ici que transitent les informations du correspondant vers l'ordinateur.
- **TX** (Transmit Data) : cette ligne est une sortie. Les données de l'ordinateur vers le correspondant sont véhiculées par son intermédiaire.
- **DTR** (Data Terminal Ready) : cette ligne est une sortie (active = logic '0', tension positive). Elle permet à l'ordinateur de signaler au correspondant que le port série a été libéré et qu'il peut être utilisé s'il le souhaite.
- **GND** (GrouND) : c'est la masse.
- **DSR** (Data Set Ready) : cette ligne est une entrée (active = logic '0', tension positive). Elle permet au correspondant de signaler qu'une donnée est prête.
- **RTS** (Request To Send) : cette ligne est une sortie (active = logic '0', tension positive). Elle indique au correspondant que l'ordinateur veut lui transmettre des données.
- **CTS** (Clear To Send) : cette ligne est une entrée (active = logic '0', tension positive). Elle indique à l'ordinateur que le correspondant est prêt à recevoir des données.

- **RI** (Ring Indicator) : cette ligne est une entrée (active = logic '0', tension positive). Elle permet à l'ordinateur de savoir qu'un correspondant veut initier une communication avec lui.

### 7.2.3. CÂBLE NULL MODEM

Lorsque l'on souhaite relier deux processeurs pour transmettre des données de l'un à l'autre, il est nécessaire d'utiliser un câble effectuant un certain nombre de croisements des lignes pour permettre le fonctionnement. Un type de câble communément utilisé est le câble dit "null modem" :

Câble destiné à relier directement 2 PC IBM AT par l'interface sérielle, le câble agissant comme modem zéro

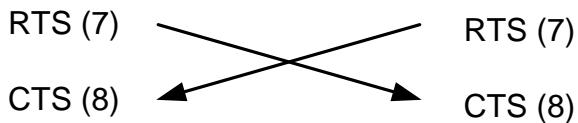


Il existe différents types de câbles croisés, tous permettant le transfert de données simple (lignes RX, TX et GND) sans contrôle de flux matériel. Cependant, plusieurs principes de contrôle de flux hardware existent, d'où des différences de câblages entre les lignes restantes. Le câble null modem ci-dessus est un type de câble simple et répandu.

### 7.2.4. CONTRÔLE DE FLUX HARDWARE

Une partie des signaux servent au contrôle du flux de la communication.

Un des mécanismes de base est le couple des signaux RTS et CTS :



Dans ce type de contrôle de flux, chaque correspondant à sa sortie RTS câblée sur l'entrée CTS de l'autre et peut ainsi signaler s'il est prêt à recevoir des données.

En réception, si le système n'est plus capable de recevoir des données (tampon plein) alors la ligne RTS sera mise au niveau logique 1 (tension négative) pour demander à l'émetteur de stopper l'émission. Dès que le tampon a de nouveau une place suffisante, la ligne RTS est remise au niveau logique 0.

En émission, il est nécessaire de tester le niveau du CTS. Si CTS = 0 il est possible d'émettre; si CTS = 1, il faut stopper l'émission.

### 7.2.5. CONTRÔLE DE FLUX SOFTWARE

Deux caractères ASCII particuliers XON et XOFF (respectivement caractères ASCII n°17 et 19) sont utilisés pour un contrôle de flux par logiciel.

Lorsqu'un système reçoit le caractère XOFF il doit cesser d'émettre jusqu'à la réception du caractère XON.

C'est en général lorsque le tampon de réception est plein à 75% que le système envoie le caractère XOFF. Lorsque le tampon de réception n'est plus utilisé qu'a 25% alors le caractère XON est envoyé.

### 7.2.6. ABSENCE DE CONTRÔLE DE FLUX

Lorsqu'un système (par exemple une carte à microcontrôleur) n'est pas équipé des lignes pour le contrôle de flux hardware, il est important sur l'autre système, par exemple un PC, de configurer la communication sans gestion du contrôle de flux hardware. Ceci évite d'avoir un blocage de la communication.

## 7.3. LES USART DU PIC32MX

Les PIC32MX795F512L possèdent 6 USART. Ils peuvent être configurés pour une transmission asynchrone Full-Duplex (bidirectionnelle) ou pour une transmission synchrone en master ou en slave. En plus il est possible d'obtenir une gestion par le processeur des lignes CTS et RTS.

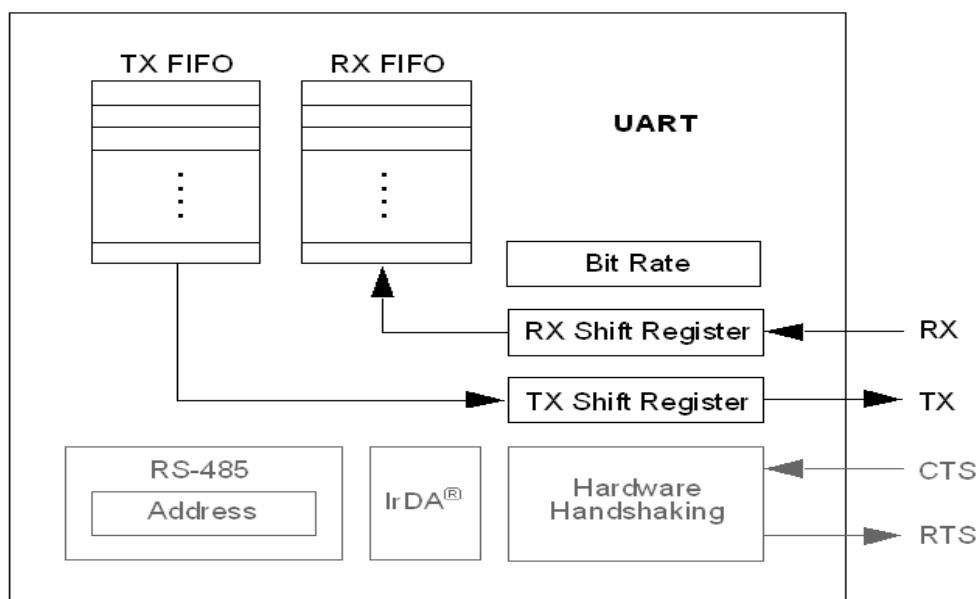
☝ Seule la transmission asynchrone nous intéresse dans le cadre de ce chapitre.

### 7.3.1. LISTE DES UART À DISPOSITION (PIC32MX795F512L)

On dispose de 6 UARTs. Les UARTS 1, 2 et 3 disposent de CTS et RTS.

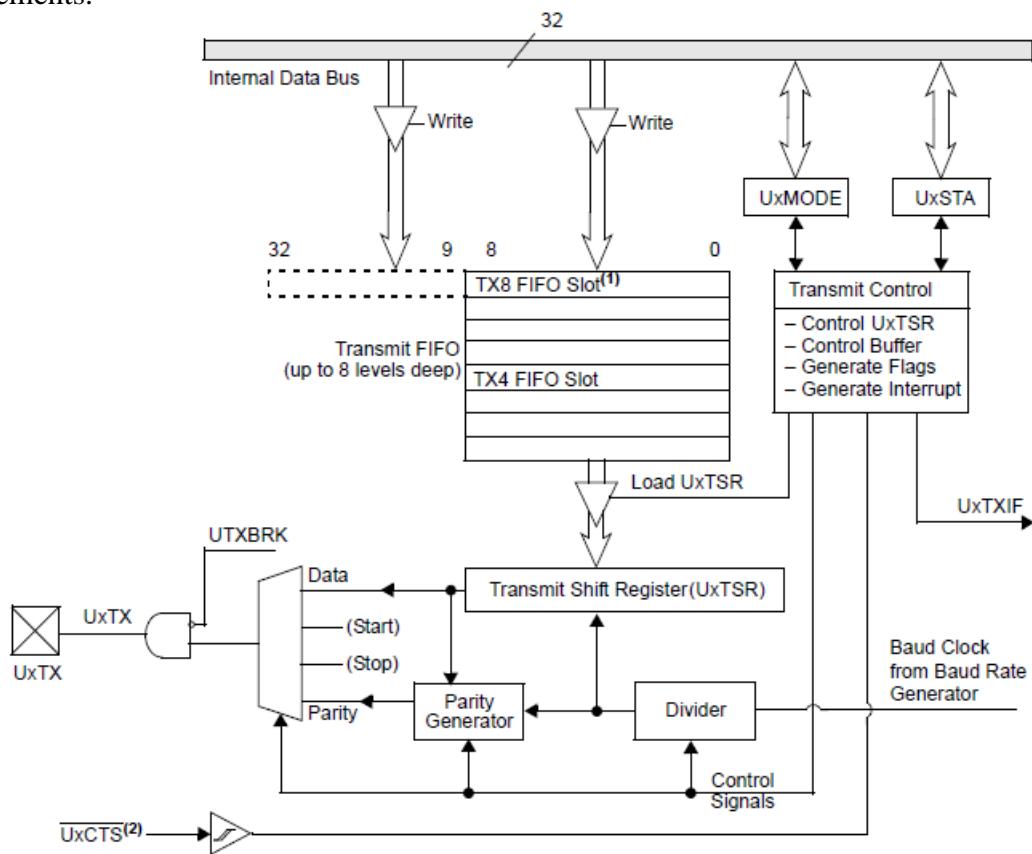
Pin Name	Pin Number <sup>(1)</sup>			Pin Type	Buffer Type	Description
	64-Pin QFN/TQFP	100-Pin TQFP	121-Pin XBGA			
U1CTS	43	47	L9	I	ST	UART1 clear to send
U1RTS	49	48	K9	O	—	UART1 ready to send
U1RX	50	52	K11	I	ST	UART1 receive
U1TX	51	53	J10	O	—	UART1 transmit
U3CTS	8	14	F3	I	ST	UART3 clear to send
U3RTS	4	10	E3	O	—	UART3 ready to send
U3RX	5	11	F4	I	ST	UART3 receive
U3TX	6	12	F2	O	—	UART3 transmit
U2CTS	21	40	K6	I	ST	UART2 clear to send
U2RTS	29	39	L6	O	—	UART2 ready to send
U2RX	31	49	L10	I	ST	UART2 receive
U2TX	32	50	L11	O	—	UART2 transmit
U4RX	43	47	L9	I	ST	UART4 receive
U4TX	49	48	K9	O	—	UART4 transmit
U6RX	8	14	F3	I	ST	UART6 receive
U6TX	4	10	E3	O	—	UART6 transmit
U5RX	21	40	K6	I	ST	UART5 receive
U5TX	29	39	L6	O	—	UART5 transmit

### 7.3.2. SCHÉMA BLOC DE PRINCIPE DE L'UART



### 7.3.3. SCHÉMA DÉTAILLÉ DE LA TRANSMISSION

Voici le schéma détaillé de la transmission. Le PIC32MX795F512L possède un FIFO de 8 éléments.



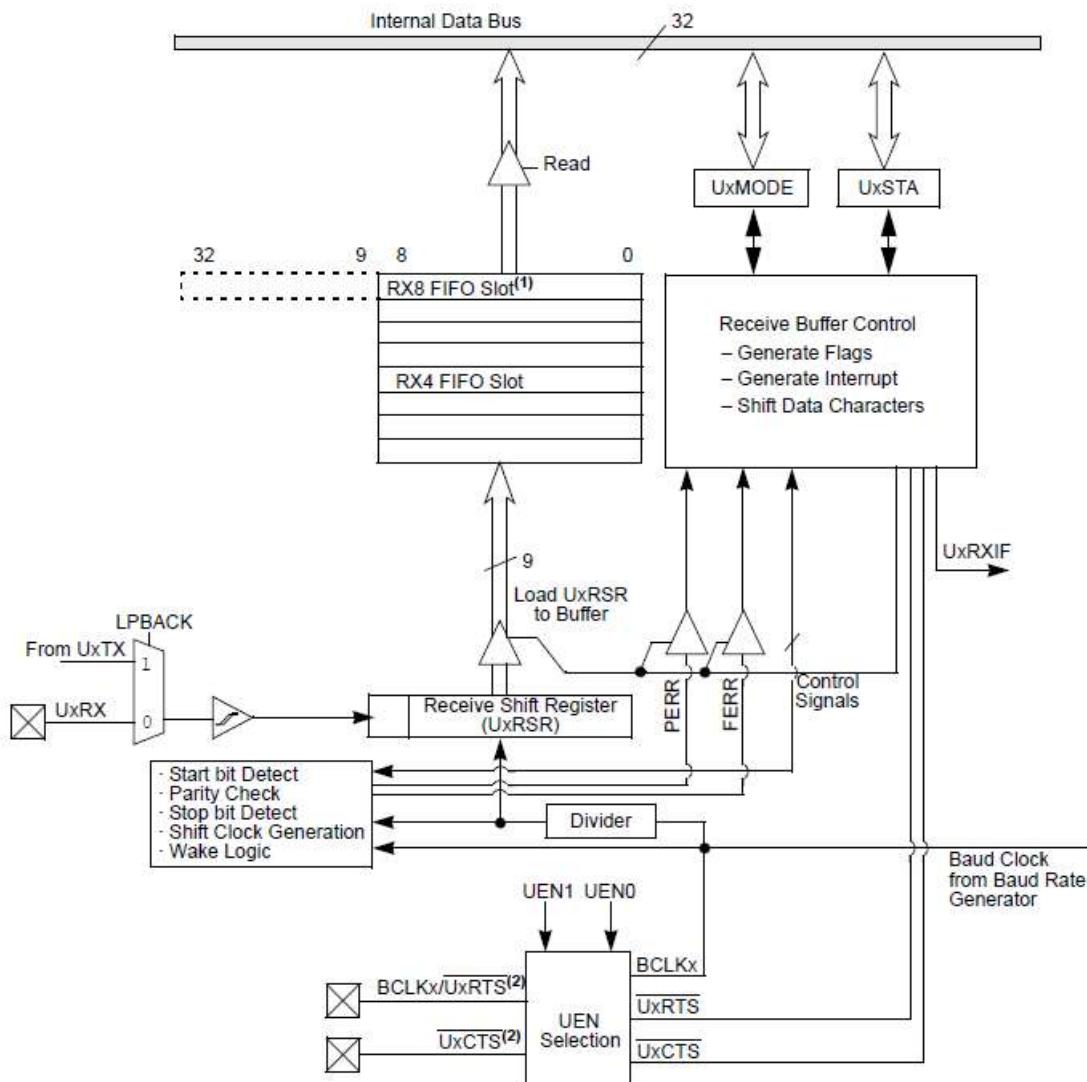
**Note 1:** Refer to the specific device data sheet for availability of 8-level-deep FIFO.

**2:** Refer to the specific device data sheet for availability of **UxCTS** pin.

On constate la présence de la broche **\_UxCTS** qui permet l'autorisation/blocage de l'émission par hardware.

### 7.3.4. SCHÉMA DÉTAILLÉ DE LA RÉCEPTION

Voici le schéma détaillé de la réception. Le PIC32MX795F512L possède un FIFO de 8 éléments.



Note 1: Refer to the specific device data sheet for availability of 8-level-deep FIFO.

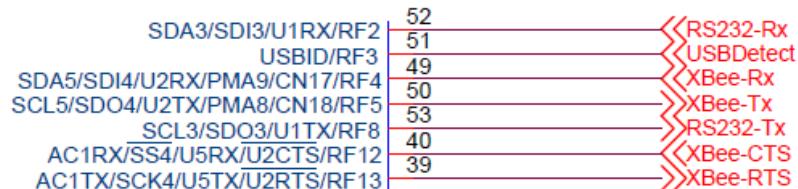
2: Refer to the specific device data sheet for availability of UxRTS and UxCTS pins.

On constate la présence des broches \_UxRTS et \_UxCTS qui permettent le handshaking hardware.

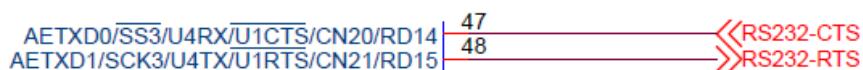
### 7.3.5. RÉALISATION DE LA LIAISON RS232 SUR LE KIT

#### 7.3.5.1. CÂBLAGE SUR LE PIC32MX795F512L

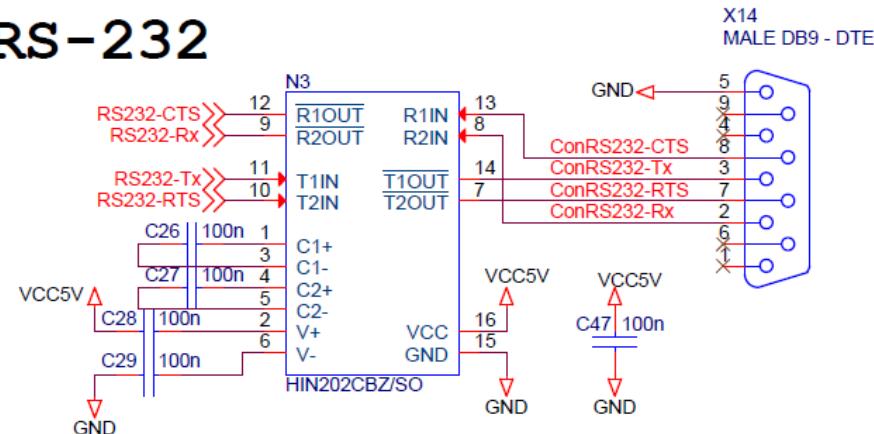
Comme on peut l'observer sur l'extrait du câblage du PIC32MX795F512L, c'est l'UART 1 qui est utilisé.



Les signaux CTS et RTS sont câblés sur les broches permettant un traitement automatique (par le PIC) du handshaking.



#### 7.3.5.2. ADAPTATION POUR LES SIGNAUX RS232



### 7.3.6. LES REGISTRES DES USART

Voici la vue d'ensemble des registres SFR en relation avec les USART :

Table 21-1: UART SFRs Summary

Name	Bit 31/23/15/7	Bit 30/22/14/6	Bit 29/21/13/5	Bit 28/20/12/4	Bit 27/19/11/3	Bit 26/18/10/2	Bit 25/17/9/1	Bit 24/16/8/0
UxMODE <sup>(1,2,3)</sup>	31:24	—	—	—	—	—	—	—
	23:16	—	—	—	—	—	—	—
	15:8	ON	FRZ	SIDL	IREN	RTSMD <sup>(4)</sup>	—	UEN<1:0> <sup>(4)</sup>
	7:0	WAKE	LPBACK	ABAUD	RXINV	BRGH	PDSEL<1:0>	STSEL
UxSTA <sup>(1,2,3)</sup>	31:24	—	—	—	—	—	—	ADM_EN
	23:16	ADDR<7:0>						
	15:8	UTXISEL<1:0>	UTXINV	URXEN	UTXBRK	UTXEN	UTXBF	TRMT
	7:0	URXISEL<1:0>	ADDEN	RIDLE	PERR	FERR	OERR	URXDA
UxTXREG	31:24	—	—	—	—	—	—	—
	23:16	—	—	—	—	—	—	—
	15:8	—	—	—	—	—	—	UTX<8>
	7:0	UTX<7:0>						
UxRXREG	31:24	—	—	—	—	—	—	—
	23:16	—	—	—	—	—	—	—
	15:8	—	—	—	—	—	—	RX<8>
	7:0	RX<7:0>						
UxBRG <sup>(1,2,3)</sup>	31:24	—	—	—	—	—	—	—
	23:16	—	—	—	—	—	—	—
	15:8	BRG<15:8>						
	7:0	BRG<7:0>						

#### 7.3.6.1. LE REGISTRE UXMODE (UARTx MODE REGISTER)

Ces registres permettent la configuration des USART.

Register 21-1: UxMODE: UARTx Mode Register<sup>(1,2,3)</sup>

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—	—
bit 31								bit 24

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—	—
bit 23								bit 16

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0
ON	FRZ	SIDL	IREN	RTSMD <sup>(4)</sup>	—	UEN<1:0> <sup>(4)</sup>	
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
WAKE	LPBACK	ABAUD	RXINV	BRGH	PDSEL<1:0>		STSEL
bit 7							bit 0

**Legend:**

R = Readable bit

W = Writable bit

P = Programmable bit

U = Unimplemented bit, read as '0'

-n = Bit Value at POR: ('0', '1', x = Unknown)

### 7.3.6.1.1. UxMode, rôle des bits

bit 31-16	<b>Unimplemented:</b> Read as '0'
bit 15	<b>ON:</b> UARTx Enable bit 1 = UARTx is enabled. UARTx pins are controlled by UARTx as defined by UEN<1:0> and UTXEN control bits 0 = UARTx is disabled. All UARTx pins are controlled by corresponding bits in the PORTx, TRISx and LATx registers; UARTx power consumption is minimal  <b>Note:</b> When using 1:1 PBCLK divisor, the user software should not read/write the peripheral SFRs in the SYSCLK cycle immediately following the instruction that clears the module's ON bit.
bit 14	<b>FRZ:</b> Freeze in Debug Exception Mode bit 1 = Freeze operation when CPU is in Debug Exception mode 0 = Continue operation when CPU is in Debug Exception mode  <b>Note:</b> FRZ is writable in Debug Exception mode only, it is forced to '0' in Normal mode.
bit 13	<b>SDL:</b> Stop in Idle Mode bit 1 = Discontinue operation when device enters Idle mode 0 = Continue operation in Idle mode
<b>Register 21-1: UxMODE: UARTx Mode Register<sup>(1,2,3)</sup> (Continued)</b>	
bit 12	<b>IREN:</b> IrDA Encoder and Decoder Enable bit 1 = IrDA is enabled 0 = IrDA is disabled
bit 11	<b>RTSMD:</b> Mode Selection for <u>UxRTS</u> Pin bit <sup>(4)</sup> 1 = <u>UxRTS</u> pin is in Simplex mode 0 = <u>UxRTS</u> pin is in Flow Control mode
bit 10	<b>Unimplemented:</b> Read as '0'
bit 9-8	<b>UEN&lt;1:0&gt;:</b> UARTx Enable bits <sup>(4)</sup> 11 = UxTX, UxRX and UxBCLK pins are enabled and used; <u>UxCTS</u> pin is controlled by corresponding bits in the PORTx register 10 = UxTX, UxRX, <u>UxCTS</u> and <u>UxRTS</u> pins are enabled and used 01 = UxTX, UxRX and <u>UxRTS</u> pins are enabled and used; <u>UxCTS</u> pin is controlled by corresponding bits in the PORTx register 00 = UxTX and UxRX pins are enabled and used; <u>UxCTS</u> and <u>UxRTS/UxBCLK</u> pins are controlled by corresponding bits in the PORTx register
bit 7	<b>WAKE:</b> Enable Wake-up on Start bit Detect During Sleep Mode bit 1 = Wake-up enabled 0 = Wake-up disabled
bit 6	<b>LPBACK:</b> UARTx Loopback Mode Select bit 1 = Loopback mode is enabled 0 = Loopback mode is disabled
bit 5	<b>ABAUD:</b> Auto-Baud Enable bit 1 = Enable baud rate measurement on the next character – requires reception of Sync character (0x55); cleared by hardware upon completion 0 = Baud rate measurement disabled or completed
bit 4	<b>RXINV:</b> Receive Polarity Inversion bit 1 = UxRX Idle state is '0' 0 = UxRX Idle state is '1'
bit 3	<b>BRGH:</b> High Baud Rate Enable bit 1 = High-Speed mode – 4x baud clock enabled 0 = Standard Speed mode – 16x baud clock enabled
bit 2-1	<b>PDSEL&lt;1:0&gt;:</b> Parity and Data Selection bits 11 = 9-bit data, no parity 10 = 8-bit data, odd parity 01 = 8-bit data, even parity 00 = 8-bit data, no parity
bit 0	<b>STSEL:</b> Stop Selection bit 1 = 2 Stop bits 0 = 1 Stop bit

### 7.3.6.2. LE REGISTRE UXSTA (UARTx STATUS AND CONTROL REGISTER)

Ces registres permettent la configuration des USART ainsi que l'obtention d'informations sur la situation.

**Register 21-2: UXSTA: UARTx Status and Control Register<sup>(1,2,3)</sup>**

U-0	U-0	U-0	U-0	U-0	U-0	U-0	R/W-0
—	—	—	—	—	—	—	ADM_EN
bit 31							bit 24
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ADDR<7:0>							bit 16
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-1
UTXISEL<1:0> <sup>(4)</sup>	UTXINV	URXEN	UTXBRK	UTXEN	UTXBF	TRMT	
bit 15							bit 8
R/W-0	R/W-0	R/W-0	R-1	R-0	R-0	R/W-0	R-0
URXISEL<1:0> <sup>(4)</sup>	ADDEN	RIDLE	PERR	FERR	OERR	URXDA	
bit 7							bit 0

**Legend:**

R = Readable bit

W = Writable bit

P = Programmable bit

U = Unimplemented bit, read as '0'

-n = Bit Value at POR: ('0', '1', x = Unknown)

#### 7.3.6.2.1. UXSTA, rôle des bits

bit 31-25

**Unimplemented:** Read as '0'

bit 24

**ADM\_EN:** Automatic Address Detect Mode Enable bit

1 = Automatic Address Detect mode is enabled

0 = Automatic Address Detect mode is disabled

bit 23-16

**ADDR<7:0>:** Automatic Address Mask bits

When the ADM\_EN bit is '1', this value defines the address character to use for automatic address detection.

bit 15-14

**UTXISEL<1:0>:** TX Interrupt Mode Selection bits<sup>(4)</sup>

For 4-level deep FIFO UART modules:

11 = Reserved, do not use

10 = Interrupt is generated when the transmit buffer becomes empty

01 = Interrupt is generated when all characters have been transmitted

00 = Interrupt is generated when the transmit buffer contains at least one empty space

For 8-level deep FIFO UART modules:

11 = Reserved, do not use

10 = Interrupt is generated and asserted while the transmit buffer is empty

01 = Interrupt is generated and asserted when all characters have been transmitted

00 = Interrupt is generated and asserted while the transmit buffer contains at least one empty space

**Register 21-2: UXSTA: UARTx Status and Control Register<sup>(1,2,3)</sup> (Continued)**

bit 13

**UTXINV:** Transmit Polarity Inversion bit

If IrDA mode is disabled (i.e., IREN (UxMODE<12>) is '0'):

1 = UxTX Idle state is '0'

0 = UxTX Idle state is '1'

If IrDA mode is enabled (i.e., IREN (UxMODE<12>) is '1'):

1 = IrDA encoded UxTX Idle state is '1'

0 = IrDA encoded UxTX Idle state is '0'

bit 12

**URXEN:** Receiver Enable bit

1 = UARTx receiver is enabled. UxRX pin is controlled by UARTx (if ON = 1)

0 = UARTx receiver is disabled. UxRX pin is ignored by the UARTx module. UxRX pin is controlled by port.

bit 11	<b>UTXBRK:</b> Transmit Break bit 1 = Send Break on next transmission. Start bit followed by twelve '0' bits, followed by Stop bit; cleared by hardware upon completion 0 = Break transmission is disabled or completed
bit 10	<b>UTXEN:</b> Transmit Enable bit 1 = UARTx transmitter is enabled. UxTX pin is controlled by UARTx (if ON = 1) 0 = UARTx transmitter is disabled. Any pending transmission is aborted and buffer is reset. UxTX pin is controlled by port.
bit 9	<b>UTXBF:</b> Transmit Buffer Full Status bit (read-only) 1 = Transmit buffer is full 0 = Transmit buffer is not full, at least one more character can be written
bit 8	<b>TRMT:</b> Transmit Shift Register is Empty bit (read-only) 1 = Transmit shift register is empty and transmit buffer is empty (the last transmission has completed) 0 = Transmit shift register is not empty, a transmission is in progress or queued in the transmit buffer
bit 7-6	<b>URXISEL&lt;1:0&gt;:</b> Receive Interrupt Mode Selection bit <sup>(4)</sup> <u>For 4-level deep FIFO UART modules:</u> 11 = Interrupt flag bit is set when receive buffer becomes full (i.e., has 4 data characters) 10 = Interrupt flag bit is set when receive buffer becomes 3/4 full (i.e., has 3 data characters) 0x = Interrupt flag bit is set when a character is received  <u>For 8-level deep FIFO UART modules:</u> 11 = Reserved; do not use 10 = Interrupt flag bit is asserted while receive buffer is 3/4 or more full (i.e., has 6 or more data characters) 01 = Interrupt flag bit is asserted while receive buffer is 1/2 or more full (i.e., has 4 or more data characters) 00 = Interrupt flag bit is asserted while receive buffer is not empty (i.e., has at least 1 data character)
bit 5	<b>ADDEN:</b> Address Character Detect bit (bit 8 of received data = 1) 1 = Address Detect mode is enabled. If 9-bit mode is not selected, this control bit has no effect. 0 = Address Detect mode is disabled

**Register 21-2: UxSTA: UARTx Status and Control Register<sup>(1,2,3)</sup> (Continued)**

bit 4	<b>RIDLE:</b> Receiver Idle bit (read-only) 1 = Receiver is Idle 0 = Data is being received
bit 3	<b>PERR:</b> Parity Error Status bit (read-only) 1 = Parity error has been detected for the current character 0 = Parity error has not been detected
bit 2	<b>FERR:</b> Framing Error Status bit (read-only) 1 = Framing error has been detected for the current character 0 = Framing error has not been detected
bit 1	<b>OERR:</b> Receive Buffer Overrun Error Status bit. This bit is set in hardware and can only be cleared (= 0) in software. Clearing a previously set OERR bit resets the receiver buffer and RSR to empty state. 1 = Receive buffer has overflowed 0 = Receive buffer has not overflowed
bit 0	<b>URXDA:</b> Receive Buffer Data Available bit (read-only) 1 = Receive buffer has data, at least one more character can be read 0 = Receive buffer is empty

### 7.3.6.3. LE REGISTRE UXTXREG (UARTx TRANSMIT REGISTER)

Ce registre permet la transmission. A noter les 8 bits utiles sur les 32.

**Register 21-3: UxTXREG: UARTx Transmit Register**

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
bit 31							bit 24

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
bit 23							bit 16

U-0	U-0	U-0	U-0	U-0	U-0	U-0	R/W-0
—	—	—	—	—	—	—	TX<8>
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
TX<7:0>							
bit 7							bit 0

**Legend:**

R = Readable bit

W = Writable bit

P = Programmable bit

U = Unimplemented bit, read as '0'

-n = Bit Value at POR: ('0', '1', x = Unknown)

bit 31-9      **Unimplemented:** Read as '0'

bit 8-0      **TX<8:0>:** Data bits 8-0 of the character to be transmitted

### 7.3.6.4. LE REGISTRE UXRXREG (UARTx RECEIVE REGISTER)

Ce registre permet la réception. A noter les 8 bits utiles sur les 32.

**Register 21-4: UxRXREG: UARTx Receive Register**

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
bit 31							bit 24

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
bit 23							bit 16

U-0	U-0	U-0	U-0	U-0	U-0	U-0	R-0
—	—	—	—	—	—	—	RX<8>
bit 15							bit 8

R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
RX<7:0>							
bit 7							bit 0

**Legend:**

R = Readable bit

W = Writable bit

P = Programmable bit

U = Unimplemented bit, read as '0'

-n = Bit Value at POR: ('0', '1', x = Unknown)

bit 31-9      **Unimplemented:** Read as '0'

bit 8-0      **RX<8:0>:** Data bits 8-0 of the received character

### 7.3.6.5. LE REGISTRE UXBRG (UARTx BAUDRATE REGISTER)

Ce registre permet la configuration du générateur de baudrate. Plus précisément il s'agit d'établir la valeur du diviseur du PB\_CLOCK pour obtenir le baudrate voulu.

**Register 21-5: UxBRG: UARTx Baud Rate Register<sup>(1,2,3)</sup>**

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
bit 31							bit 24
U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
bit 23							bit 16
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
BRG<15:8>							
bit 15							bit 8
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
BRG<7:0>							
bit 7							bit 0

**Legend:**

R = Readable bit

W = Writable bit

P = Programmable bit

U = Unimplemented bit, read as '0'

-n = Bit Value at POR: ('0', '1', x = Unknown)

bit 31-16      **Unimplemented:** Read as '0'

bit 15-0      **BRG<15:0>:** Baud Rate Divisor bits

### 7.3.7. LE GÉNÉRATEUR DE BAUDRATE

Chaque USART possède un générateur de baudrate indépendant qui lui est associé. Il s'agit d'établir la valeur de UxBRG selon la formule ci-dessous.

**Equation 21-1: UART Baud Rate with BRGH = 0**

$$\text{Baud Rate} = \frac{F_{PB}}{16 \cdot (UxBRG + 1)}$$

$$UxBRG = \frac{F_{PB}}{16 \cdot \text{Baud Rate}} - 1$$

**Note:**  $F_{PB}$  denotes the PBCLK frequency.

En fonction du bit 3 du registre UxMODE on dispose d'une pré-division par 4 ou 16.

bit 3

**BRGH:** High Baud Rate Enable bit

1 = High-Speed mode – 4x baud clock enabled

0 = Standard Speed mode – 16x baud clock enabled

Si BRGH=1, l'équation ci-dessus est modifiée avec un facteur 4 au lieu de 16. Cela résulte en une meilleure finesse de réglage pour les baudrates élevés.

### 7.3.7.1. EXEMPLE CALCUL DU BAUDRATE

Avec BRGH = 0 :

$$\text{Baudrate} = \text{PB\_FREQ} / (16 * (\text{X}+1))$$

où X représente UxBRG.

$$\text{D'où : } \text{X} = ((\text{PB\_FREQ} / (16 * \text{Baudrate})) - 1)$$

Par exemple pour 57'600 bauds et PB\_FREQ = 80 MHz

$$\text{X} = ((80'000'000 / (57'600 * 16)) - 1 = 86.805 - 1 = 85.8 \rightarrow \text{choix 86}$$

Il est nécessaire de calculer l'erreur :

$$\text{Baudrate réel} = \text{PB\_FREQ} / (16(\text{X}+1)) = 80'000'000 / (16(86+1)) = \\ 80'000'000 / 1'392 = 57'471.2$$

L'erreur est de  $(57'471.2 - 57'600) / 57'600 = -0.0022$  soit - 0.22 %

L'erreur ne doit pas dépasser 2%, sinon il y a des risques d'erreur de lecture. On peut améliorer la précision pour les baudrate élevés en utilisant le pré-diviseur par 4.

#### 7.3.7.1.1. Tableau des écarts pour 40 MHz

Le fabriquant fournit un tableau des écarts pour PB\_FREQ = 40 MHz et BRGH = 0

Target Baud Rate	Peripheral Bus Clock: 40 MHz		
	Actual Baud Rate	% Error	BRG Value (decimal)
110	110.0	0.00	22726.0
300	300.0	0.00	8332.0
1200	1200.2	0.02	2082.0
2400	2399.2	-0.03	1041.0
9600	9615.4	0.16	259.0
19.2 K	19230.8	0.16	129.0
38.4 K	38461.5	0.16	64.0
56 K	55555.6	-0.79	44.0
115 K	113636.4	-1.19	21.0
250 K	250000.0	0.00	9.0
300 K	500000.0	0.00	4.0
500 K	500000.0	0.00	4.0
Min. Rate	38.1	0.0	65535
Max. Rate	2500000	0.0	0

## 7.4. CONFIGURATION DE L'USART AVEC LA PLIB\_USART

La configuration de l'USART est réalisée par les étapes suivantes :

- Sélection du baudrate (**BaudRateSet**)
- Sélection du mode de handshake (**HandshakeModeSelect**)
- Sélection du mode d'opération (**OperationModeSelect**)
- Activation du transmetteur (**TransmitterEnable**)
- Sélection du mode d'interruption du transmetteur (**TransmitterInterruptModeSelect**)
- Activation du récepteur (**ReceiverEnable**)
- Sélection du mode d'interruption du récepteur (**ReceiverInterruptModeSelect**)

### 7.4.1. LE TYPE ÉNUMÉRÉ USART\_MODULE\_ID

Le type énuméré USART\_MODULE\_ID liste les USART à disposition.

```
typedef enum {
    USART_ID_1 = 0,
    USART_ID_3,
    USART_ID_2,
    USART_ID_4,
    USART_ID_6,
    USART_ID_5,
    USART_NUMBER_OF_MODULES
} USART_MODULE_ID;
```

### 7.4.2. LA FONCTION PLIB\_USART\_BAUDRATESET

La fonction **PLIB\_USART\_BaudRateSet**, permet d'établir le baudrate.

```
void PLIB_USART_BaudRateSet(USART_MODULE_ID index, uint32_t clockFrequency, uint32_t baudRate);
```

Exemple :

```
PLIB_USART_BaudRateSet(USART_ID_1,
                      SYS_DEVCON_SYSTEM_CLOCK, 57600);
```

### 7.4.3. LA FONCTION PLIB\_USART\_HANDSHAKEMODESELECT

La fonction **PLIB\_USART\_HandshakeModeSelect**, permet d'établir comment est géré le handshaking.

```
void PLIB_USART_HandshakeModeSelect(USART_MODULE_ID index, USART_HANDSHAKE_MODE handshakeConfig);
```

Pour le handshakeConfig, on dispose de 2 possibilités :

Members	Description
USART_HANDSHAKE_MODE_FLOW_CONTROL	Enables flow control
USART_HANDSHAKE_MODE_SIMPLEX	Enables simplex mode communication, no flow control

Le mode FLOW\_CONTROL établit le mode de gestion par le hardware. Le mode SIMPLEX permet une gestion par logiciel.

Exemple :

```
PLIB_USART_HandshakeModeSelect(USART_ID_1,
                                USART_HANDSHAKE_MODE_SIMPLEX);
```

#### 7.4.4. LA FONCTION PLIB\_USART\_OPERATIONMODESELECT

La fonction **PLIB\_USART\_OperationModeSelect**, permet d'établir comment sont gérées les broches TX, RX, CTS et RTS.

```
void PLIB_USART_OperationModeSelect(USART_MODULE_ID index, USART_OPERATION_MODE operationmode);
```

Pour l'operation mode, on dispose de 4 possibilités :

Members	Description
USART_ENABLE_TX_RX_BCLK_USED	TX, RX and BCLK pins are used by USART module
USART_ENABLE_TX_RX_CTS_RTS_USED	TX, RX, CTS and RTS pins are used by USART module
USART_ENABLE_TX_RX_RTS_USED	TX, RX and RTS pins are used by USART module
USART_ENABLE_TX_RX_USED	TX and RX pins are used by USART module

Pour être cohérent avec le mode SIMPLEX, il faut utiliser le dernier élément pour utiliser uniquement les broches TX et RX de l'USART.

Exemple :

```
PLIB_USART_OperationModeSelect(USART_ID_1,
                                USART_ENABLE_TX_RX_USED);
```

#### 7.4.5. LA FONCTION PLIB\_USART\_LINECONTROLMODESELECT

La fonction **PLIB\_USART\_LineControlModeSelect**, permet d'établir comment sont traitées les données.

```
void PLIB_USART_LineControlModeSelect(USART_MODULE_ID index, USART_LINECONTROL_MODE dataFlowConfig);
```

On dispose des possibilités suivantes pour le dataFlowConfig :

Members	Description
USART_8N1	8 Data Bits, No Parity, one Stop Bit
USART_8E1	8 Data Bits, Even Parity, 1 stop bit
USART_8O1	8 Data Bits, odd Parity, 1 stop bit
USART_8N2	8 Data Bits, No Parity, two Stop Bits
USART_8E2	8 Data Bits, Even Parity, 2 stop bits
USART_8O2	8 Data Bits, odd Parity, 2 stop bits
USART_9N1	9 Data Bits, No Parity, 1 stop bit
USART_9N2	9 Data Bits, No Parity, 2 stop bits

#### 7.4.5.1. LE TYPE ÉNUMÉRÉ USART\_LINECONTROL\_MODE

Le type énuméré USART\_LINECONTROL\_MODE est défini ainsi:

```
typedef enum {
    USART_8N1 = 0x00,
    USART_8E1 = 0x01,
    USART_8O1 = 0x02,
    USART_9N1 = 0x03,
    USART_8N2 = 0x04,
    USART_8E2 = 0x05,
    USART_8O2 = 0x06,
    USART_9N2 = 0x07
} USART_LINECONTROL_MODE;
```

#### 7.4.5.2. USART\_LINECONTROL\_MODE, EXEMPLE

Etablit le classique data bits = 8, parity = none et stop bits = 1

```
PLIB_USART_LineControlModeSelect(USART_ID_1,
                                  USART_8N1);
```

#### 7.4.6. LA FONCTION PLIB\_USART\_TRANSMITTERENABLE

La fonction **PLIB\_USART\_TransmitterEnable**, permet d'activer le transmetteur.

Exemple :

```
PLIB_USART_TransmitterEnable(USART_ID_1);
```

#### 7.4.7. PLIB\_USART\_TRANSMITTERINTERRUPTMODESELECT

La fonction **PLIB\_USART\_TransmitterInterruptModeSelect**, permet d'établir comment doit réagir l'interruption de transmission.

```
void PLIB_USART_TransmitterInterruptModeSelect(USART_MODULE_ID index, USART_TRANSMIT_INTR_MODE
                                              fifolevel);
```

Les choix sont les suivants :

Members	Description
USART_TRANSMIT_FIFO_EMPTY	Interrupt when the transmit buffer becomes empty
USART_TRANSMIT_FIFO_IDLE	Interrupt when all characters are transmitted
USART_TRANSMIT_FIFO_NOT_FULL	Interrupt when at least one location is empty in the transmit buffer

#### 7.4.7.1. LE TYPE ÉNUMÉRÉ USART\_TRANSMIT\_INTR\_MODE

Le type énuméré USART\_TRANSMIT\_INTR\_MODE est défini ainsi :

```
typedef enum {
    USART_TRANSMIT_FIFO_NOT_FULL = 0x00,
    USART_TRANSMIT_FIFO_IDLE = 0x01,
    USART_TRANSMIT_FIFO_EMPTY = 0x02
} USART_TRANSMIT_INTR_MODE;
```

#### 7.4.7.2. PLIB\_USART\_TRANSMITTERINTERRUPTMODESELECT, EXEMPLE

Dans cet exemple, l'interruption de transmission est configurée pour se produire lorsque le tampon de transmission est vide.

```
PLIB_USART_TransmitterInterruptModeSelect(USART_ID_1,
                                         USART_TRANSMIT_FIFO_EMPTY);
```

#### 7.4.8. LA FONCTION PLIB\_USART\_RECEIVERENABLE

La fonction **PLIB\_USART\_ReceiverEnable**, permet d'activer le récepteur.

Exemple :

```
PLIB_USART_ReceiverEnable(USART_ID_1);
```

#### 7.4.9. PLIB\_USART\_RECEIVERINTERRUPTMODESELECT

La fonction **PLIB\_USART\_ReceiverInterruptModeSelect**, permet d'établir comment doit réagir l'interruption de réception.

```
void PLIB_USART_ReceiverInterruptModeSelect(USART_MODULE_ID index, USART_RECEIVE_INTR_MODE interruptMode);
```

Les choix sont les suivants :

Members	Description
USART_RECEIVE_FIFO_HALF_FULL	Interrupt when receive buffer is half full
USART_RECEIVE_FIFO_3B4FULL	Interrupt when receive buffer is 3/4 full
USART_RECEIVE_FIFO_ONE_CHAR	Interrupt when a character is received

#### 7.4.9.1. LE TYPE ÉNUMÉRÉ USART\_RECEIVE\_INTR\_MODE

Le type énuméré USART\_TRANSMIT\_INTR\_MODE est défini ainsi :

```
typedef enum {
    USART_RECEIVE_FIFO_3B4FULL = 0x02,
    USART_RECEIVE_FIFO_HALF_FULL = 0x01,
    USART_RECEIVE_FIFO_ONE_CHAR = 0x00
} USART_RECEIVE_INTR_MODE;
```

#### 7.4.9.2. PLIB\_USART\_RECEIVERINTERRUPTMODESELECT, EXEMPLE

Dans cet exemple, l'USART est configuré pour que l'interruption de réception se produise lorsque le tampon est aux ¾ plein.

```
PLIB_USART_ReceiverInterruptModeSelect(USART_ID_1,
                                         USART_RECEIVE_FIFO_3B4FULL);
```

#### 7.4.10. EXEMPLE DE CONFIGURATION

Voici un exemple complet de configuration avec aussi la configuration des interruptions. La configuration établit 57'600 bauds et data bits = 8, parity = none et stop bits = 1.

```

PLIB_USART_Disable(USART_ID_1);
PLIB_USART_BaudRateSet(USART_ID_1, SYS_DEVCON_SYSTEM_CLOCK,
                        57600);
PLIB_USART_HandshakeModeSelect(USART_ID_1,
                               USART_HANDSHAKE_MODE_SIMPLEX);
PLIB_USART_OperationModeSelect(USART_ID_1,
                               USART_ENABLE_TX_RX_USED);
PLIB_USART_LineControlModeSelect(USART_ID_1, USART_8N1);
PLIB_USART_TransmitterEnable(USART_ID_1);
PLIB_USART_TransmitterInterruptModeSelect(USART_ID_1,
                                         USART_TRANSMIT_FIFO_EMPTY);
PLIB_USART_ReceiverEnable(USART_ID_1);
PLIB_USART_ReceiverInterruptModeSelect(USART_ID_1,
                                         USART_RECEIVE_FIFO_3B4FULL);

/* Initialize interrupts */
PLIB_INT_SourceEnable(INT_ID_0,
                      INT_SOURCE_USART_1_TRANSMIT);
PLIB_INT_SourceEnable(INT_ID_0,
                      INT_SOURCE_USART_1_RECEIVE);
PLIB_INT_SourceEnable(INT_ID_0, INT_SOURCE_USART_1_ERROR);
PLIB_INT_VectorPrioritySet(INT_ID_0, INT_VECTOR_UART1,
                           INT_PRIORITY_LEVEL5);
PLIB_INT_VectorSubPrioritySet(INT_ID_0, INT_VECTOR_UART1,
                             INT_SUBPRIORITY_LEVEL0);
// activation en tout dernier
PLIB_USART_Enable(USART_ID_1);

```

## 7.5. FONCTIONS DE L'ÉMETTEUR

On dispose d'un certain nombre de fonctions liées à l'émission.

Name	Description
PLIB_USART_Transmitter9BitsSend	Data to be transmitted in the byte mode with the 9th bit.
PLIB_USART_TransmitterBreakSend	Transmits the break character.
PLIB_USART_TransmitterBreakSendIsComplete	Returns the status of the break transmission
PLIB_USART_TransmitterBufferIsFull	Gets the transmit buffer full status.
PLIB_USART_TransmitterByteSend	Data to be transmitted in the Byte mode.
PLIB_USART_TransmitterDisable	Disables the specific USART module transmitter.
PLIB_USART_TransmitterEnable	Enables the specific USART module transmitter.
PLIB_USART_TransmitterIdleIsLowDisable	Disables the Transmit Idle Low state.
PLIB_USART_TransmitterIdleIsLowEnable	Enables the Transmit Idle Low state.
PLIB_USART_TransmitterInterruptModeSelect	Set the USART transmitter interrupt mode.
PLIB_USART_TransmitterIsEmpty	Gets the transmit shift register empty status.
PLIB_USART_TransmitterIsIdle	Gets the transmit buffer full status.
PLIB_USART_TransmitterAddressGet	Returns the address of the USART Tx register

Voici la description des fonctions les plus utiles.

### 7.5.1. LA FONCTION PLIB\_USART\_TRANSMITTERBYTESEND

La fonction **PLIB\_USART\_TransmitterByteSend** permet de transmettre un octet.

```
void PLIB_USART_TransmitterByteSend(USART_MODULE_ID index, int8_t data);
```

Il est nécessaire de déterminer s'il y a de la place dans le tampon d'émission avant d'appeler la fonction.

### 7.5.2. LA FONCTION PLIB\_USART\_TRANSMITTERBUFFERISFULL

La fonction **PLIB\_USART\_TransmitterBufferIsFull** permet de connaître la situation du tampon d'émission.

```
bool PLIB_USART_TransmitterBufferIsFull(USART_MODULE_ID index);
```

Si true, le buffer est plein. Avec false, le buffer n'est pas plein, il y a au moins la place pour un caractère.

### 7.5.3. EXEMPLE D'ÉMISSION AVEC TRANSMITTERBYTESEND

Voici un exemple d'émission d'une chaîne de caractères (terminée par un caractère 0x00). Ce code va attendre tant que le tampon d'émission est plein avant chaque byte.

```
while (Buffer[i] != 0) {
    // Attente si buffer full
    while (PLIB_USART_TransmitterBufferIsFull
        (USART_ID_1)) {
    }
    PLIB_USART_TransmitterByteSend(USART_ID_1, Buffer[i]);
    i++;
}
```

## 7.6. FONCTIONS DU RÉCEPTEUR

On dispose d'un certain nombre de fonctions liées à la réception.

Name	Description
<a href="#">PLIB_USART_ReceiverAddressAutoDetectDisable</a>	Disables the automatic Address Detect mode.
<a href="#">PLIB_USART_ReceiverAddressAutoDetectEnable</a>	Setup the automatic Address Detect mode.
<a href="#">PLIB_USART_ReceiverAddressDetectDisable</a>	Enables the Address Detect mode.
<a href="#">PLIB_USART_ReceiverAddressDetectEnable</a>	Enables the Address Detect mode.
<a href="#">PLIB_USART_ReceiverAddressIsReceived</a>	Checks and return if the data received is an address.
<a href="#">PLIB_USART_ReceiverByteReceive</a>	Data to be received in the Byte mode.
<a href="#">PLIB_USART_ReceiverDataIsAvailable</a>	Identifies if the receive data is available for the specified USART module.
<a href="#">PLIB_USART_ReceiverDisable</a>	Disables the USART receiver.
<a href="#">PLIB_USART_ReceiverEnable</a>	Enables the USART receiver.
<a href="#">PLIB_USART_ReceiverFramingErrorHasOccurred</a>	Gets the framing error status.
<a href="#">PLIB_USART_ReceiverIdleStateLowDisable</a>	Disables receive polarity inversion.
<a href="#">PLIB_USART_ReceiverIdleStateLowEnable</a>	Enables receive polarity inversion.
<a href="#">PLIB_USART_ReceiverInterruptModeSelect</a>	Sets the USART receiver FIFO level.
<a href="#">PLIB_USART_ReceiverIsIdle</a>	Identifies if the receiver is idle.
<a href="#">PLIB_USART_ReceiverModeSelect</a>	Disables Single Receive mode.
<a href="#">PLIB_USART_ReceiverOverrunErrorClear</a>	Clears a USART Overrun error.
<a href="#">PLIB_USART_ReceiverOverrunHasOccurred</a>	Identifies if there was a receiver overrun error.
<a href="#">PLIB_USART_ReceiverParityErrorHasOccurred</a>	Gets the parity error status.
<a href="#">PLIB_USART_ReceiverAddressGet</a>	Returns the address of the USART Rx register

Voici la description des fonctions les plus utiles.

### 7.6.1. LA FONCTION PLIB\_USART\_RECEIVERBYTERECEIVE

La fonction **PLIB\_USART\_ReceiverByteReceive** permet de lire un byte des données reçues par l'USART et stockées dans le tampon de réception.

```
int8_t PLIB_USART_ReceiverByteReceive(USART_MODULE_ID index);
```

Il faut s'assurer qu'un byte est disponible.

### 7.6.2. LA FONCTION PLIB\_USART\_RECEIVERDATAISAVAILABLE

La fonction **PLIB\_USART\_ReceiverDataIsAvailable** permet de savoir si une donnée est disponible.

```
bool PLIB_USART_ReceiverDataIsAvailable(USART_MODULE_ID index);
```

#### 7.6.2.1. PLIB\_USART\_RECEIVERDATAISAVAILABLE, EXEMPLE

Cet exemple repris de la réponse à l'interruption de réception montre comment exploiter le tampon de réception. Avec int8\_t c;

```
while ( PLIB_USART_ReceiverDataIsAvailable(USART_ID_1) )
{
    c = PLIB_USART_ReceiverByteReceive(USART_ID_1);
    PutCharInFifo ( &descrFifoRX, c );
}
```

### 7.6.3. FONCTION PLIB\_USART\_RECEIVEROVERRUNERRORCLEAR

La fonction **PLIB\_USART\_ReceiverOverrunErrorClear** permet de supprimer une erreur d'overrun (débordement du buffer de réception). L'appel de cette fonction efface les données reçues.

```
void PLIB_USART_ReceiverOverrunErrorClear(USART_MODULE_ID index);
```

## 7.7. FONCTIONS GÉNÉRALES DE L'USART

Voici les nombreuses fonctions générales qui ne s'appliquent pas toutes pour le PIC32MX.

Name	Description
<a href="#">PLIB_USART_AlternateIODisable</a>	Disables the use of alternate I/O pins for the receiver and transmitter.
<a href="#">PLIB_USART_AlternateIOEnable</a>	Enables the use of alternate I/O pins for the receiver and transmitter.
<a href="#">PLIB_USART_Disable</a>	Disables the specific USART module
<a href="#">PLIB_USART_Enable</a>	Enables the specific USART module.
<a href="#">PLIB_USART_HandshakeModeSelect</a>	Sets the data flow configuration.
<a href="#">PLIB_USART_IrDADisable</a>	Disables the IrDA encoder and decoder.
<a href="#">PLIB_USART_IrDAEnable</a>	Enables the IrDA encoder and decoder.
<a href="#">PLIB_USART_LineControlModeSelect</a>	Sets the data flow configuration.
<a href="#">PLIB_USART_LoopbackDisable</a>	Disables Loopback mode.
<a href="#">PLIB_USART_LoopbackEnable</a>	Enables Loopback mode.
<a href="#">PLIB_USART_OperationModeSelect</a>	Configures the operation mode of the USART module.
<a href="#">PLIB_USART_StopInIdleDisable</a>	Disables the Stop in Idle mode (the USART module continues operation when the device is in Idle mode).
<a href="#">PLIB_USART_StopInIdleEnable</a>	Discontinues operation when the device enters Idle mode.
<a href="#">PLIB_USART_SyncClockPolarityIdleHighDisable</a>	Sets the idle state of the clock to low.
<a href="#">PLIB_USART_SyncClockPolarityIdleHighEnable</a>	Sets the idle state of the clock to high.
<a href="#">PLIB_USART_SyncClockSourceSelect</a>	Sets the clock source.
<a href="#">PLIB_USART_SyncModeSelect</a>	Selects the USART mode to be asynchronous.
<a href="#">PLIB_USART_WakeOnStartDisable</a>	Disables the wake-up on start bit detection feature during Sleep mode.
<a href="#">PLIB_USART_WakeOnStartEnable</a>	Enables the wake-up on start bit detection feature during Sleep mode.
<a href="#">PLIB_USART_WakeOnStartIsEnabled</a>	Gets the state of the sync break event completion.
<a href="#">PLIB_USART_ErrorsGet</a>	Return the status of all errors in the specified USART module.

Voici la description des fonctions les plus utiles :

### 7.7.1. LA FONCTION PLIB\_USART\_DISABLE

La fonction **PLIB\_USART\_Disable** permet de désactiver l'USART. Cela a pour effet de vider les tampons d'émission et de réception.

```
void PLIB_USART_Disable(USART_MODULE_ID index);
```

### 7.7.2. LA FONCTION PLIB\_USART\_ENABLE

La fonction **PLIB\_USART\_Enable** permet d'activer l'USART. Cela a pour effet que les broches RX et TX sont gérées par l'USART.

```
void PLIB_USART_Enable(USART_MODULE_ID index);
```

### 7.7.3. LA FONCTION PLIB\_USART\_ErrorGet

La fonction **PLIB\_USART\_ErrorGet** permet d'obtenir la situation des bits d'erreurs.

```
USART_ERROR PLIB_USART_ErrorGet(USART_MODULE_ID index);
```

Le type USART\_ERROR est défini ainsi :

```
typedef enum {

    USART_ERROR_NONE = 0x00,
    USART_ERROR_RECEIVER_OVERRUN = 0x01,
    USART_ERROR_FRAMING = 0x02,
    USART_ERROR_PARITY = 0x04
} USART_ERROR;
```

### 7.7.4. EXEMPLE GESTION DES ERREURS

L'exemple repris de la réponse à l'interruption de réception montre le traitement des erreurs.

```
USART_ERROR UsartStatus = PLIB_USART_ErrorGet(USART_ID_1);

if ( (UsartStatus & (USART_ERROR_PARITY |
                      USART_ERROR_FRAMING |
                      USART_ERROR_RECEIVER_OVERRUN)) == 0) {
    // OK traitement de réception possible
} else {
    // Suppression des erreurs
    // La lecture des erreurs les efface sauf pour overrun
    if ( (UsartStatus & USART_ERROR_RECEIVER_OVERRUN) ==
                    USART_ERROR_RECEIVER_OVERRUN) {
        PLIB_USART_ReceiverOverrunErrorClear(USART_ID_1);
    }
}
```

## 7.8. EMISSION ET RÉCEPTION SANS INTERRUPTION

Cette section a pour but de montrer l'utilisation des fonctions de base pour réaliser une transmission et une réception.

### 7.8.1. CONFIGURATION USART SANS INTERRUPTION

Voici la configuration de l'USART sans interruption, pour un baudrate de 19'200.

```
void DRV_USART0_Initialize(void)
{
    /* Initialize USART */
    PLIB_USART_BaudRateSet(USART_ID_1,
                           SYS_DEVCON_SYSTEM_CLOCK, 19200);
    PLIB_USART_HandshakeModeSelect(USART_ID_1,
                                   USART_HANDSHAKE_MODE_SIMPLEX);
    PLIB_USART_OperationModeSelect(USART_ID_1,
                                   USART_ENABLE_TX_RX_USED);
    PLIB_USART_LineControlModeSelect(USART_ID_1,
                                    USART_8N1);
    PLIB_USART_TransmitterEnable(USART_ID_1);
    PLIB_USART_TransmitterInterruptModeSelect(USART_ID_1,
                                              USART_TRANSMIT_FIFO_EMPTY);
    PLIB_USART_ReceiverEnable(USART_ID_1);
    PLIB_USART_ReceiverInterruptModeSelect(USART_ID_1,
                                              USART_RECEIVE_FIFO_ONE_CHAR);
    PLIB_USART_Enable(USART_ID_1) ; // nécessaire
}
```

Dans ce code, on configure tout de même le comportement des interruptions de l'USART, même si on ne les utilise pas.

### 7.8.2. EMISSION EN POLLING

Voici une fonction qui reçoit la valeur de deux entrées analogiques pour en fabriquer un message complet et le transmettre :

```
void APP_SendMessAD ( S_ADCResults AdcRes )
{
    char Buffer[40];
    int i;

    sprintf(Buffer, "Canal0 %4d Canal1 %4d \n",
            AdcRes.Chan0, AdcRes.Chan1);

    while (Buffer[i] != 0) {
        // Attente si buffer full
        while (PLIB_USART_TransmitterBufferIsFull
              (USART_ID_1)) {
            BSP_LEDToggle(BSP_LED_2);
        }
        PLIB_USART_TransmitterByteSend(USART_ID_1, Buffer[i]);
        i++;
    }
}
```

Cette fonction attend tant que le tampon d'émission est plein en utilisant la fonction **PLIB\_USART\_TransmitterBufferIsFull**. On insère dans la boucle d'attente l'inversion de la LED\_2 pour observer la durée des attentes.

Lorsque l'on sort de la boucle d'attente, on appelle la fonction **PLIB\_USART\_TransmitterByteSend** qui dépose le caractère à émettre dans le tampon de transmission qui a alors au moins une place de disponible.

### 7.8.3. UTILISATION DE LA FONCTION D'ÉMISSION ET OBSERVATIONS

Grace à une interruption, on active l'application toutes les 20 ms et on place l'appel de la fonction dans le case APP\_STATE\_SERVICE\_TASKS.

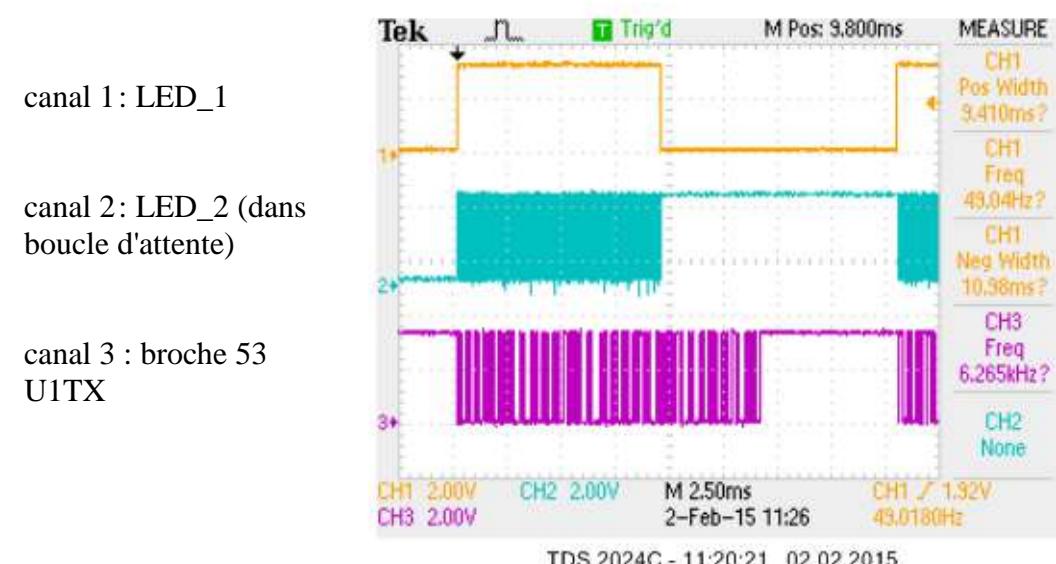
```
case APP_STATE_SERVICE_TASKS:
    // Lecture des 2 pots
    AdcRes = BSP_ReadAllADC();
    // affichage local des valeurs de l'AD
    lcd_gotoxy(1,3);
    printf_lcd("Ad Chan0 %4d      ", AdcRes.Chan0);
    lcd_gotoxy(1,4);
    printf_lcd("Ad Chan1 %4d      ", AdcRes.Chan1);

    BSP_LEDOff(BSP_LED_1);
    // Emet sur la comm serie
    APP_SendMessAD(AdcRes);
    BSP_LEDOn(BSP_LED_1);

    appData.state = APP_STATE_WAIT;
break;
```

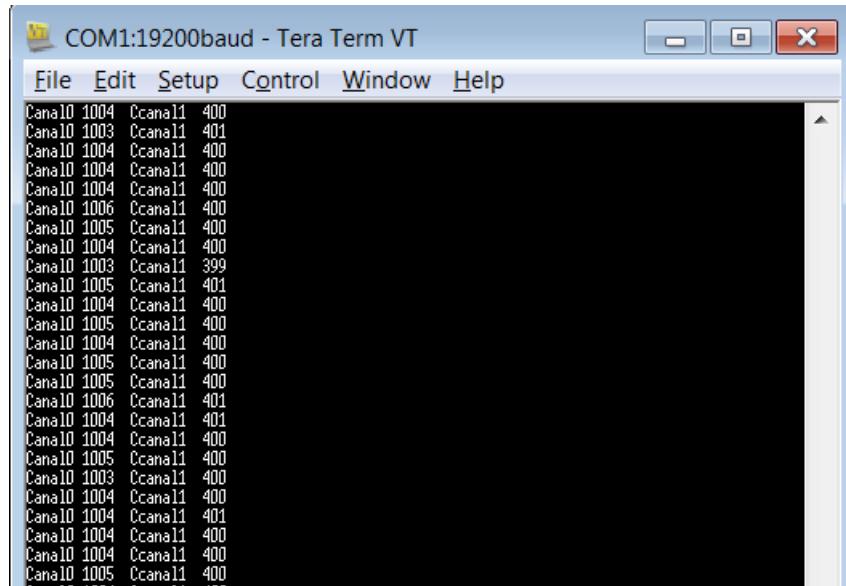
#### 7.8.3.1. COMPORTEMENT DE L'ÉMISSION

De l'observation ci-dessous, on déduit que le temps d'exécution de la fonction APP\_SendMessAD est de 9.4 ms. On observe encore que la transmission continue après la fin de la fonction car on a rempli le tampon d'émission.



#### 7.8.3.2. RÉSULTAT ÉMISSION AVEC TERMINAL

On obtient :



The screenshot shows a window titled "COM1:19200baud - Tera Term VT". The menu bar includes File, Edit, Setup, Control, Window, and Help. The main window displays a series of text entries, likely representing serial port data. The entries consist of three columns: "Canal0", "1004", and "Ccanal1". The values in the first column range from 1003 to 1006. The values in the second column are either 400 or 399. The values in the third column are either 400, 401, or 402. The data is repeated multiple times.

Canal0	1004	Ccanal1
Canal0	1003	Ccanal1
Canal0	1004	401
Canal0	1004	400
Canal0	1004	400
Canal0	1004	400
Canal0	1006	400
Canal0	1005	400
Canal0	1004	400
Canal0	1003	Ccanal1
Canal0	1005	401
Canal0	1004	400
Canal0	1005	400
Canal0	1004	400
Canal0	1005	400
Canal0	1005	400
Canal0	1005	400
Canal0	1006	401
Canal0	1004	401
Canal0	1004	400
Canal0	1005	400
Canal0	1003	Ccanal1
Canal0	1004	400
Canal0	1004	401
Canal0	1004	400
Canal0	1004	400
Canal0	1005	400

### 7.8.4. RÉCEPTION EN POLLING

Pour tester la réception, nous utiliserons une application qui envoie un message toutes les 25 ms. Il s'agit d'un message de type texte reprenant le principe du message émis, mais auquel nous ajoutons un caractère de début et un caractère de fin. Voici un exemple :

```
!Ch0 0976 Ch1 0478#
```

Cela nous fait un message de 19 caractères. A 19'200 bauds, avec 10 bits par caractère, le temps pour un bit est de  $1/19'200 = 0.052$  ms, donc 0.52 ms pour un caractère et 9.88 ms pour la transmission du message complet. Comme l'application a un cycle de 20 ms, cela convient pour le traitement du message. Par contre, comme le tampon de réception hardware de l'USART ne contient que 8 caractères, il est indispensable de traiter beaucoup plus rapidement la réception des caractères. Nous disposons d'une interruption à 400 us qui convient bien même pour une réception caractère par caractère. Nous avons encore besoin d'un élément de stockage pour l'ensemble du message avant son traitement par l'application, pour cela nous utiliserons un FIFO software.

#### 7.8.4.1. FONCTION DE REMPLISSAGE DU FIFO

Cette fonction est appelée cycliquement (période de 400 us). Elle prend les caractères reçus dans le tampon de réception et les copie dans le FIFO.

```
void APP_FillRxFifo(void) {
    USART_ERROR UsartStatus;
    int8_t c;

    // Test si erreur parité ou overrun
    UsartStatus = PLIB_USART_ErrorsGet(USART_ID_1);

    if ( (UsartStatus & (USART_ERROR_PARITY |
                           USART_ERROR_FRAMING |
                           USART_ERROR_RECEIVER_OVERRUN)) == 0) {
        // transfert dans le FIFO de tous les char reçus
        while (PLIB_USART_ReceiverDataIsAvailable
               (USART_ID_1))
    {
        c = PLIB_USART_ReceiverByteReceive
            (USART_ID_1);
        PutCharInFifo (&descrFifoRX, c);
        BSP_LEDToggle(BSP_LED_3);
    }
    } else {
        // La lecture des erreurs les efface
        // sauf pour overrun
        if ( (UsartStatus & USART_ERROR_RECEIVER_OVERRUN)
            == USART_ERROR_RECEIVER_OVERRUN) {
            PLIB_USART_ReceiverOverrunErrorClear
                (USART_ID_1);
        }
    }
}
```

#### 7.8.4.2. APPEL DE LA FONCTION DANS LA RÉPONSE À L'INTERRUPTION

Dans la réponse à l'interruption du Timer1, on appelle à chaque cycle la fonction **APP\_FillRx\_fifo()**. Tous les 50 cycles (20 ms) on active l'application.

```
// ISR Timer1 période 400 us
void __ISR(_TIMER_1_VECTOR, ipl4)
    _IntHandlerDrvTmrInstance0(void)
{
    static int count = 0;
    PLIB_INT_SourceFlagClear(INT_ID_0, INT_SOURCE_TIMER_1);
    BSP_LEDToggle(BSP_LED_0); // pour contrôle au scope
    // Appel fonction de réception
    APP_FillRx_fifo();

    count++;
    if ( count > 50 ) { // 20 ms
        count = 0;
        APP_UpdateState(APP_STATE_SERVICE_TASKS);
    }
}
```

#### 7.8.4.3. LA FONCTION APP\_GETMESSAD

Cette fonction traite le message en fonction du nombre de caractères disponibles. Elle se synchronise sur le caractère de début '!' et termine le traitement lorsqu'elle trouve le caractère de fin '#'. La fonction retourne vrai lorsque le message est disponible.

```
bool APP_GetMessAD(char *pMess) {
    bool stat = false;
    bool EndMess = false;
    uint8_t NbCharToRead = 0;
    static uint8_t RxC;
    static int GetSituation = WaitSTX;
    static int MessIdx = 0;

    // Détermine le nombre de caractères à lire
    NbCharToRead = GetReadSize (&descrFifoRX);

    EndMess = false;
    while ( (NbCharToRead >= 1) &&
            (EndMess == false) ) {
```

```

// Lis un caractère sans gestion du status
GetCharFromFifo ( &descrFifoRX, &RxC ) ;
NbCharToRead--;

switch (GetSituation) {
    case WaitSTX :
        if (RxC == '!') {
            MessIdx = 0;
            pMess[MessIdx] = RxC;
            MessIdx++;
            GetSituation = WaitETX;
            BSP_LEDToggle(BSP_LED_4);
        }
        break;

    case WaitETX :
        pMess[MessIdx] = RxC;
        MessIdx++;
        if (RxC == '#') {
            pMess[MessIdx] = 0; // nul
            MessIdx = 0;
            EndMess = true;
            stat = true;
            GetSituation = WaitSTX;
            BSP_LEDToggle(BSP_LED_4);
        }
        break;
    } // end switch
} // end while
return stat;
}

```

#### 7.8.4.4. UTILISATION DANS L'APPLICATION

Appel de la fonction dans le case APP\_STATE\_SERVICE\_TASKS avec affichage si StatMess est vrai.

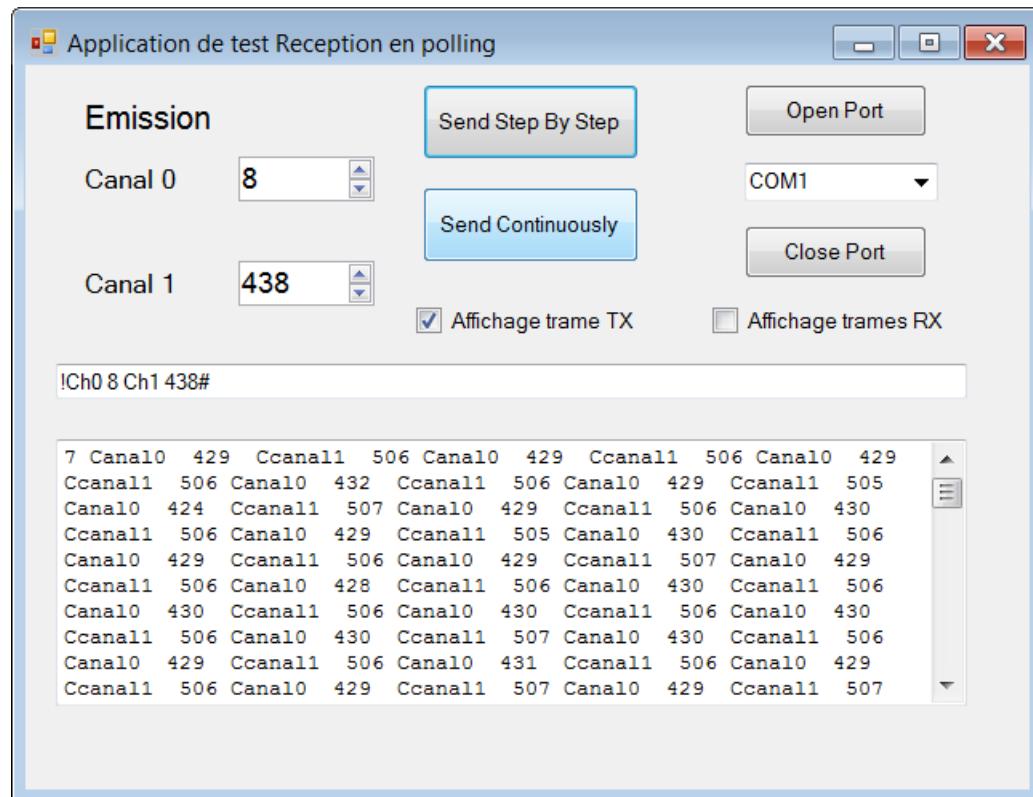
```

StatMess = APP_GetMessAD(RxMess);
if (StatMess == true) {
    lcd_gotoxy(1,4);
    printf_lcd("%s", RxMess);
    StatMess == false; // consommation du message
}

```

#### 7.8.4.5. EXEMPLE DE RÉSULTAT AVEC L'APPLICATION

Sur l'affichage du KIT, on retrouve bien le message émis :



#### 7.8.4.6. COMPORTEMENT DE LA RÉCEPTION

canal 1: LED\_1

canal 2: LED\_3  
(inversion à chaque caractère obtenu du tampon HW de réception)

canal 3 : broche 52 U1RX



Le signal sur la LED\_3 montre que l'on n'obtient pas un caractère à chaque interruption, donc que le tampon de réception ne stocke pas.

#### 7.8.4.7. CONCLUSION SUR RÉCEPTION EN POLLING

Cette solution est fonctionnelle et convient si l'on dispose d'une interruption cyclique. Un FIFO software est par contre indispensable.

## 7.9. NÉCESSITÉ DU RECOURS À UN FIFO

Un tampon (*Buffer*) est très souvent utilisé en communication pour gérer le classique problème du producteur et du consommateur. En général, le producteur (réception des caractères) travaille caractère par caractère (en général sous interruption). Le consommateur (traitement du message reçu) travaille plutôt par à-coup en lisant message après message.

Au niveau émission, le principe s'inverse : le producteur dépose un message et ce message est émis caractère par caractère en fonction des possibilités de l'USART.

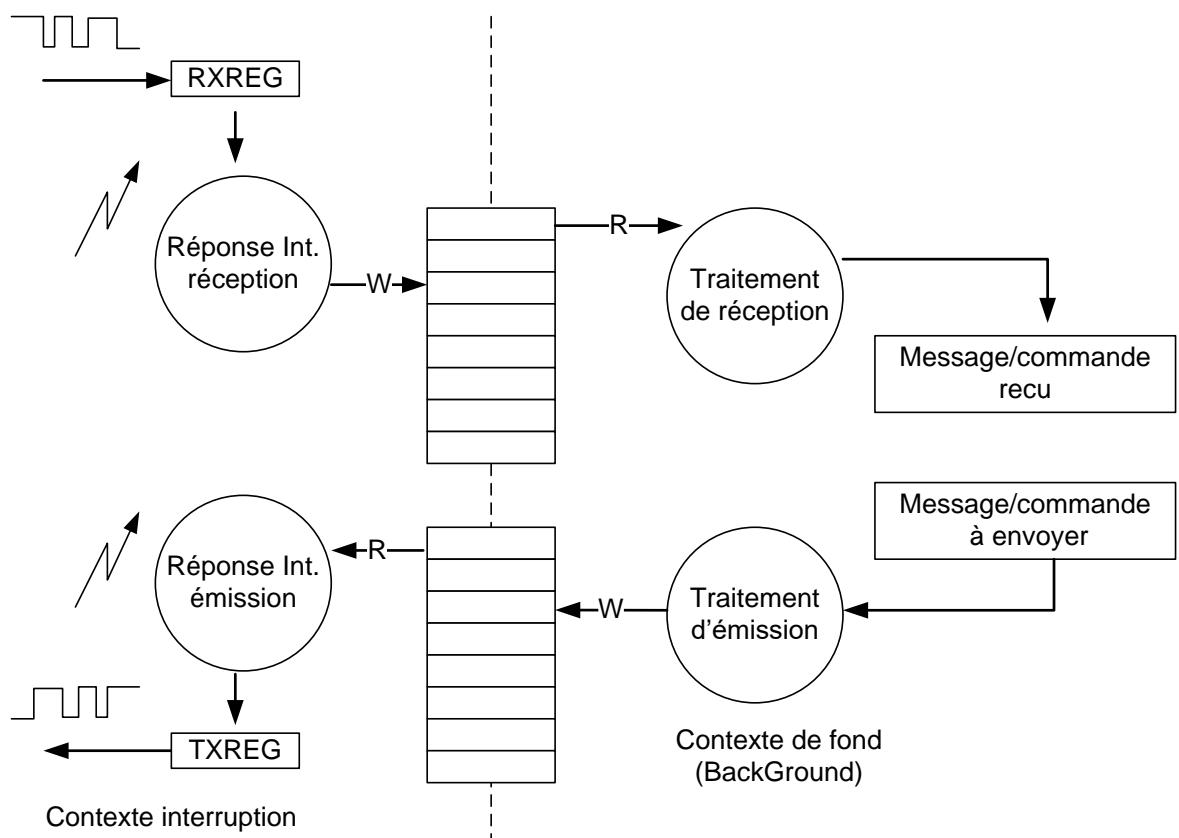
### 7.9.1. RÔLES DU FIFO

Le FIFO a 2 fonctions importantes :

- Absorber temporairement les à-coups production/consommation.
- Séparer 2 contextes (par exemple, routine de réponse à l'interruption et programme principal).

### 7.9.2. SITUATION COMMUNICATION AVEC DES FIFO

Le diagramme ci-dessous illustre une situation de communication avec un FIFO de réception et un FIFO d'émission. Il illustre aussi la situation des tampons hardware de réception et d'émission.



## 7.10. PRINCIPE DU FIFO

L'abréviation FIFO provient de l'anglais *First In First Out* (à ne pas confondre avec une pile, LIFO, *Last In First Out*).

Un FIFO est donc un élément de stockage temporaire, comme par exemple une série de billes enfilées dans un tube.

Dans la pratique, on ne va pas déplacer les données dans le FIFO mais faire évoluer le pointeur d'écriture et de lecture pour « simuler » cette situation de déplacement des données.

Par FIFO, on sous-entend FIFO circulaire, en ce sens que lorsque l'on atteint la fin du FIFO on recommence au début.

Au niveau de la gestion du FIFO il y a l'aspect production (écriture ou remplissage) et l'aspect consommation (lecture ou vidage).

### 7.10.1. SÉPARATION DES CONTEXTES

Lorsqu'un FIFO est utilisé pour séparer un contexte d'interruption du contexte du programme principal, il faut soigner les fonctions d'écriture et de lecture de telle manière que des erreurs sur la situation du FIFO ne puissent se produire.

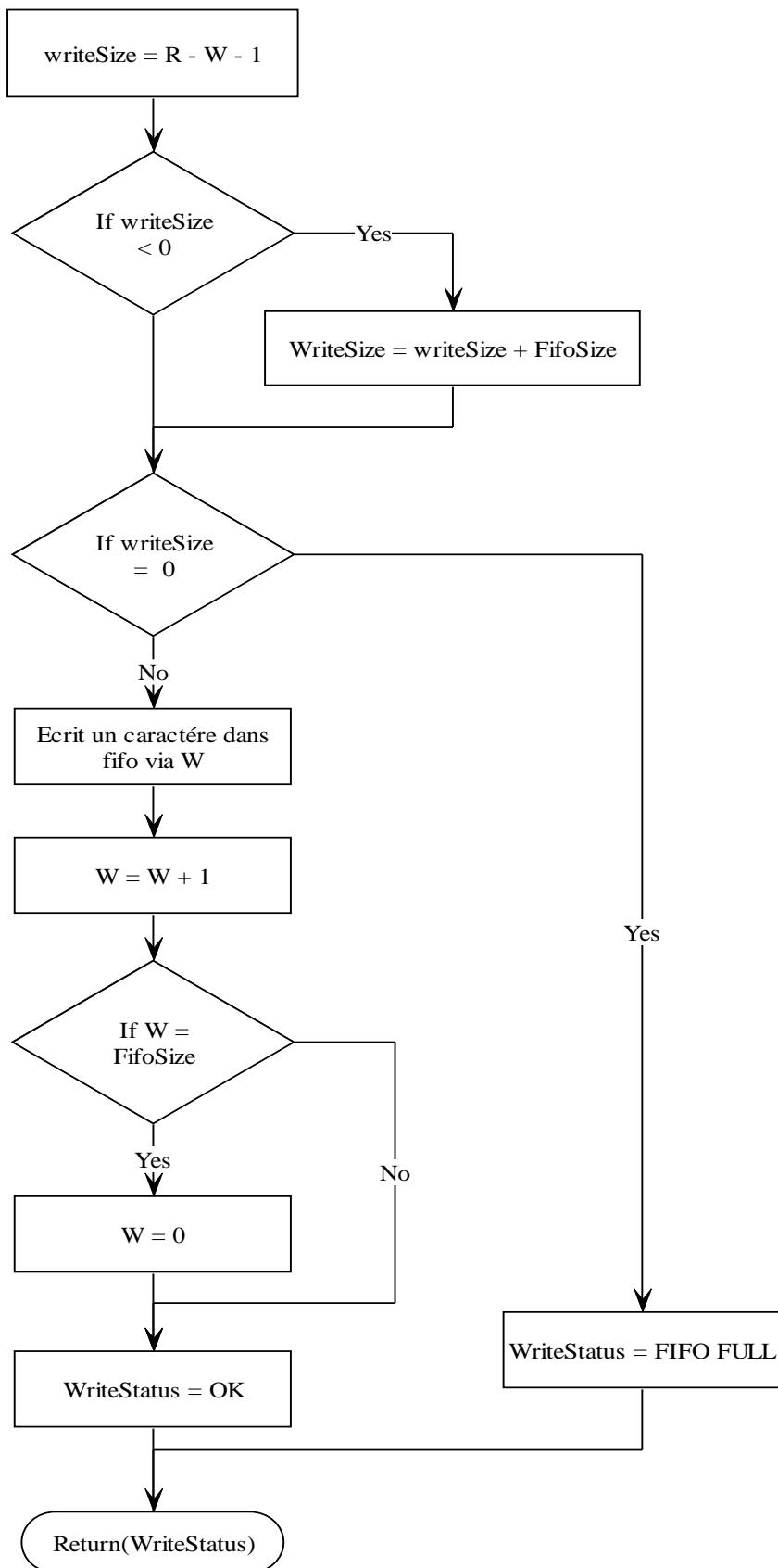
Par exemple lors de la détermination de la situation en lecture, il ne faut lire qu'une seule fois le pointeur d'écriture et le mémoriser dans une variable locale (car l'interruption peut survenir et ajouter un caractère, donc modifier le pointeur pendant le traitement).

### 7.10.2. PRINCIPE DE L'ÉCRITURE DANS LE FIFO

La contrainte lors de l'écriture est de déterminer s'il y a encore de la place dans le FIFO. Si le FIFO est plein (FIFO full), il est interdit d'écrire car on écrase une donnée non encore lue. Le problème est souvent : que faire lorsque le FIFO est plein ?

Remarque : seule la fonction d'écriture a le droit de modifier le pointeur d'écriture, le pointeur de lecture peut être lu pour déterminer la situation.

Voici un organigramme décrivant le principe d'écriture. Dans l'organigramme, **W** représente le pointeur ou l'indice d'écriture, **R** celui de lecture.



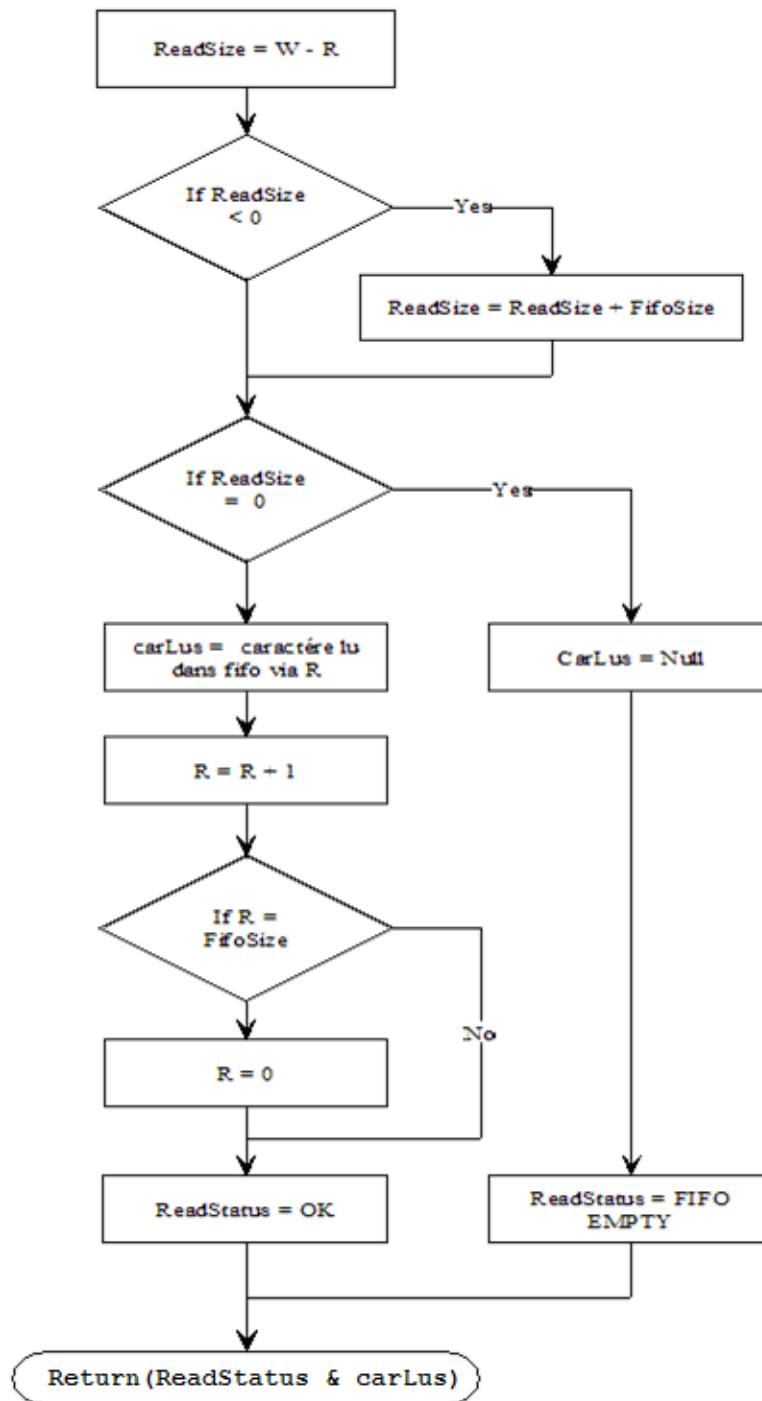
Ecriture dans le FIFO

### 7.10.3. PRINCIPE DE LA LECTURE DANS LE FIFO

La lecture commence toujours par le test de la présence de données dans le FIFO. S'il n'y a pas de donnée, le FIFO est vide (FIFO empty).

Remarque : seule la fonction de lecture a le droit de modifier le pointeur de lecture, le pointeur d'écriture peut être lu pour déterminer la situation.

Voici un organigramme décrivant le principe de lecture dans le FIFO. Dans l'organigramme, R représente le pointeur ou l'indice de lecture, W celui d'écriture.



Lecture dans le FIFO

## 7.11. ANALYSE DU FONCTIONNEMENT D'UN FIFO

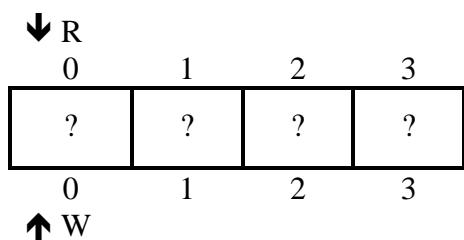
Pour bien comprendre comment fonctionne un FIFO circulaire, nous allons étudier quelques situations sur la base d'un FIFO de 4 caractères. Cela permettra de mieux comprendre les organigrammes fournis précédemment.

↓ R représente l'index ou le pointeur de lecture.

↑ W représente l'index ou le pointeur d'écriture.

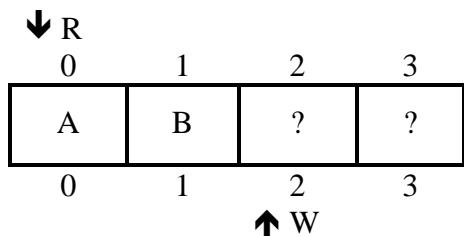
### 7.11.1. SITUATION DE DÉPART

Situation après l'initialisation. Les deux pointeurs sont au début du FIFO.



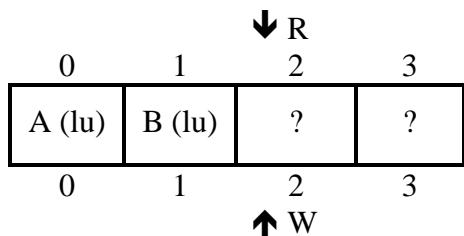
### 7.11.2. SITUATION APRÈS ÉCRITURE DE 2 CARACTÈRES

Situation après l'écriture de 2 caractères. Le nombre de caractères disponibles pour la lecture correspond à  $W - R$  ( $2 - 0 = 2$ ). Si la différence est nulle, cela signifie qu'il n'y a pas de caractères de disponible. Notez que le pointeur d'écriture est géré de la manière suivante : écriture du caractère, puis incrément du pointeur. Il pointe donc sur la prochaine place disponible pour écriture.



### 7.11.3. SITUATION APRÈS LECTURE DES 2 CARACTÈRES

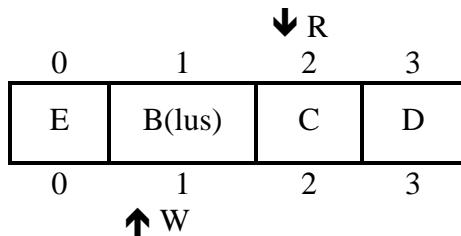
Situation après la lecture des 2 caractères. Le nombre de caractères disponibles pour la lecture correspond à  $W - R$  ( $2 - 2 = 0$ ). Dans ce cas, la différence est nulle, cela signifie qu'il n'y a pas de caractères de disponible. Notez que le pointeur de lecture est géré de la manière suivante : lecture du caractère, puis incrément du pointeur. Il pointe donc sur le prochain caractère disponible pour lecture.



#### 7.11.4. SITUATION DE REBOUCLEMENT

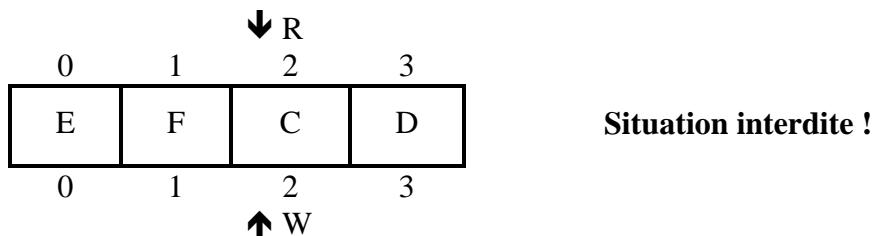
Situation après écriture de 3 caractères supplémentaires (C, D, E). Le nombre de caractères disponibles pour la lecture correspond  $W - R$  ( $1 - 2 = -1$ ). Si la différence est négative, il faut la corriger en ajoutant la taille du FIFO, d'où  $-1 + 4 = 3$ , soit 3 caractères à lire.

En ce qui concerne l'écriture, il faut vérifier la place disponible par la différence des pointeurs, soit  $R - W$ . Dans notre situation  $2 - 1 = 1$ , il est théoriquement encore possible d'écrire 1 caractère.



#### 7.11.5. SITUATION FIFO PLEIN

La situation précédente est déjà une situation de FIFO plein. Car si l'on écrit encore un caractère (F) sans modifier la situation de la lecture, on se trouve dans une situation qui n'est pas admissible :  $R=W$ , ce qui donne une différence nulle donc un FIFO vide côté lecture, alors qu'il y a des caractères à lire.



#### 7.11.6. FIFO PLEIN CONCLUSION

Donc, de cette situation interdite, on conclut les règles suivantes :

- Dans un FIFO de taille **n**, il est possible d'écrire **n-1** caractères.
- Le FIFO est plein si  $R - W - 1 = 0$ .
- Le nombre de caractères que l'on peut écrire est donné par la formule  $R - W - 1$ , lorsque  $R = W = 0$ , on obtient  $0 - 0 - 1 = -1 + 4 = 3$ .

## 7.12. RÉALISATION FIFO AVEC DES POINTEURS

Dans cet exemple, en plus de l'usage de pointeurs, nous allons introduire une structure décrivant un FIFO dans le but d'obtenir des fonctions qui ne soient plus spécifiques à un FIFO, mais au contraire que l'on puisse utiliser pour gérer plusieurs FIFOs.

### 7.12.1. STRUCTURE DÉCRIVANT UN FIFO

```
typedef struct fifo {
    int32_t fifoSize;      // taille du fifo
    int8_t *pDebFifo;     // pointeur sur début du fifo
    int8_t *pFinFifo;     // pointeur sur fin du fifo
    int8_t *pWrite;        // pointeur d'écriture
    int8_t *pRead;         // pointeur de lecture
} S_fifo;
```

Le pointeur de début est nécessaire lorsqu'on détecte le reboulement du FIFO pour réinitialiser le pointeur courant sur le début. Le pointeur de fin permet de détecter le reboulement. La taille du FIFO est utile pour les calculs de nombres d'éléments disponibles.

### 7.12.2. DÉCLARATION D'UN FIFO

Voici la déclaration d'un FIFO, il 'agit de variables globales.

```
// Déclaration du FIFO pour la réception
#define FIFO_RX_SIZE ( (2*8) + 1) // 2 messages

int8_t fifoRX[FIFO_RX_SIZE];
// Déclaration du descripteur du FIFO de réception
S_fifo descrFifoRX;
```

### 7.12.3. INITIALISATION D'UN FIFO

Voici l'initialisation d'un descripteur de FIFO. Cette initialisation est effectuée dans le programme principal. Pour faciliter l'initialisation, une fonction est mise à disposition. Il est nécessaire de connaître l'adresse du descripteur du FIFO, l'adresse du début du FIFO et la taille du FIFO.

```
/*-----*/
/* InitFifo      */
/*-----*/
// Initialisation du descripteur de FIFO
// avec possibilité de fournir une valeur de remplissage
void InitFifo ( S_fifo *pDescrFifo, int32_t FifoSize,
    int8_t *pDebFifo, int8_t InitVal )
{
    int32_t i;
    int8_t *pFif;
    pDescrFifo->fifoSize = FifoSize;
    pDescrFifo->pDebFifo = pDebFifo; // début du fifo
    // fin du fifo
    pDescrFifo->pFinFifo = pDebFifo + (FifoSize - 1);
    pDescrFifo->pWrite = pDebFifo; // début du fifo
```

```

pDescrFifo->pRead      = pDebFifo; // début du fifo
pFif = pDebFifo;
for (i=0; i < FifoSize; i++) {
    *pFif = InitVal;
    pFif++;
}
} /* InitFifo */

```

Appel de la fonction d'initialisation :

```
InitFifo ( &descrFifoRX, FIFO_RX_SIZE, fifoRX, 0 );
```

#### 7.12.4. LA FONCTION GETWRITESPACE

Cette fonction permet de déterminer la place disponible en écriture dans le FIFO. Elle est utilisée dans la fonction d'écriture dans le FIFO et aussi comme test indépendant en vue de la gestion du contrôle de flux.

```

/*-----*/
/* GetWriteSpace */
/*-----*/
// Retourne la place disponible en écriture
int32_t GetWriteSpace ( S_fifo *pDescrFifo)
{
    int32_t writeSize;

    // Détermine le nb de car.que l'on peut déposer
    writeSize = pDescrFifo->pRead - pDescrFifo->pWrite -1;
    if (writeSize < 0) {
        writeSize = writeSize + pDescrFifo->fifoSize;
    }
    return (writeSize);
} /* GetWriteSpace */

```

La détermination du nombre de caractères que l'on peut écrire est réalisée par la différence entre les pointeurs.

#### 7.12.5. LA FONCTION GETREADSIZE

Cette fonction permet de déterminer le nombre de caractères présents dans le FIFO disponibles pour la lecture. Elle est utilisée dans la fonction de lecture du FIFO et aussi comme test indépendant en vue de la gestion du contrôle de flux.

```

/*-----*/
/* GetReadSize */
/*-----*/
// Retourne le nombre de caractères à lire
int32_t GetReadSize ( S_fifo *pDescrFifo)
{
    int32_t readSize;

    readSize = pDescrFifo->pWrite - pDescrFifo->pRead;
}

```

```

        if (readSize < 0) {
            readSize = readSize + pDescrFifo->fifoSize;
        }
        return (readSize);
    } /* GetReadSize */
}

```

La détermination du nombre de caractères que l'on peut lire est réalisée par la différence entre les pointeurs.

### 7.12.6. LA FONCTION PUTCHARINFIFO

Cette fonction permet de déposer un caractère dans le FIFO dont on fournit le descripteur. Cette fonction n'est plus spécifique à un FIFO.

```

/*-----*/
/* PutCharInFifo */
/*=====*/
// Dépose un caractère dans le FIFO
// Retourne 0 si OK, 1 si FIFO full
uint8_t PutCharInFifo ( S_fifo *pDescrFifo,
                        int8_t charToPut )
{
    uint8_t writeStatus;

    // test si fifo est FULL
    if (GetWriteSpace(pDescrFifo) == 0) {
        writeStatus = 1; // fifo FULL
    }
    else {
        // écrit le caractère dans le FIFO
        *(pDescrFifo->pWrite) = charToPut;

        // incrément le pointeur d'écriture
        pDescrFifo->pWrite++;
        // gestion du reboulement
        if (pDescrFifo->pWrite > pDescrFifo->pFinFifo) {
            pDescrFifo->pWrite = pDescrFifo->pDebFifo;
        }

        writeStatus = 0; // OK
    }
    return (writeStatus);
} // PutCharInFifo

```

La détection du reboulement se fait par test du dépassement par pWrite de la fin du FIFO.

### 7.12.7. LA FONCTION GETCHARFROMFIFO

Cette fonction permet d'obtenir un caractère du FIFO dont on fournit le descripteur. Cette fonction n'est plus spécifique à un FIFO.

```

/*-----*/
/* GetCharFromFifo */
/*=====*/
// Obtient (lecture) un caractère du fifo
// retourne 0 si OK, 1 si empty
// le caractère lu est retourné par référence
uint8_t GetCharFromFifo (S_fifo *pDescrFifo, int8_t *carLu)
{
    int32_t readSize;
    uint8_t readStatus;

    // détermine le nb de car. que l'on peut lire
    readSize = GetReadSize(pDescrFifo);

    // test si fifo est vide
    if (readSize == 0) {
        readStatus = 1; // fifo EMPTY
        *carLu = 0;      // carLu = NULL
    }
    else {
        // lis le caractère dans le FIFO
        *carLu = *(pDescrFifo->pRead);

        // incrément du pointeur de lecture
        pDescrFifo->pRead++;
        // gestion du reboulement
        if (pDescrFifo->pRead > pDescrFifo->pFinFifo) {
            pDescrFifo->pRead = pDescrFifo->pDebFifo;
        }
        readStatus = 0; // OK
    }
    return (readStatus);
} // GetCharFromFifo

```

La détection du reboulement se fait par test du dépassement par pRead de la fin du FIFO.

### 7.12.8. GESTION DES FIFO, LIBRAIRIE À DISPOSITION

L'ensemble des fonctions de gestion du FIFO correspondant à ce chapitre est regroupé sur le réseau dans les fichiers **GesFifoTh32.h** et **GesFifoTh32.c** sous:

...\\Maitres-Eleves\\SLO\\Modules\\SL229\_MINF\\PIC32MX\_Utilitaires(PlibHarmony)\\  
Fifo&Antirebond.

## 7.13. EXEMPLE UTILISATION DES FIFOs ET DES INTERRUPTIONS

Cette application utilise la réception et la transmission sous interruption en utilisant un FIFO pour chacune, ainsi que la gestion du contrôle de flux.

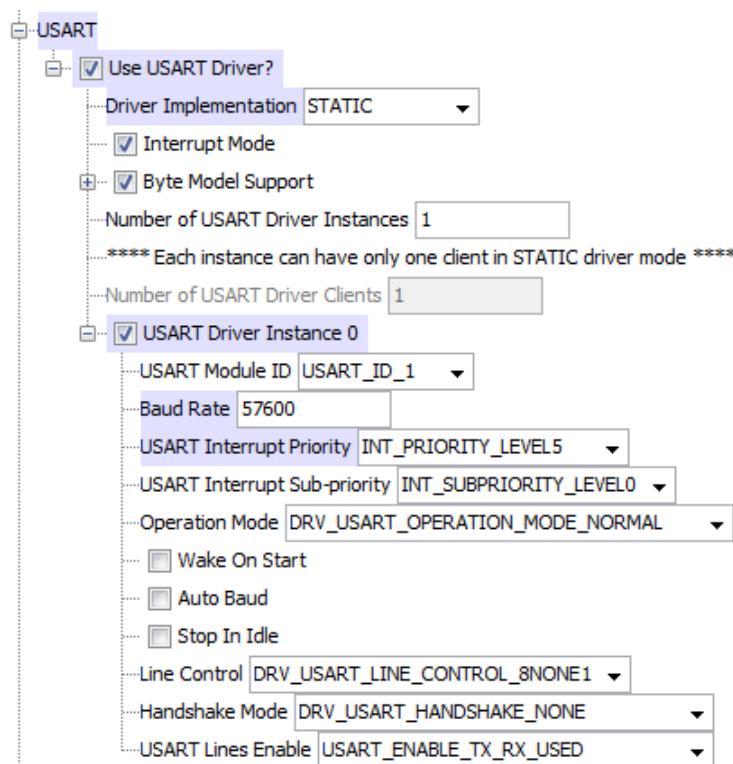
L'objectif de cet exemple est de montrer le comportement de l'UART du PIC32MX qui possède un tampon de réception et un tampon d'émission, comportement en relation avec les FIFOs software et la problématique de la gestion du contrôle de flux par le programme.

Dans cet exemple, nous utilisons un message d'une taille de 6 caractères et nous nous focalisons uniquement sur les mécanismes de communication sans nous préoccuper de comment sont formatés et reconstitués les messages (cet autre aspect fait l'objet d'un exemple complet à la fin de ce chapitre).

### 7.13.1. CONFIGURATION DE L'USART

Voici la configuration de base d'un USART driver au niveau de Harmony 2.05. Des modifications seront réalisées au niveau du code généré pour vérifier l'effet de certains paramètres.

#### 7.13.1.1. CONFIGURATION D'UN USART DRIVER AVEC HARMONY



### 7.13.1.2. CONTENU DE LA FONCTION DRV\_USART0\_INITIALIZE

L'ensemble des actions de configuration de l'USART sont regroupées dans la fonction DRV\_USART0\_Initialize, dont voici le contenu.

```

SYS_MODULE_OBJ DRV_USART0_Initialize(void)
{
    uint32_t clockSource;

    /* Disable the USART module to configure it*/
    PLIB_USART_Disable (USART_ID_1);

    /* Initialize the USART based on configuration
       settings */
    PLIB_USART_InitializeModeGeneral(USART_ID_1,
        false, /*Auto baud*/
        false, /*LoopBack mode*/
        false, /*Auto wakeup on start*/
        false, /*IRDA mode*/
        false); /*Stop In Idle mode*/

    /* Set the line control mode */
    PLIB_USART_LineControlModeSelect(USART_ID_1,
        DRV_USART_LINE_CONTROL_8NONE1);

    /* We set the receive interrupt mode to receive an
       interrupt whenever FIFO is not empty */
    PLIB_USART_InitializeOperation(USART_ID_1,
        USART_RECEIVE_FIFO_ONE_CHAR,
        USART_TRANSMIT_FIFO_IDLE,
        USART_ENABLE_TX_RX_USED);

    /* Get the USART clock source value*/
    clockSource = SYS_CLK_PeripheralFrequencyGet (
        CLK_BUS_PERIPHERAL_1 );

    /* Set the baud rate and enable the USART */
    PLIB_USART_BaudSetAndEnable(USART_ID_1,
        clockSource, 57600); /*Desired Baud rate value*/

    /* Clear the interrupts to be on the safer side*/
    SYS_INT_SourceStatusClear(INT_SOURCE_USART_1_TRANSMIT);
    SYS_INT_SourceStatusClear(INT_SOURCE_USART_1_RECEIVE);
    SYS_INT_SourceStatusClear(INT_SOURCE_USART_1_ERROR);

    /* Enable the error interrupt source */
    SYS_INT_SourceEnable(INT_SOURCE_USART_1_ERROR);

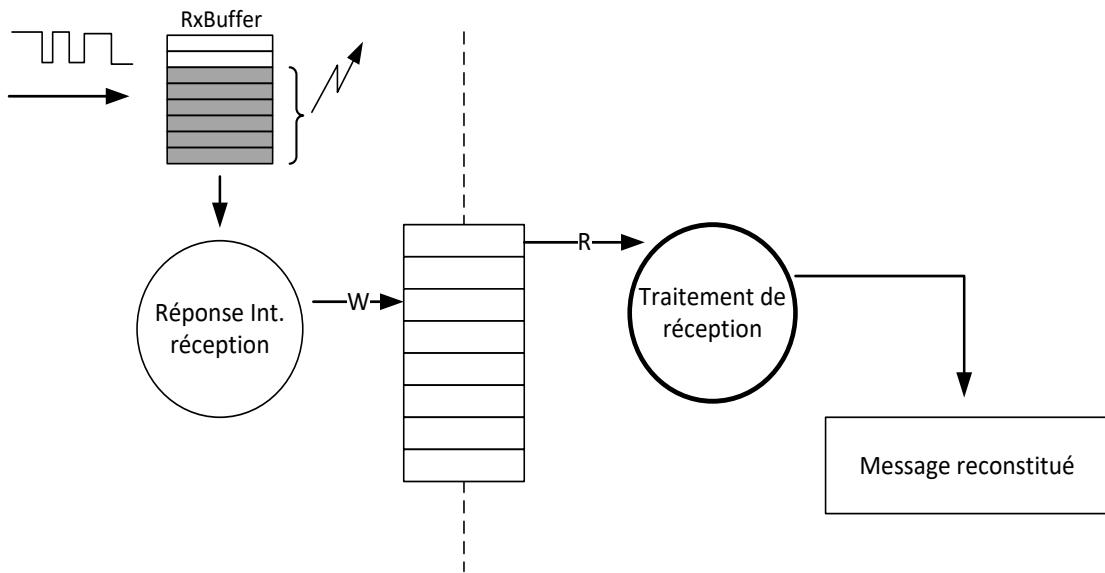
    /* Enable the Receive interrupt source */
    SYS_INT_SourceEnable(INT_SOURCE_USART_1_RECEIVE);

    /* Return the driver instance value*/
    return (SYS_MODULE_OBJ)DRV_USART_INDEX_0;
}

```

### 7.13.2. CONTEXTE DE RÉCEPTION

Voici un diagramme montrant le contexte de réception.



### 7.13.3. DÉTAILS CONFIGURATION INTERRUPTION RÉCEPTION

Voici une partie de la configuration de l'USART, en particulier la section qui configure le comportement de l'interruption de réception :

```
PLIB_USART_InitializeOperation(USART_ID_1,
    USART_RECEIVE_FIFO_HALF_FULL,
    USART_TRANSMIT_FIFO_IDLE,
    USART_ENABLE_TX_RX_USED);
...
SYS_INT_SourceStatusClear(INT_SOURCE_USART_1_RECEIVE);
...
SYS_INT_SourceEnable(INT_SOURCE_USART_1_RECEIVE);
```

Et après l'appel de la fonction DRV\_USART0\_Initialize, on trouve :

```
SYS_INT_VectorPrioritySet(INT_VECTOR_UART1,
    INT_PRIORITY_LEVEL5);
SYS_INT_VectorSubprioritySet(INT_VECTOR_UART1,
    INT_SUBPRIORITY_LEVEL0);
```

### 7.13.3.1. CHOIX D'UN MODE DE DÉCLENCHEMENT DE INT RX

Il y a 3 possibilités pour le déclenchement de l'interruption RX, suivant le niveau de remplissage du buffer de réception :

Members	Description
USART_RECEIVE_FIFO_HALF_FULL	Interrupt when receive buffer is half full
USART_RECEIVE_FIFO_3B4FULL	Interrupt when receive buffer is 3/4 full
USART_RECEIVE_FIFO_ONE_CHAR	Interrupt when a character is received

La taille du tampon hardware de réception est de 8 caractères, donc avec HALF\_FULL on déclenche l'interruption dès la réception de 4 caractères. Pour 3B4FULL c'est lorsque l'on a reçu les  $\frac{3}{4}$  soit 6 caractères. Avec ONE\_CHAR l'interruption est déclenchée à chaque caractère.

Le choix de déclencher l'interruption de réception lorsque le tampon de réception est à moitié plein (donc 4 caractères) aura un comportement particulier avec les paquets de 6 caractères de nos messages.

A l'origine, le code généré contenait le mode USART\_RECEIVE\_FIFO\_ONE\_CHAR. Le configurateur graphique ne permettant pas de modifier ce choix, il faut le faire manuellement dans le code.

#### 7.13.4. SECTION RÉCEPTION DE L'INTERRUPTION DE L'USART

Voici la section réception de l'interruption de l'USART :

```

if ( PLIB_INT_SourceFlagGet(INT_ID_0,
                           INT_SOURCE_USART_1_RECEIVE) &&
    PLIB_INT_SourceIsEnabled(INT_ID_0,
                           INT_SOURCE_USART_1_RECEIVE) ) {
    BSP_LEDToggle(BSP_LED_3); // marque int RX
    // Oui, test si erreur parité ou overrun
    UsartStatus = PLIB_USART_ErrorsGet(USART_ID_1);

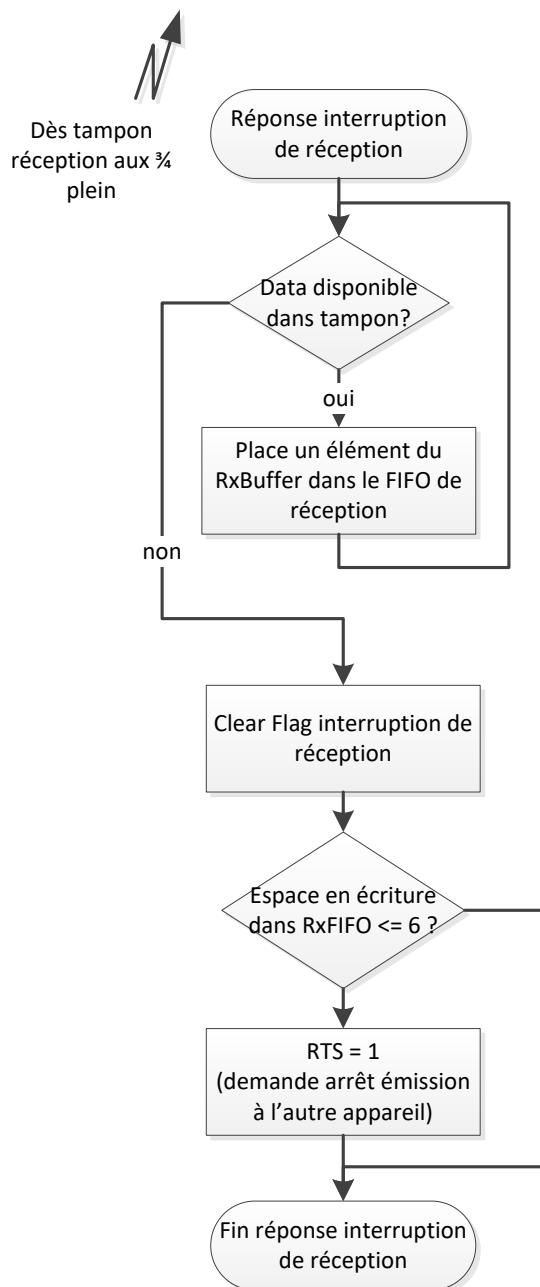
    if ( (UsartStatus & (USART_ERROR_PARITY |
                           USART_ERROR_FRAMING |
                           USART_ERROR_RECEIVER_OVERRUN)) == 0) {
        // Transfert dans le FIFO de tous les chars reçus
        // 1 Si ONE_CHAR, 4 si HALF_FULL et 6 3B4FULL
        while (PLIB_USART_ReceiverDataIsAvailable
                           (USART_ID_1))
    {
        c = PLIB_USART_ReceiverByteReceive(USART_ID_1);
        PutCharInFifo (&descrFifoRX, c);
        BSP_LEDToggle(BSP_LED_5); // pour comptage
    }
    // buffer is empty, clear interrupt flag
    PLIB_INT_SourceFlagClear(INT_ID_0,
                           INT_SOURCE_USART_1_RECEIVE);
} else {
    // La lecture des erreurs les efface
    // sauf pour overrun
    if ((UsartStatus & USART_ERROR_RECEIVER_OVERRUN)
        == USART_ERROR_RECEIVER_OVERRUN) {
        PLIB_USART_ReceiverOverrunErrorClear(USART_ID_1);
    }
}

// Traitement du contrôle de flux
freeSize = GetWriteSpace (&descrFifoRX);
if (freeSize <= 6) // a cause d'un int pour 6 char
{
    // Demande de ne plus émettre
    RS232 RTS = 1;
    if (freeSize == 0) {
        ErrFiFoFull = 1; // pour debugging
    }
}
} // end if RX

```

### 7.13.5. PRINCIPE TRAITEMENT INTERRUPTION DE RÉCEPTION

Le principe du traitement de l'interruption est le suivant :



Comme la gestion du contrôle de flux est supposée garantir que le FIFO de réception permette l'enregistrement de 6 caractères, il est possible de réaliser une boucle de copie du tampon de réception HW dans le RxFifo software sans test de la situation du FIFO soft.

La situation du FIFO en écriture est testée après. Si la place disponible est  $\leq$  à 6 alors il faut activer RTS à high pour demander l'arrêt de l'émission.

Cet organigramme et les quelques suivants incluent une gestion du contrôle de flux par software. Une gestion automatique par hardware est également possible.

Le traitement du contrôle de flux dans la réponse à l'interruption, qui consiste à imposer RTS = 1 pour demander à l'autre appareil de stopper l'émission, implique que le RTS doit être remis à 0 hors de l'interruption, lorsqu'il y a à nouveau suffisamment de place dans le FIFO software.

☞ La valeur de 6 est liée au réglage ¾ full de l'interruption de réception, cela correspond aussi à la taille d'un message.

### 7.13.6. COMPORTEMENT INTERRUPTION RX AVEC HALF\_FULL

Voici l'observation du comportement de l'interruption de réception avec le réglage HALF\_FULL. Obtenu par :

```
PLIB_USART_InitializeOperation(USART_ID_1,
    USART_RECEIVE_FIFO_HALF_FULL,
    USART_TRANSMIT_FIFO_IDLE,
    USART_ENABLE_TX_RX_USED);
```

canal 1: LED\_0  
(marque traitement dans l'application)

canal 2: LED\_3  
(inversion à chaque int RX)

canal 3 : broche 52 U1RX

canal 4: LED\_5  
(inversion dans boucle copie)



On constate 3 interruptions pour le même paquet de 12 caractères fournis par l'application Windows.

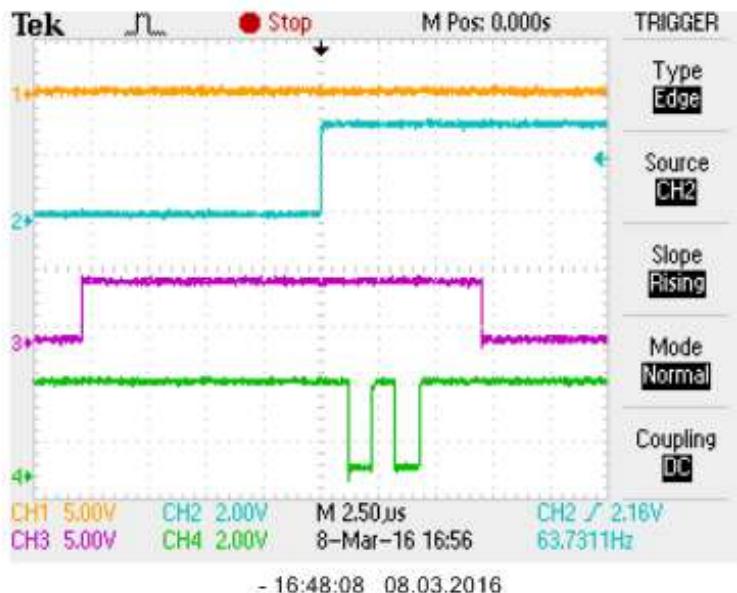
Et avec l'observation détaillée :

canal 1: LED\_0  
(marque traitement dans l'application)

canal 2: LED\_3  
(inversion à chaque int RX)

canal 3 : broche 52 U1RX

canal 4: LED\_5  
(inversion dans boucle copie)



Et on observe bien les 4 transitions indiquant l'extraction de 4 caractères du tampon de réception de l'USART.

### 7.13.7. COMPORTEMENT INTERRUPTION RX AVEC 3B4FULL

Voici l'observation du comportement de l'interruption de réception avec le réglage 3B4\_FULL. Obtenu par :

```
PLIB_USART_InitializeOperation(USART_ID_1,
    USART_RECEIVE_FIFO_3B4FULL,
    USART_TRANSMIT_FIFO_IDLE,
    USART_ENABLE_TX_RX_USED);
```

canal 1: LED\_0  
(marque traitement dans l'application)

canal 2: LED\_3  
(inversion à chaque int. RX)

canal 3 : broche 52 U1RX

canal 4: LED\_5  
(inversion dans boucle copie)



On constate 2 interruptions pour le même paquet de 12 caractères fournis par l'application Windows.

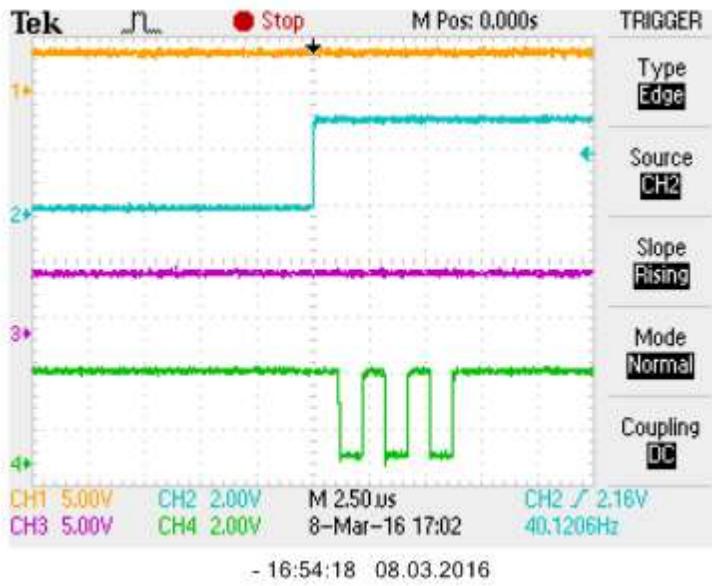
Et avec l'observation détaillée :

canal 1: LED\_0  
(marque traitement dans l'application)

canal 2: LED\_3  
(inversion à chaque int RX)

canal 3 : broche 52 U1RX

canal 4: LED\_5  
(inversion dans boucle copie)



Et on observe bien les 6 transitions indiquant l'extraction de 6 caractères du tampon de réception de l'USART.

### 7.13.8. COMPORTEMENT INTERRUPTION RX AVEC ONE\_CHAR

Voici l'observation du comportement de l'interruption de réception avec le réglage **ONE\_CHAR**. Obtenu par :

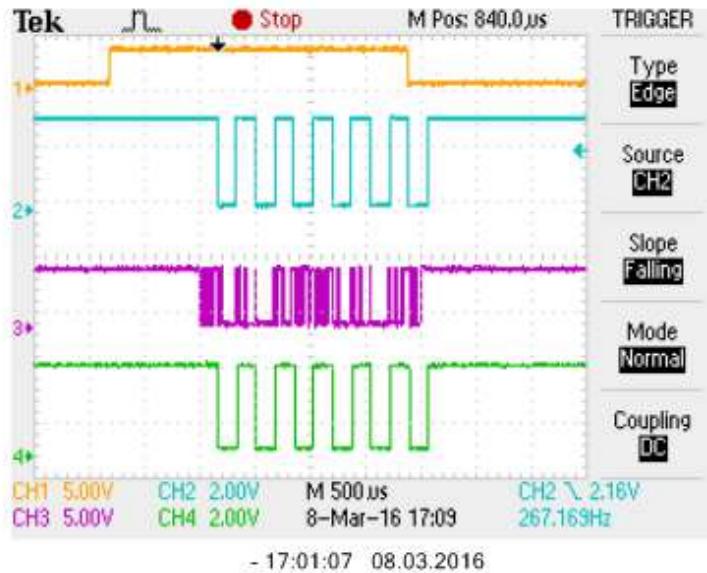
```
PLIB_USART_InitializeOperation(USART_ID_1,
    USART_RECEIVE_FIFO_ONE_CHAR,
    USART_TRANSMIT_FIFO_IDLE,
    USART_ENABLE_TX_RX_USED);
```

canal 1: LED\_0  
(marque traitement dans l'application)

canal 2: LED\_3  
(inversion à chaque int RX)

canal 3 : broche 52 U1RX

canal 4: LED\_5  
(inversion dans boucle copie)



On constate 12 interruptions pour le même paquet de 12 caractères fournis par l'application Windows.

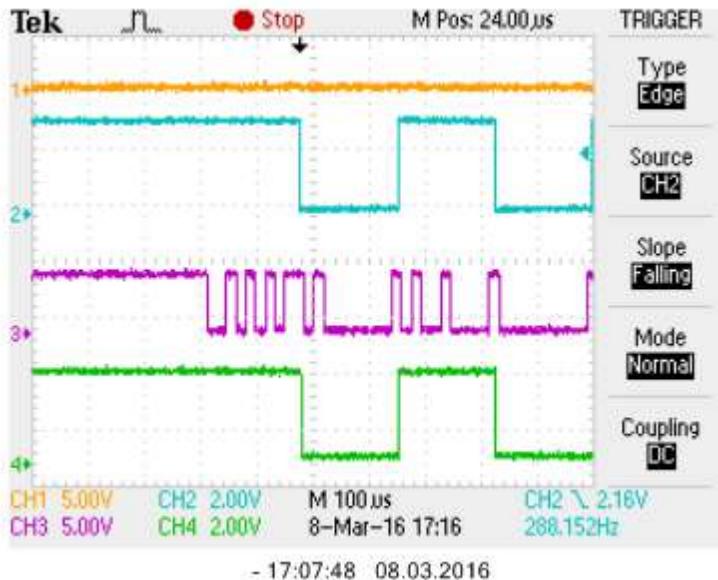
Et avec l'observation détaillée :

canal 1: LED\_0  
(marque traitement dans l'application)

canal 2: LED\_3  
(inversion à chaque int RX)

canal 3 : broche 52 U1RX

canal 4: LED\_5  
(inversion dans boucle copie)



Et on observe qu'après la réception d'un seul caractère, on obtient une interruption et une action de copie du tampon de réception de l'USART.

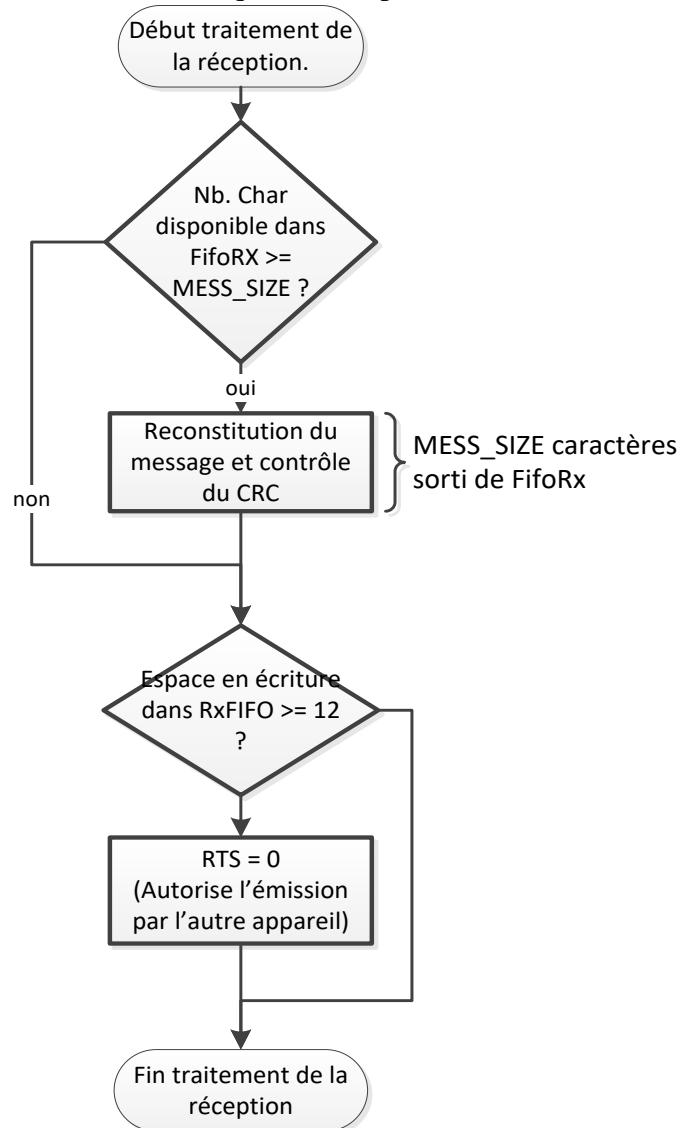
:( Ce mode est peu performant; il génère inutilement des interruptions. Mais il peut s'avérer nécessaire suivant le cas de figure de transmission.

### 7.13.9. PRINCIPE DU TRAITEMENT DE LA RÉCEPTION (GETMESSAGE)

Dans le traitement de la réception au niveau de la partie application (fonction GetMessage), il y a deux aspects :

- La gestion du FIFO software (disponibilité d'un message et à l'opposé place disponible pour les nouveaux messages, ceci en relation avec la gestion du control de flux).
- La reconstitution de message et le contrôle de son CRC.

Dans ce qui suit, nous ne traiterons que le 1<sup>er</sup> aspect.



#### 7.13.9.1. TEST DISPONIBILITÉ DU MESSAGE DANS RXFIFO

Pour faciliter la reconstitution du message, on teste la situation de remplissage du RxFifo. Si le FIFO contient au minimum le nb de caractères correspondant à la taille du message, alors on essaie de traiter le message. Cela permet une lecture du FIFO en boucle sans nécessité de tester la situation du FIFO à chaque appel de **GetCharFromFifo**.

Voici le code correspondant :

```
// Détermine le nombre de caractères à lire
NbCharToRead = GetReadSize ( &descrFifoRX);

// Si >= taille message alors traite
if (NbCharToRead >= MESS_SIZE) {

    EndMess = 0;
    while ( (NbCharToRead >= 1) && ( EndMess == 0) ) {

        // Lis un caractère sans gestion du status
        GetCharFromFifo ( &descrFifoRX, &RxC );
        NbCharToRead--;
        // Traitement selon situation du message
        switch (GetSituation) {
```

#### 7.13.9.2. TRAITEMENT DU CONTRÔLE DE FLUX EN RÉCEPTION

Il faut se souvenir que dans la réponse à l'interruption, on bloque l'émission par l'autre appareil via RTS = 1.

Si on n'agit pas pour rétablir l'émission (RTS = 0) l'autre appareil n'émet plus. Par contre, il est nécessaire de rétablir l'émission lorsque l'on dispose d'un peu de marge d'où le 12 = 2 \* 6. Pour autoriser l'émission lorsqu'on a la place pour 2 paquets d'une taille de 6 octets chacun.

¶ La valeur de 6 est liée au réglage ¾ full de l'interruption de réception qui correspond par hasard à la taille du message.

Voici le code correspondant:

```
// Test si place en écriture pour 2 paquets de 6 char
if (GetWriteSpace ( &descrFifoRX ) >= 12) {

    // autorise émission par l'autre
    RS232_RTS = 0;
}
```

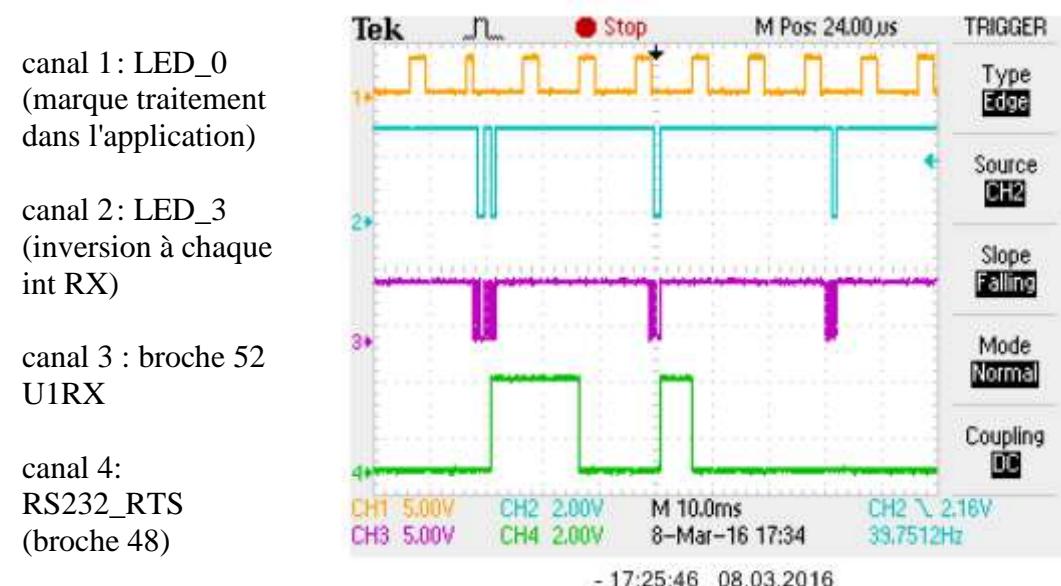
### 7.13.10. VÉRIFICATION CONTRÔLE DE FLUX DE RÉCEPTION

Pour vérifier la gestion de la ligne RTS, nous devons créer une situation où le débit des messages émis dépasse celui de réception, idéalement par à-coup. Obtenu en modifiant l'application pour un envoi un cycle sur 3 de 4 messages.

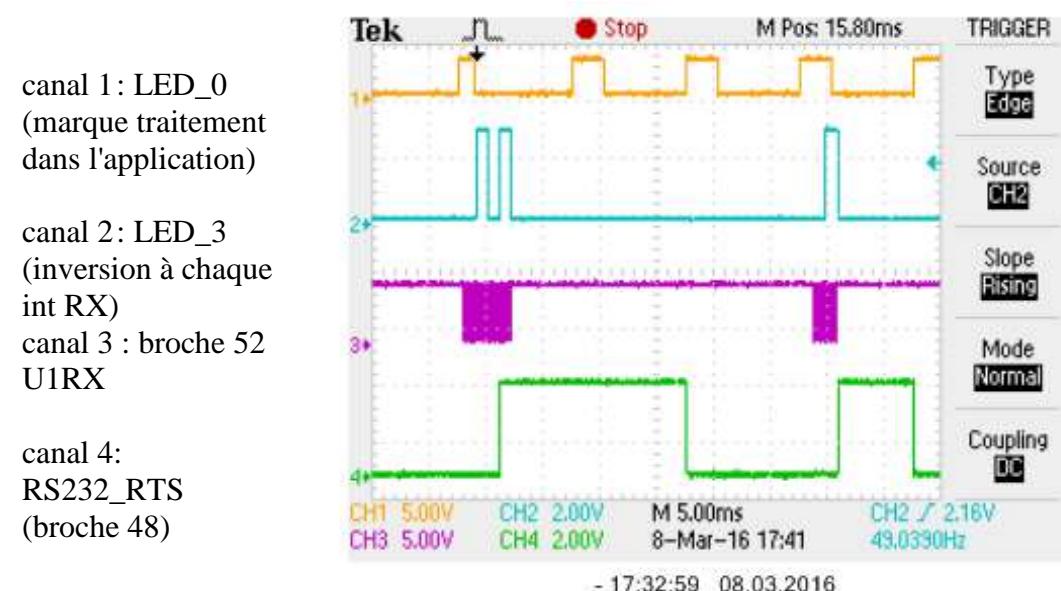
Nous observons la ligne RTS en relation avec l'interruption de réception ainsi que la broche RX. Nous conservons l'observation de la LED\_0 pour le cycle d'exécution de l'application.

⌚ Observation en 3B4FULL pour l'interruption de réception.

#### 7.13.10.1. VUE D'ENSEMBLE CONTRÔLE FLUX RÉCEPTION



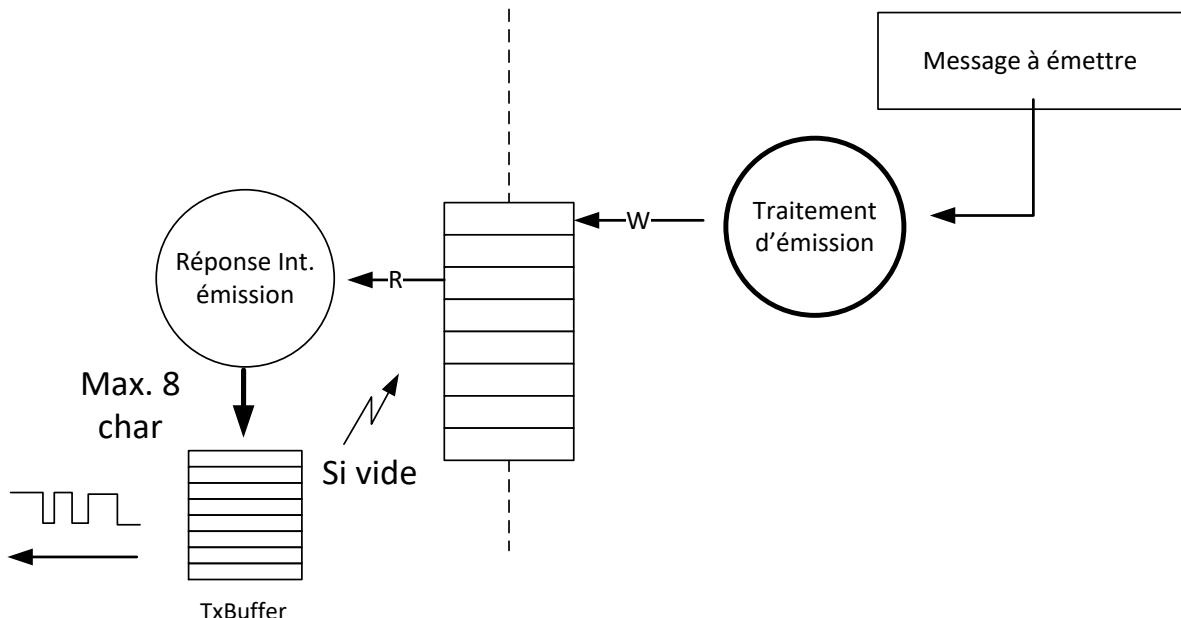
#### 7.13.10.1. VUE DE DÉTAIL CONTRÔLE FLUX RÉCEPTION



On peut observer que suite à l'arrivée du bloc de 4 messages, la ligne RTS est mise à 1 et qu'elle n'est remise à 0 qu'après 2 cycles de traitement de l'application.

### 7.13.11. CONTEXTE D'ÉMISSION

Le diagramme ci-dessous décrit le contexte d'émission.



En émission, nous obtenons une interruption d'émission lorsque le tampon d'émission (TxBuffer) est vide. Il est possible de le remplir jusqu'à ce qu'il soit plein, soit un maximum de 8 caractères.

### 7.13.12. DÉTAIL DE LA CONFIGURATION DE L'INTERRUPTION D'ÉMISSION

Voici les éléments de la configuration de l'USART qui concernent l'interruption d'émission :

```

PLIB_USART_InitializeOperation(USART_ID_1,
    USART_RECEIVE_FIFO_HALF_FULL,
    USART_TRANSMIT_FIFO_EMPTY,
    USART_ENABLE_TX_RX_USED);
...
SYS_INT_SourceStatusClear(INT_SOURCE_USART_1_TRANSMIT);

```

Et après l'appel de la fonction DRV\_USART0\_Initialize, on trouve :

```

SYS_INT_VectorPrioritySet(INT_VECTOR_UART1,
    INT_PRIORITY_LEVEL5);
SYS_INT_VectorSubprioritySet(INT_VECTOR_UART1,
    INT_SUBPRIORITY_LEVEL0);

```

Le code généré n'active pas l'interruption d'émission, ce qui est logique. Cela devra être fait par l'application lorsqu'une émission sera souhaitée.

Concernant le comportement de l'interruption d'émission selon le niveau de remplissage du buffer, les choix disponibles sont :

Members	Description
USART_TRANSMIT_FIFO_EMPTY	Interrupt when the transmit buffer becomes empty
USART_TRANSMIT_FIFO_IDLE	Interrupt when all characters are transmitted
USART_TRANSMIT_FIFO_NOT_FULL	Interrupt when at least one location is empty in the transmit buffer

**Tout comme pour l'interruption de réception, le configrateur graphique ne permet pas ce choix. Il faut le faire manuellement dans le code. A l'origine, le code généré contenait le mode USART\_TRANSMIT\_FIFO\_IDLE.**

### 7.13.13. RÉPONSE À L'INTERRUPTION D'ÉMISSION

L'interruption d'émission a été configurée pour se produire lorsque le tampon d'émission est vide. Dans cette situation il est possible de réaliser une boucle de transfert entre le FIFO d'émission software et le tampon hardware d'émission.

```

// Is this an TX interrupt ?
if ( PLIB_INT_SourceFlagGet(INT_ID_0,
                               INT_SOURCE_USART_1_TRANSMIT) &&
     PLIB_INT_SourceIsEnabled(INT_ID_0,
                               INT_SOURCE_USART_1_TRANSMIT) ) {
    BSP_LEDToggle(BSP_LED_4); // marque int TX

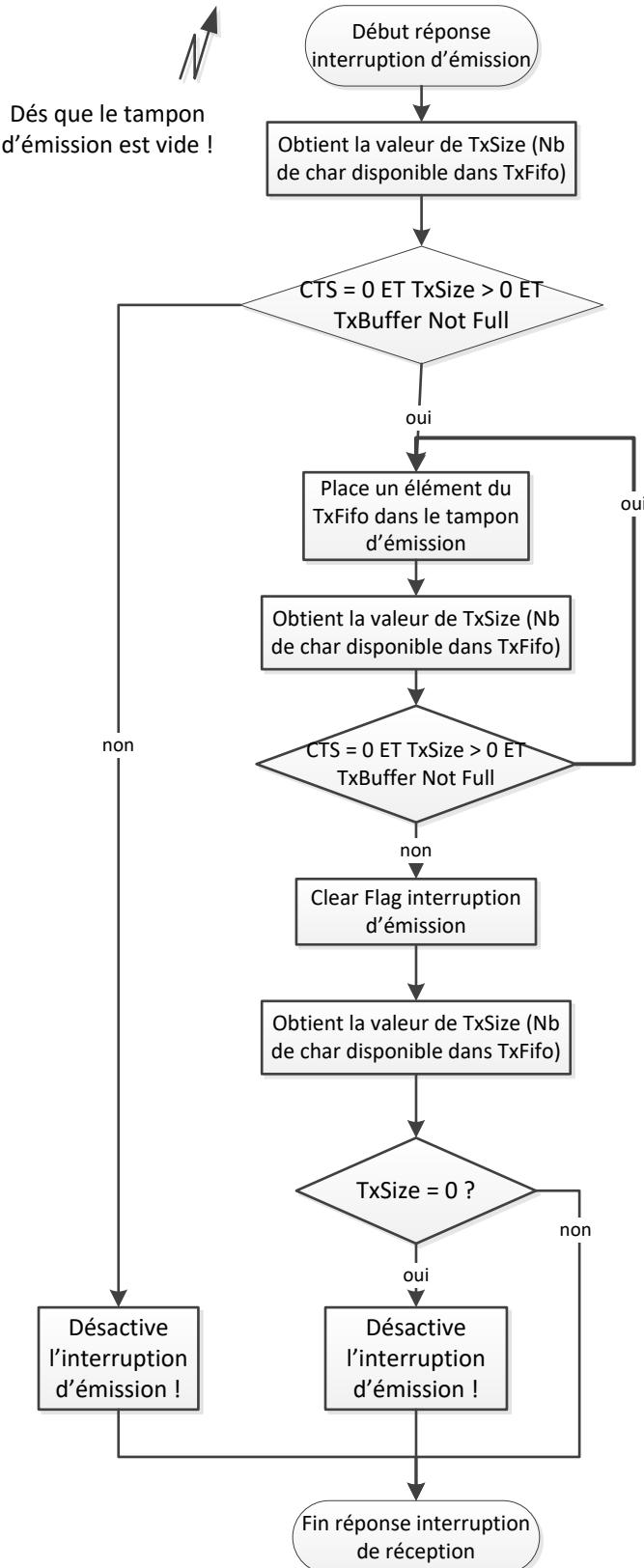
    // On vérifie 3 conditions :
    // Si CTS = 0 (autorisation d'émettre)
    // Si il y a des caractères à émettre
    // Si le TxBuffer n'est pas plein
    i_cts = RS232_CTS;
    TXsize = GetReadSize (&descrFifoTX);
    TxBuffFull =
        PLIB_USART_TransmitterBufferIsFull(USART_ID_1);
    if ( (i_cts == 0) && ( TXsize > 0 ) &&
        TxBuffFull == false ) {
        do {
            GetCharFromFifo(&descrFifoTX, &c);
            PLIB_USART_TransmitterByteSend(USART_ID_1, c);
            BSP_LEDToggle(BSP_LED_6); // pour comptage
            i_cts = RS232_CTS;
            TXsize = GetReadSize (&descrFifoTX);
            TxBuffFull =
                PLIB_USART_TransmitterBufferIsFull(USART_ID_1);
        } while ( (i_cts == 0) && ( TXsize > 0 ) &&
                  TxBuffFull == false );
    }

    // Clear the TX interrupt Flag
    // (Seulement après TX)
    PLIB_INT_SourceFlagClear(INT_ID_0,
                             INT_SOURCE_USART_1_TRANSMIT);
    TXsize = GetReadSize (&descrFifoTX);
    if (TXsize == 0) {
        // pour éviter une interruption inutile
        PLIB_INT_SourceDisable(INT_ID_0,
                               INT_SOURCE_USART_1_TRANSMIT);
    }
} else {
    // disable TX interrupt
    PLIB_INT_SourceDisable(INT_ID_0,
                           INT_SOURCE_USART_1_TRANSMIT);
}
}

```

### 7.13.14. PRINCIPE DE L'INTERRUPTION D'ÉMISSION

Voici un diagramme qui décrit le principe de l'interruption d'émission.



Pour émettre un caractère, il faut que  $CTS=0$  (autorisation d'émettre) et qu'il y ait des caractères dans le FIFO. Il faut également que le tampon d'émission ne soit pas plein.

Il est possible de réaliser une boucle prenant un caractère du FIFO d'émission (TxFifo) et le déposant dans le tampon d'émission (TxBuffer).

Dès qu'une des conditions n'est plus vraie (TxBuffer full ou plus de caractère dans le TxFifo), on quitte la boucle.

Le flag d'interruption ne doit être mis à 0 que lorsque que l'on a rempli le TxBuffer.

Pour éviter une interruption inutile, il faut désactiver l'interruption d'émission si le FIFO d'émission est vide.

Si les conditions ne permettent pas de placer des caractères dans le TxBuffer, il faut impérativement désactiver l'interruption. Ceci évite d'avoir en permanence l'interruption active alors que l'on n'a rien à émettre.

#### 7.13.14.1. GESTION DU CONTROLE DE FLUX DANS INTERRUPTION D'ÉMISSION

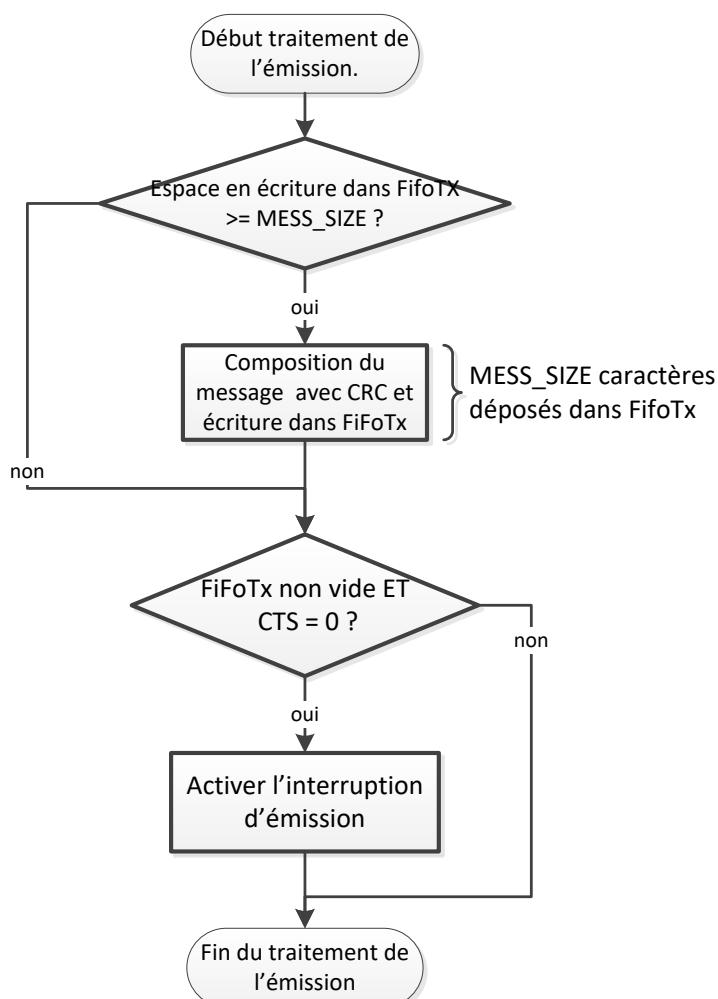
Le traitement du contrôle de flux dans la réponse à l'interruption, qui consiste à inhiber l'interruption d'émission lorsque CTS n'est pas égal à 0, implique que l'interruption d'émission doit être autorisée à nouveau **hors de l'interruption**, lorsqu'il y a des caractères dans le FIFO d'émission et que la ligne CTS vaut 0.

#### 7.13.15. PRINCIPE DU TRAITEMENT DE L'ÉMISSION (SENDMESSAGE)

Dans le traitement de l'émission, au niveau de la partie application (fonction SendMessage), il y a deux aspects :

- La gestion du FIFO software d'émission (place pour déposer un message et à l'opposé contenu à émettre, ceci en relation avec la gestion du contrôle de flux).
- La composition du message et le calcul de son CRC.

Dans ce qui suit, nous ne traiterons que le 1<sup>er</sup> aspect.



S'il y a assez de place dans le FIFO d'émission, on y dépose un message complet.

Après la tentative d'écriture dans le FIFO, si le nombre de caractères placé dans le FIFO est > 0 ET que CTS = 0, il y a autorisation de l'interruption d'émission, ce qui permettra l'émission des caractères.

### 7.13.15.1. CODE PARTIEL DU MÉCANISME D'ÉMISSION

Voici le code partiel du mécanisme d'émission au niveau application (SendMessage) :

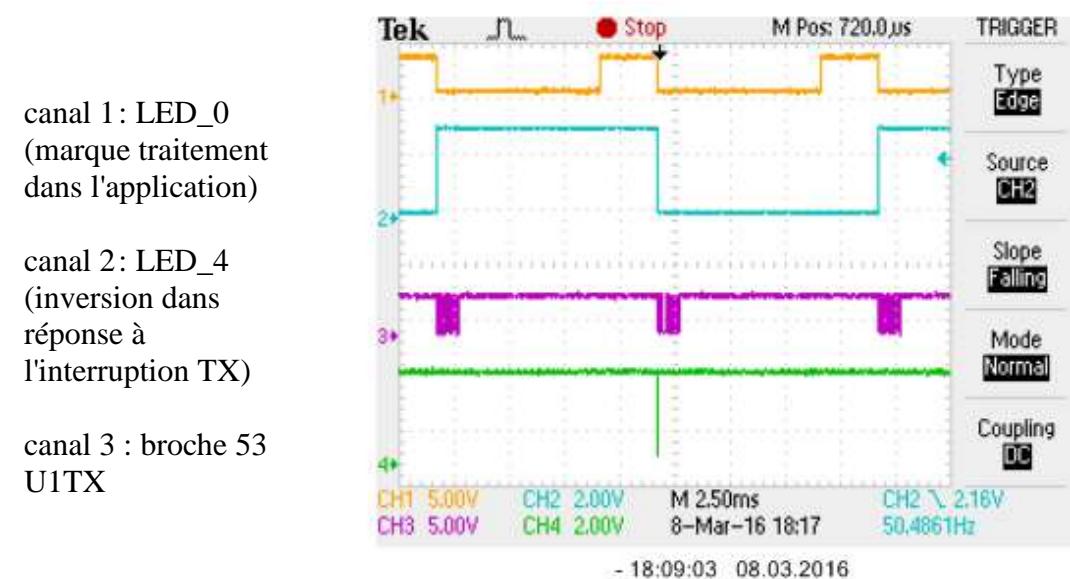
```
// Test si place pour écrire 1 message
FreeSize = GetWriteSpace ( &descrFifoTX);
if (FreeSize >= (MESS_SIZE) ) {

    // Compose le message
    // Ajoute le CRC au message
    // Dépose le message dans le FIFO d'émission
}
// Gestion du control de flux
// Si on a un caractère à envoyer et que CTS = 0
FreeSize = GetReadSize(&descrFifoTX);
if ((RS232_CTS == 0) && (FreeSize > 0))
{
    // Autorise int émission
    PLIB_INT_SourceEnable(INT_ID_0,
                          INT_SOURCE_USART_1_TRANSMIT);
}
}
```

### 7.13.16. OBSERVATION DE L'ÉMISSION

#### 7.13.16.1. VUE D'ENSEMBLE

La situation est la suivante :

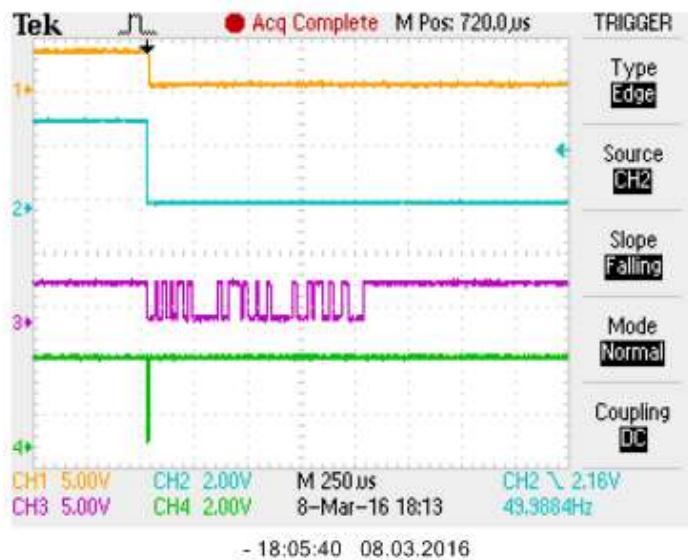


On peut observer qu'à la fin du traitement dans l'application, on voit apparaître une interruption de transmission et que sur la broche TX on voit apparaître la transmission des caractères.

Le message émis étant de 6 caractères et le tampon d'émission ayant une taille de 8, nous avons une seule interruption à l'occasion de laquelle il est possible de placer les 6 caractères dans le tampon.

### 7.13.16.2. VUE DE DÉTAIL

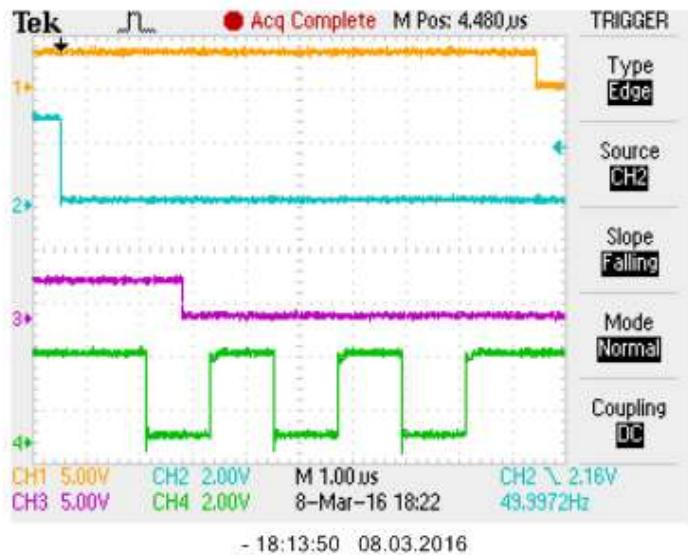
- canal 1: LED\_0  
(marque traitement dans l'application)
- canal 2: LED\_4  
(inversion dans réponse à l'interruption TX)
- canal 3 : broche 53 U1TX
- canal 4: LED\_6  
(inversion dans réponse à l'interruption, dans la boucle de dépôt dans le tampon d'émission)



### 7.13.16.3. CHECK DES 6 CARACTÈRES

On vérifie que l'on a bien transféré 6 caractères dans le tampon hardware.

- canal 1: LED\_0  
(marque traitement dans l'application)
- canal 2: LED\_4  
(inversion dans réponse à l'interruption)
- canal 3 : broche 53 U1TX
- canal 4: LED\_6  
(inversion dans réponse à l'interruption, dans la boucle de dépôt dans le tampon d'émission)



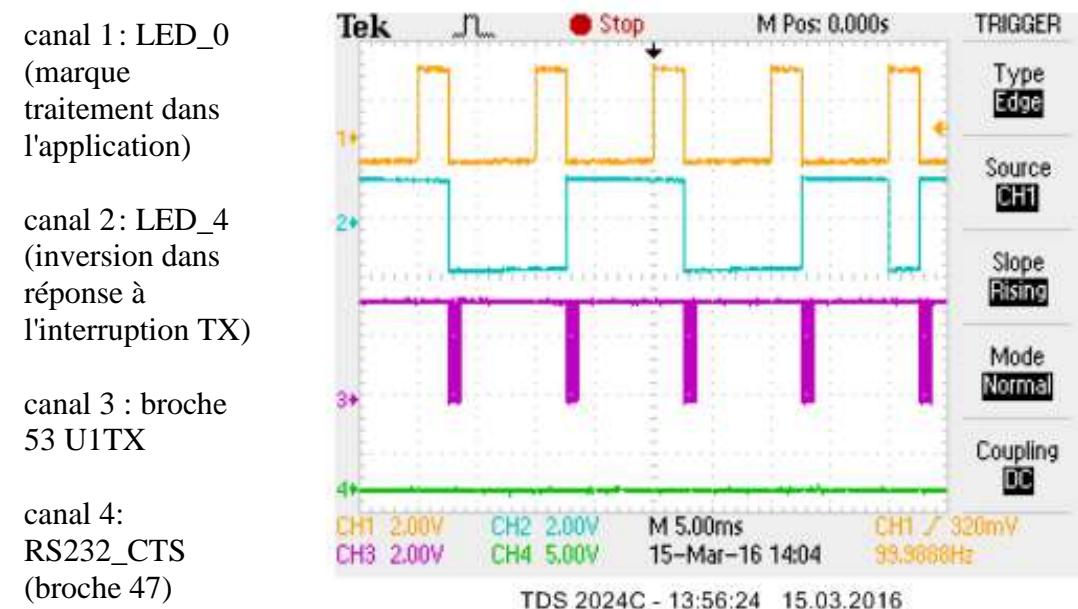
### 7.13.17. VÉRIFICATION DU CONTRÔLE DE FLUX À L'ÉMISSION

Nous allons vérifier si notre système cesse d'émettre lorsque l'entrée CTS n'est plus au niveau bas. Pour vérifier cela, nous devons créer une situation avec une émission par salves (par exemple en appelant 2 fois SendMessage en plus tous les 3 cycles).

Nous observons la ligne CTS en relation avec l'interruption d'émission ainsi que la broche TX. Nous conservons l'observation de la LED\_0 pour le cycle d'exécution de l'application.

#### 7.13.17.1. VUE SANS ACTION DU CONTRÔLE FLUX

Situation sans l'envoi de trame supplémentaire.

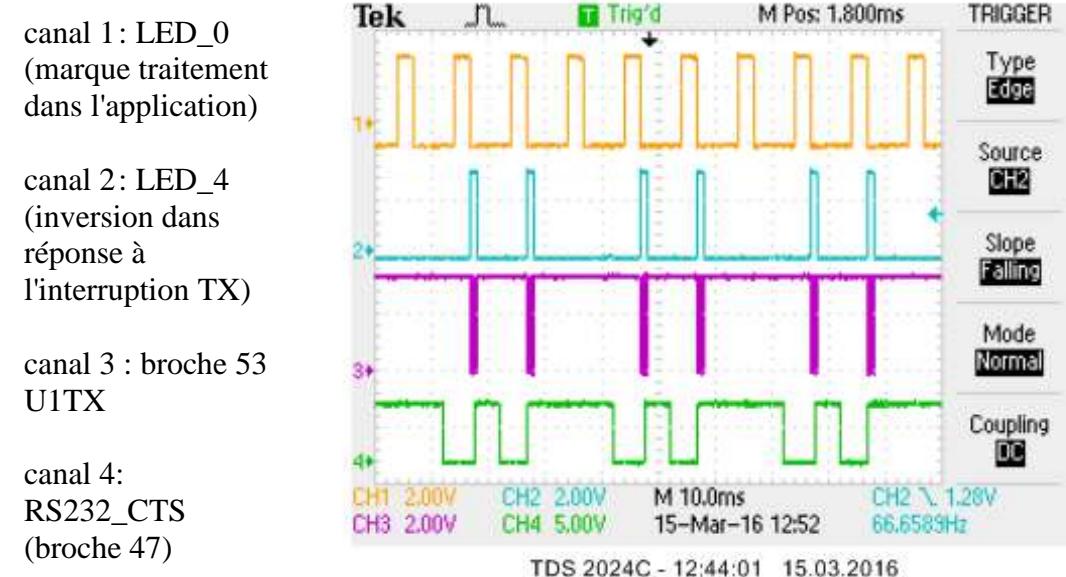


A chaque cycle application (10 ms), envoi d'un message. Le mécanisme de réception supporte ce débit.

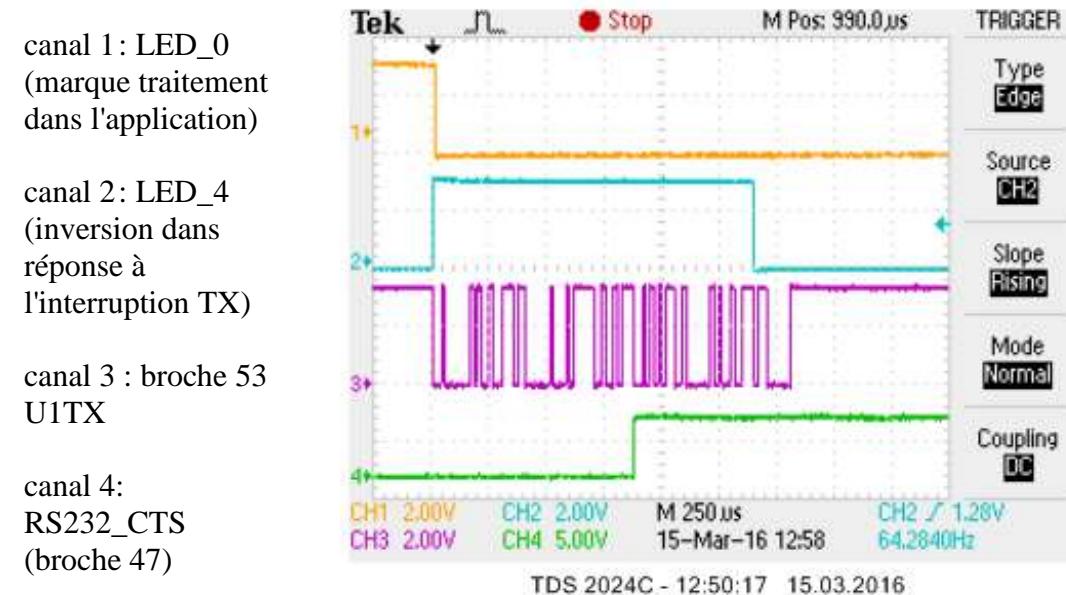
### 7.13.17.2. VUE D'ENSEMBLE CONTRÔLE FLUX D'ÉMISSION

Avec l'ajout de trames supplémentaires, on aboutit à une situation d'accumulation des messages à envoyer. On constate l'envoi d'un message par cycle application mais avec 2 interruptions et d'un cycle sans envoi à cause du CTS à 1.

Il est possible d'observer la situation du signal CTS, donc la réaction du système récepteur.



### 7.13.17.1. VUE DE DÉTAIL DU CONTRÔLE FLUX D'ÉMISSION



On peut observer que dès que le signal CTS passe à 1, la 2<sup>ème</sup> interruption d'émission ne produit plus de transmission de caractères.

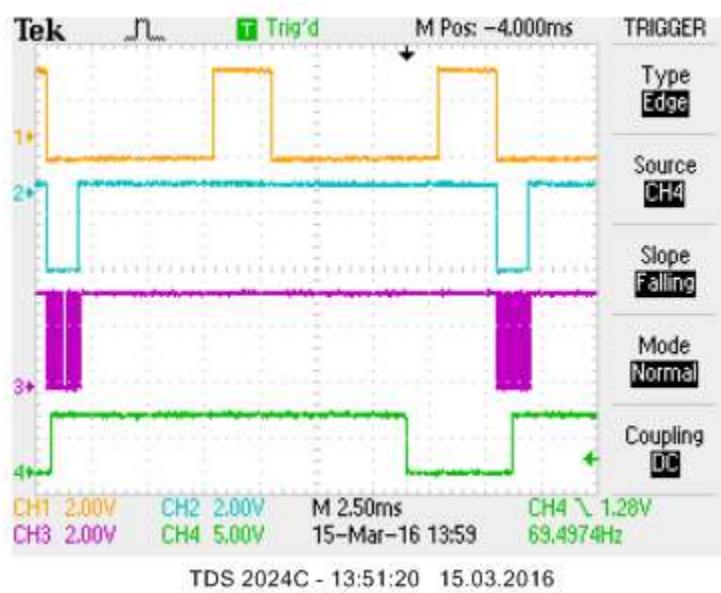
### 7.13.17.2. VUE DE LA REPRISE DE L'ÉMISSION

canal 1: LED\_0  
(marque traitement dans l'application)

canal 2: LED\_4  
(inversion dans réponse à l'interruption TX)

canal 3 : broche 53 U1TX

canal 4:  
RS232\_CTS  
(broche 47)



On observe une situation avec 1 cycle application ne produisant pas de transmission car CTS = 1, puis la réapparition de l'émission qui est à nouveau stoppée.

## 7.14. STRUCTURE D'UN MESSAGE

La structure d'un message n'est pas forcement de taille fixe. Bien souvent, on est amené à créer des messages de taille variable. Et si la fiabilité de la liaison n'est pas bonne (par exemple transmission sans fil), on ajoutera au message une somme de contrôle (Checksum) ou un CRC.

Pour faciliter le l'analyse du message (appelé *parsing* en anglais), on ajoute un caractère de début du message STX et un caractère de fin du message ETX. Un champ du message est chargé d'indiquer la longueur des données utiles.

Exemple de structure :

STX	LEN	DATA	CRC	ETX
-----	-----	------	-----	-----

Lors de la réception d'un tel message, il est nécessaire d'attendre le STX, puis le byte de longueur (LEN). Ensuite il est nécessaire de lire les N bytes de DATA, N étant donné par la valeur du champ LEN.

Ensuite, s'il s'agit d'une valeur de CRC sur 16 bits, il y aura lecture des deux caractères donnant la valeur du CRC. En général le CRC est calculé au fur et à mesure de la lecture. La valeur calculée est comparée à la valeur fournie dans le message ou si on a pris certaines précautions, la valeur calculée y compris le CRC doit valoir zéro. Il est important de définir si la valeur de CRC correspond seulement à la zone DATA ou si on calcule le CRC en incluant STX + LEN + DATA

Si la comparaison échoue, on en déduit que les données reçues ont été altérées, et le message ne sera pas pris en compte.

Pour les messages au contenu binaire, il n'est pas très utile de disposer d'un caractère de fin de message qu'il serait nécessaire de lire après le traitement du CRC.

### 7.14.1. DONNÉES BINAIRES OU ASCII

Pour obtenir un message plus court ou transmettre directement la valeur d'une variable (8, 16 ou 32 bits), il est fréquent de transmettre la valeur binaire. Un des inconvénients est que chaque octet peut prendre n'importe quelle valeur, d'où parfois des risques de confondre un byte de donnée avec le début du message. Ce risque est important si un message a été mal reçu ou fragmenté.

L'autre option est de transmettre les valeurs des variables sous forme de caractères ASCII. Une variable 16 bits dont la valeur hexadécimale est 0x1234 demandera l'envoi de 4 caractères (1, 2, 3 et 4). Par contre ce type de message est facile à observer et évite la confusion entre un caractère de donnée et un caractère spécial comme STX ou ETX.

### 7.14.2. CALCUL D'UN CRC 16 BITS

Le calcul d'un CRC est un calcul successif, chaque caractère du message est utilisé pour cumuler la valeur du CRC. Il est possible d'utiliser une table pour obtenir rapidement la valeur à cumuler.

Avant de passer à un exemple pratique, il est nécessaire de revoir le principe du calcul d'un CRC.

## 7.15. PRINCIPE DU CALCUL D'UN CRC

Le CRC (Cyclic Redundancy Codes) est une évolution du principe d'une simple somme de contrôle (Checksum). Une somme de contrôle consiste à ajouter successivement la valeur des caractères (8 bits) d'un message à une variable 16 ou 32 bits en ne se préoccupant pas des reports si la somme dépasse la taille prévue.

Si on effectue une simple sommation sur 8 bits, pour les 2 messages ci-dessous :

- Pour la chaîne "Bonjour Papa" : La somme est 0x81.
- Pour la chaîne "Banjour Papo" : La somme est 0x81.

Le message n'est pas le même, mais le calcul de la somme donne la même valeur. Dans cet exemple, le deuxième message contient exactement les mêmes caractères, mais dans un ordre différent, ce qui produit la même somme. C'est cette faiblesse qui conduit à augmenter la complexité du calcul.

L'idée fondamentale des algorithmes de CRC est simplement de traiter le message comme une grande valeur numérique, de le diviser par une valeur fixe, de prendre le reste de la division en tant que somme de contrôle.

La valeur numérique du message est bien supérieure à la valeur maximum d'un mot de 32 bits de large. Nous allons donc faire la division Byte par Byte mais cette fois en binaire. Le calcul bit à bit se fait par un "ou exclusif" (encore appelé XOR).

Ce principe peut être simplifié et adapté pour aboutir à l'usage d'une table utilisée pour calculer chaque étape du CRC.

La table doit être construite en fonction du polynôme utilisé pour le calcul du CRC.

Sans entrer dans les détails, il paraît important de mentionner à quoi correspondent les polynômes utilisés pour le calcul des CRC.

### 7.15.1. LES POLYNÔMES UTILISÉS POUR LE CALCUL DU CRC

Les polynômes utilisés pour le calcul du CRC sont des polynômes binaires. En voici quelques-uns couramment utilisés :

- Le polynôme CRC16-CCITT correspond à  $x^{16} + x^{12} + x^5 + x^0$ .  
Si on considère  $x=2$ , on peut calculer la valeur du polynôme soit  $65536 + 4096 + 32 + 1 = 69665$ , soit 0x11021. La valeur utilisée en pratique est 0x1021, donc sans le  $x^{16}$ .
- Le polynôme CRC16 correspond à  $x^{16} + x^{15} + x^2 + x^0$ .  
Si on considère  $x=2$ , on peut calculer la valeur du polynôme soit  $65536 + 32768 + 4 + 1 = 98309$ , soit 0x18005. La valeur utilisée en pratique est 0x8005, donc sans le  $x^{16}$ .
- Le polynôme CRC32 pour les trames Ethernet correspond à :  

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x^1 + x^0$$

### 7.15.2. VARIÉTÉ D'ALGORITHMES

De par ces différents polynômes, il existe donc différents algorithmes de calcul de CRC, qui donneront chacun un CRC différent pour des mêmes données. Mais il peut y avoir encore d'autres différences entre les algorithmes, qui peuvent provenir :

- du polynôme utilisé,
- de la taille du polynôme et donc du CRC résultant (typiquement 8, 16 ou 32 bits),
- de la valeur d'initialisation avant le calcul,
- de la manière d'utiliser le polynôme dans les calculs.

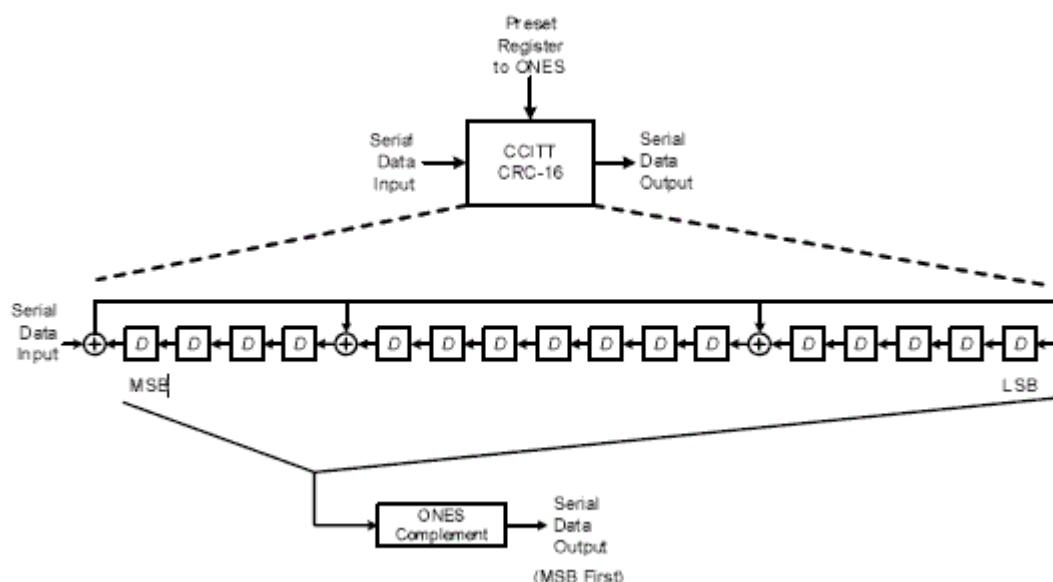
A ce sujet, différentes sources se contredisent, et on rencontre plusieurs versions donnant des résultats différents pour un polynôme et une valeur d'initialisation identiques.

☞ Il est donc important d'utiliser des algorithmes de génération et vérification du CRC qui soient compatibles.

La version de CRC présentée dans ce qui suit est un CRC 16 bits basé sur le polynôme CRC16-CCITT, avec valeur d'initialisation 0xFFFF.

### 7.15.3. EXEMPLE DE SCHÉMA LOGIQUE

Voici un schéma d'exemple hardware servant à calculer un CRC, basé sur le polynôme CRC16-CCITT :



Source : UWB PLCP header generator, Keysight,  
[http://literature.cdn.keysight.com/litweb/pdf/ads2008/uwb/ads2008/UWB\\_PLCP\\_Header.html](http://literature.cdn.keysight.com/litweb/pdf/ads2008/uwb/ads2008/UWB_PLCP_Header.html)

On remarque la simplicité des composants utilisés : des bascules D ainsi que des portes XOR. A l'initialisation, on doit présenter les bascules à 1, puis on fait circuler le message bit à bit. A la fin, le CRC est contenu dans les bascules.

On notera la correspondance entre les connexions des bascules et le polynôme CRC16-CCITT (0x1021).

## 7.15.4. EXEMPLE D'ALGORITHME

### 7.15.4.1. CALCUL DU CRC BIT À BIT SUR 1 BYTE

Voici la fonction qui met à jour le CRC en fonction d'un byte :

```
// Fonction pour calcul du CRC byte à byte avec
// algorithme bit à bit
unsigned short updateCRC16(unsigned short crc,
                           unsigned char data)
{
    unsigned short i, xor_flag, ch;
#define polyDirect 0x1021

    ch = data << 8;
    for(i = 0; i < 8; i++)
    {
        if ((crc ^ ch) & 0x8000)      // get MSBit
        {
            crc <= 1;
            crc ^= polyDirect;
        }
        else
        {
            crc <= 1;
        }
        ch <= 1;
    }
    return crc;
}
```

On peut voir dans cet algorithme le traitement successif de chaque bit du nouvel octet reçu avec mise à jour du CRC en fonction.

### 7.15.4.2. CALCUL DU CRC D'UN MESSAGE

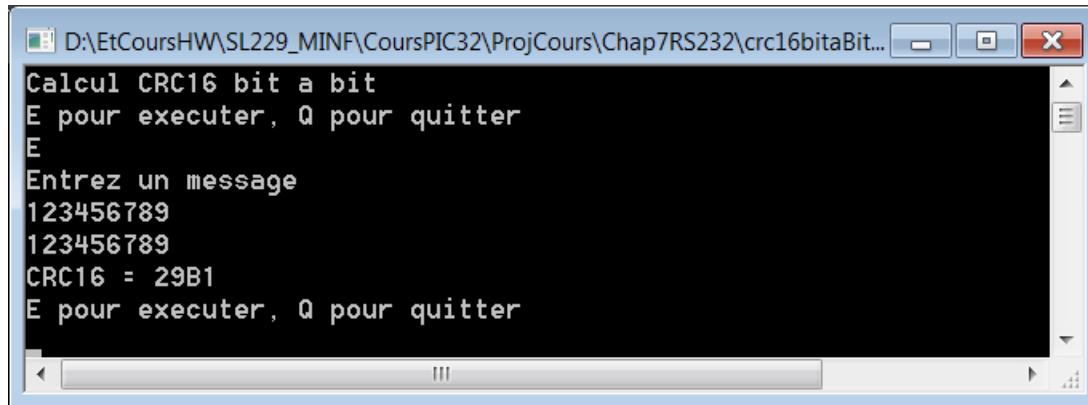
Pour calculer le CRC16 d'un message, il suffit d'initialiser la valeur du CRC à 0xFFFF, puis d'appeler la fonction updateCrc16 pour chaque byte du message.

```
unsigned short crc16

// initialisation selon CCITT
crc16 = 0xFFFF;
//--- Boucle de lecture de la chaîne
for (i = 0; i < strlen(Mess); i++) {
    crc16 = updateCRC16(crc16, Mess[i]);
}
printf("CRC16 = %04X \n", crc16);
```

#### 7.15.4.3. EXEMPLE DE RÉSULTAT

Voici les résultats obtenus avec le message "123456789"



```
D:\EtCoursHW\SL229_MINF\CoursPIC32\ProjCours\Chap7RS232\crc16bitaBit...
Calcul CRC16 bit a bit
E pour executer, Q pour quitter
E
Entrez un message
123456789
123456789
CRC16 = 29B1
E pour executer, Q pour quitter
```

On obtient 0x29B1, ce qui correspond à la valeur d'un CRC16 CCITT :

<b>"123456789"</b>	
1 byte checksum	<b>221</b>
CRC-16	<b>0xBB3D</b>
CRC-16 (Modbus)	<b>0x4B37</b>
CRC-16 (Sick)	<b>0x56A6</b>
CRC-CCITT (XModem)	<b>0x31C3</b>
CRC-CCITT (0xFFFF)	<b>0x29B1</b>
CRC-CCITT (0x1D0F)	<b>0xE5CC</b>
CRC-CCITT (Kermit)	<b>0x8921</b>
CRC-DNP	<b>0x82EA</b>
CRC-32	<b>0xCB43926</b>

Source : <https://www.lammertbies.nl/comm/info/crc-calculation.html>

Relevons par ailleurs la multitude de versions de CRC proposée ici.

### 7.15.5. CALCUL DU CRC AU MOYEN D'UNE TABLE

Pour un traitement plus rapide, la succession de décalages pour un byte peut être remplacée par un calcul utilisant une valeur pré-calculée dans une table et une combinaison de OU-exclusif, comme ci-dessous.

Le choix est alors laissé au programmeur entre consommation mémoire (pour la table de 256 valeurs), ou le temps d'exécution (succession d'opérations-décalages).

#### 7.15.5.1. FORMULE

```
crc = CRC16_table[((crc >> 8) & 0xFF) ^ data] ^ (crc << 8);
```

Afin de pouvoir vérifier le fonctionnement du calcul du CRC avec une table, voici l'adaptation en projet Visual C++, de l'exemple fourni par Michael Neumann.

#### 7.15.5.2. TABLE CALCUL DU CRC

```
// Table calcul CRC16 (Polynome 0x1021)
const unsigned short CRC16_table[256] = {
    0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50a5, 0x60c6, 0x70e7,
    0x8108, 0x9129, 0xa14a, 0xb16b, 0xc18c, 0xd1ad, 0xe1ce, 0xf1ef,
    0x1231, 0x0210, 0x3273, 0x2252, 0x52b5, 0x4294, 0x72f7, 0x62d6,
    0x9339, 0x8318, 0xb37b, 0xa35a, 0xd3bd, 0xc39c, 0xf3ff, 0xe3de,
    0x2462, 0x3443, 0x0420, 0x1401, 0x64e6, 0x74c7, 0x44a4, 0x5485,
    0xa56a, 0xb54b, 0x8528, 0x9509, 0xe5ee, 0xf5cf, 0xc5ac, 0xd58d,
    0x3653, 0x2672, 0x1611, 0x0630, 0x76d7, 0x66f6, 0x5695, 0x46b4,
    0xb75b, 0xa77a, 0x9719, 0x8738, 0xf7df, 0xe7fe, 0xd79d, 0xc7bc,
    0x48c4, 0x58e5, 0x6886, 0x78a7, 0x0840, 0x1861, 0x2802, 0x3823,
    0xc9cc, 0xd9ed, 0xe98e, 0xf9af, 0x8948, 0x9969, 0xa90a, 0xb92b,
    0x5af5, 0x4ad4, 0x7ab7, 0x6a96, 0x1a71, 0x0a50, 0x3a33, 0x2a12,
    0xdbfd, 0xcbdc, 0xfbff, 0xeb9e, 0x9b79, 0x8b58, 0xbb3b, 0xab1a,
    0x6ca6, 0x7c87, 0x4ce4, 0x5cc5, 0x2c22, 0x3c03, 0x0c60, 0x1c41,
    0xedae, 0xfd8f, 0xcdec, 0xddcd, 0xad2a, 0xbd0b, 0x8d68, 0x9d49,
    0x7e97, 0x6eb6, 0x5ed5, 0x4ef4, 0x3e13, 0x2e32, 0x1e51, 0x0e70,
    0xff9f, 0xefbe, 0xdfdd, 0cfffc, 0xbf1b, 0xaf3a, 0x9f59, 0x8f78,
    0x9188, 0x81a9, 0xb1ca, 0xa1eb, 0xd10c, 0xc12d, 0xf14e, 0xe16f,
    0x1080, 0x00a1, 0x30c2, 0x20e3, 0x5004, 0x4025, 0x7046, 0x6067,
    0x83b9, 0x9398, 0xa3fb, 0xb3da, 0xc33d, 0xd31c, 0xe37f, 0xf35e,
    0x02b1, 0x1290, 0x22f3, 0x32d2, 0x4235, 0x5214, 0x6277, 0x7256,
    0xb5ea, 0xa5cb, 0x95a8, 0x8589, 0xf56e, 0xe54f, 0xd52c, 0xc50d,
    0x34e2, 0x24c3, 0x14a0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405,
    0xa7db, 0xb7fa, 0x8799, 0x97b8, 0xe75f, 0xf77e, 0xc71d, 0xd73c,
    0x26d3, 0x36f2, 0x0691, 0x16b0, 0x6657, 0x7676, 0x4615, 0x5634,
    0xd94c, 0xc96d, 0xf90e, 0xe92f, 0x99c8, 0x89e9, 0xb98a, 0xa9ab,
    0x5844, 0x4865, 0x7806, 0x6827, 0x18c0, 0x08e1, 0x3882, 0x28a3,
    0xcb7d, 0xdb5c, 0xeb3f, 0xfb1e, 0x8bf9, 0x9bd8, 0xabbb, 0xbb9a,
    0x4a75, 0x5a54, 0x6a37, 0x7a16, 0xaaf1, 0x1ad0, 0x2ab3, 0x3a92,
    0xfd2e, 0xed0f, 0xdd6c, 0xcd4d, 0xbdaa, 0xad8b, 0x9de8, 0x8dc9,
    0x7c26, 0x6c07, 0x5c64, 0x4c45, 0x3ca2, 0x2c83, 0x1ce0, 0x0cc1,
    0xef1f, 0xff3e, 0xcf5d, 0xdf7c, 0xaf9b, 0xbfba, 0x8fd9, 0x9ff8,
    0xe17, 0x7e36, 0x4e55, 0x5e74, 0x2e93, 0x3eb2, 0x0ed1, 0x1ef0
};
```

La fonction de calcul du CRC pour 1 byte devient très simple.

**7.15.5.3. FONCTION POUR CALCUL DU CRC AVEC LA TABLE**

```
unsigned short updateCRC16(unsigned short crc,  
                           unsigned char data)  
{  
    // retourne la nouvelle valeur du crc  
    return (CRC16_table[((crc >> 8) & 0xFF) ^ data] ^ (crc << 8));  
}
```

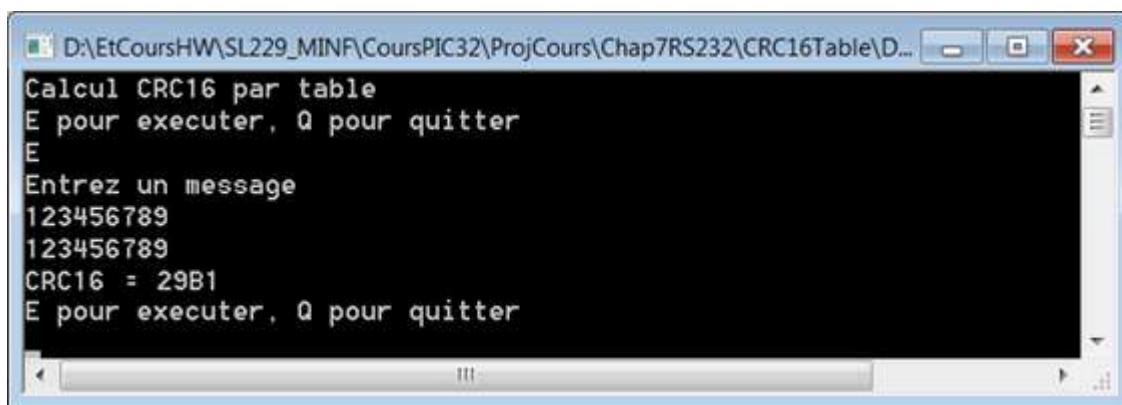
### 7.15.5.1. CALCUL DU CRC D'UN MESSAGE

Le principe du calcul du CRC16 d'un message reste le même, il suffit d'appeler la fonction updateCRC16 pour chaque byte du message.

```
unsigned short crc16  
// initialisation selon CCITT  
crc16 = 0xFFFF;  
  
//--- Boucle de lecture de la chaine  
for (i = 0; i < strlen(Mess); i++) {  
    crc16 = updateCRC16(crc16, Mess[i]);  
}  
printf("CRC16 = %04X \n", crc16);
```

### 7.15.5.2. EXEMPLE DE RÉSULTAT

Voici les résultats obtenus avec le message "123456789"



On obtient bien 0x29B1.

### 7.15.6. CALCUL DU CRC AU MOYEN DE DEUX TABLES

Un compromis pour réduire l'encombrement de la table de 256 valeurs, est possible au moyen de deux tables plus petites.

Cet algorithme provient de Ashley Roll, Digital Nemesis Pty Ltd, [www.digitalnemesis.com](http://www.digitalnemesis.com).

### 7.15.6.1. LES 2 TABLES DE CALCUL DU CRC

```
// CRC16 Lookup tables (High and Low Byte) for 4 bits per iteration.
// Polynome CCITT 0x1021
unsigned char CRC16_LookupHigh[16] = {
    0x00, 0x10, 0x20, 0x30, 0x40, 0x50, 0x60, 0x70,
    0x81, 0x91, 0xA1, 0xB1, 0xC1, 0xD1, 0xE1, 0xF1
};

unsigned char CRC16_LookupLow[16] = {
    0x00, 0x21, 0x42, 0x63, 0x84, 0xA5, 0xC6, 0xE7,
    0x08, 0x29, 0x4A, 0x6B, 0x8C, 0xAD, 0xCE, 0xEF
};
```

### 7.15.6.2. SOUS-FONCTIONS ET FONCTIONS

```
// Process 4 bits of the message to update the CRC Value.
// Note that the data must be in the low nibble of val.
void CRC16_Update4Bits( unsigned char val )
{
    unsigned char      t;

    // Extract the Most significant 4 bits of the CRC register
    t = (CRC16_High >> 4) & 0x0F;

    // XOR in the Message Data into the extracted bits
    t = t ^ val;

    // Shift the CRC Register left 4 bits
    CRC16_High = ((CRC16_High << 4) & 0xF0) | ((CRC16_Low >> 4)
                                                & 0x0F);
    CRC16_Low = CRC16_Low << 4;

    // Do the table lookups and XOR the result into the CRC Tables
    CRC16_High = CRC16_High ^ CRC16_LookupHigh[t];
    CRC16_Low  = CRC16_Low  ^ CRC16_LookupLow[t];
}

// Process one Message Byte to update the current CRC Value
void CRC16_Update( unsigned char val )
{
    CRC16_Update4Bits( (val >> 4) & 0x0F );      // High nibble first
    CRC16_Update4Bits( val & 0x0F );                // Low nibble
}
```

### 7.15.6.3. CALCUL DU CRC D'UN MESSAGE

Le principe du calcul du CRC16 d'un message reste le même, il suffit d'appeler la fonction updateCRC16 pour chaque byte du message.

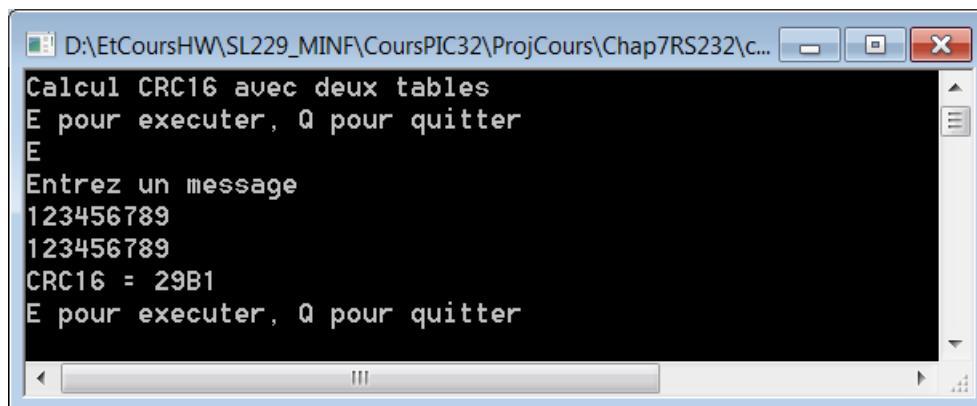
```
unsigned char CRC16_High, CRC16_Low;

// Initialise the CRC to 0xFFFF for the CCITT specification
CRC16_High = 0xFF;
CRC16_Low = 0xFF;

//--- Boucle de lecture de la chaine
for (i = 0; i < strlen(Mess); i++) {
    CRC16_Update(Mess[i]);
    Res1 = (CRC16_High << 8) | CRC16_Low;
}
Res1 = (CRC16_High << 8) | CRC16_Low;
printf("CRC16 = %04X \n", Res1);
```

### 7.15.6.4. EXEMPLE DE RÉSULTAT

Voici les résultats obtenus avec le message "123456789" :



On obtient le même résultat que précédemment et la valeur 0x29B1 correspond à la valeur indiquée par l'auteur pour le message "123456789".

### 7.15.7. EXEMPLE D'ÉVOLUTION DU CRC SUR 2 TRAMES

Voici la comparaison des valeurs de CRC obtenues sur 2 messages de 3 octets qui diffèrent peu :

The screenshot shows a terminal window with the following text output:

```
D:\EtCoursHW\SL229_MINF\CoursPIC32\ProjCours\Chap7RS232\Crc16Comp...\ Test Calcul CRC16 sur message TP3 2014-2015 C. Huber  
E pour executer, Q pour quitter  
E  
Entrez Vitesse -99 to +99  
25  
Entrez Angle -90 to +90  
45  
Contenu du message : AA 19 2D  
ValCrc FAE5  
E pour executer, Q pour quitter  
E  
Entrez Vitesse -99 to +99  
25  
Entrez Angle -90 to +90  
46  
Contenu du message : AA 19 2E  
ValCrc CA86  
E pour executer, Q pour quitter
```

Evolution de la valeur de CRC pour une modification de la valeur du 3<sup>ème</sup> octet du message.

- Message 1 : AA 19 2D, CRC = FAE5
- Message 2 : AA 19 2E, CRC = CA86

On constate que la valeur du CRC change complètement, ce qui est une force de d'un algorithme de type CRC par rapport aux autres types de vérification d'erreur (par exemple parité ou checksum).

### 7.15.8. VÉRIFICATION DU CRC À LA RÉCEPTION

En organisant un message binaire comme ci-dessous, lors du calcul du CRC à la réception sur l'entier du message nous devons obtenir 0, ce qui facilite le test.

Start [0]	Vitesse [1]	Angle [2]	[3]	CRC16	[4]
0xAA	1 byte	1 byte		MSB CRC	LSB CRC

⚠️ Attention : ceci est uniquement valable si on place le CRC dans le message avec MSB puis LSB. Comme précédemment, la méthode de calcul du CRC à l'émission dépend de l'algorithme utilisé.

#### 7.15.8.1. ALGORITHME DE CALCUL DU CRC LORS DE L'ÉMISSION

Voici la préparation du message, le calcul du CRC et sa mise en place dans le message.

```
// Préparation du message (start + payload)
TxBuffer2[0] = 0xAA;
TxBuffer2[1] = Vitesse;
TxBuffer2[2] = Angle;
crc16 = 0xFFFF;
// Calcul du CRC sur les 3 1er octets
for (i=0 ; i < 3 ; i++) {
    crc16 = updateCRC16_new(crc16, TxBuffer2[i]);
}
// mise en place du CRC dans le message
uTemp.val = crc16;
TxBuffer2[3] = uTemp.shl.msb;
TxBuffer2[4] = uTemp.shl.lsb;
```

#### 7.15.8.2. CONTRÔLE DU CRC À LA RÉCEPTION

Calcul du CRC sur l'entier du message, y compris les 2 octets du CRC.

```
crc16 = 0xFFFF;
printf("CRC calcule sur 5 octets \n");
printf("Contenu message : \n");
for (i = 0; i < MESS_SIZE; i++) {
    printf ("%02X ", TxBuffer2[i]);
    crc16 = updateCRC16_new(crc16, TxBuffer2[i]);
}
printf(" CRC obtenu en reception = %04X \n", crc16);
if (crc16 == 0) {
    printf("CRC OK \n");
} else {
    printf("CRC pas OK : val = %04X \n", crc16);
}
```

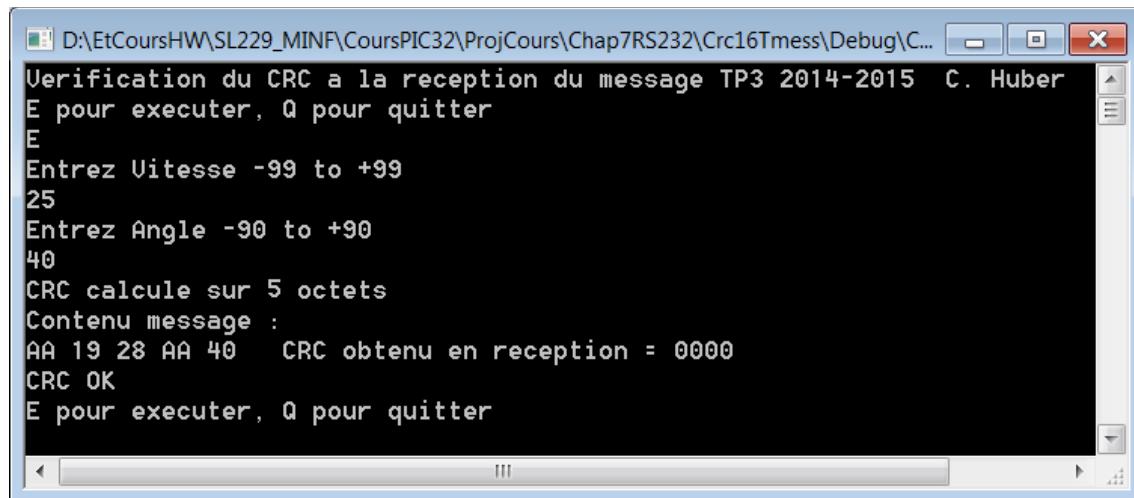
### 7.15.8.2.1. Définition de l'union

Voici la définition de l'union utilisée pour manipuler une valeur 16 bits.

```
typedef union {
    unsigned short val;
    struct {unsigned char lsb;
            unsigned char msb;} shl;
} U_manip16;
```

### 7.15.8.3. VÉRIFICATION DU RÉSULTAT

Comme le montre la copie d'écran ci-dessous, on obtient bien 0 comme résultat du CRC calculé sur l'entier du message :



The screenshot shows a terminal window with the following text output:

```
D:\EtCoursHW\SL229_MINF\CoursPIC32\ProjCours\Chap7RS232\Crc16Tmess\Debug\...  
Verification du CRC a la reception du message TP3 2014-2015 C. Huber  
E pour executer, Q pour quitter  
E  
Entrez Vitesse -99 to +99  
25  
Entrez Angle -90 to +90  
40  
CRC calcule sur 5 octets  
Contenu message :  
AA 19 28 AA 40    CRC obtenu en reception = 0000  
CRC OK  
E pour executer, Q pour quitter
```

### 7.15.9. CALCUL CRC16, LIBRAIRIE À DISPOSITION

La fonction de calcul du CRC16-CCITT et la table des 256 valeurs sont fournies sur le réseau dans les fichiers **Mc32CalCrc16.h** et **Mc32CalCrc16.c** sous:

...\\Maitres-Eleves\\SLO\\Modules\\SL229\_MINF\\PIC32MX\_Utilitaires(PlibHarmony)\\  
Fifo&Antirebond.

## 7.16. EXEMPLE COMPLET AVEC ÉMISSION ET RÉCEPTION

Dans cet exemple, nous allons montrer l'émission d'un message sécurisé par CRC ainsi que le traitement de la réception, avec la reconstitution du message qui est la partie la plus délicate.

Nous allons prendre le cas d'un message binaire. Dans cet exemple nous nous focalisons sur les traitements hors interruption, c'est-à-dire les fonctions SendMessage et GetMessage.

### 7.16.1. STRUCTURE DU MESSAGE DE L'EXEMPLE

	No	DATA		CRC16	
0xAA	1 ou 2	Lsb	Msb	Msb	Lsb

En émission, le CRC16 est calculé sur 0xAA + No Canal + DATA.

Pour le CRC16 on utilisera le CRC16-CCITT. Calcul par table de 256 valeurs.

En réception, le calcul du CRC est réalisé sur l'entier du message et s'il est valable la valeur obtenue doit être de zéro.

Le data est une valeur 16 bits qui permet de transporter la valeur lue correspondant au convertisseur A/D des PIC32MX. Le No indique le numéro du canal de l'A/D 1 ou 2.

☞ Le data est transmis avec le Lsb en premier ce qui permet la manipulation aisée par une union. Comme nous l'avons vu précédemment le CRC doit être transmis avec le Msb en premier pour permettre le contrôle du CRC sur l'entier du message.

### 7.16.2. CYCLES DE TRAITEMENTS

L'application est activée toutes les 10 ms. La fonction GetMessage est appelée à chaque activation de l'application. La fonction SendMessage est appelée au même rythme de manière à envoyer un seul message en alternant le numéro du canal.

Le débit maximum de la transmission avec une taille de message de 6 octets et un baudrate de 57'600 sans parité et avec 1 seul bit de stop, ce qui fait 10 bits pour un octet donc 60 bits par message. D'où  $57'600 / 60 = 960$  messages par seconde.

Le système mis en place est capable de recevoir 100 messages par seconde et il en émet 100 par seconde.

### 7.16.3. DÉFINITIONS ET ÉLÉMENTS GLOBAUX

Voici les différents éléments permettant la gestion des messages. Le choix s'est porté sur une structure pour décrire le message, ce qui donne une meilleure description mais ne permet pas de traitement en boucle.

On trouve aussi la déclaration des FIFO avec le choix de leur taille pour accueillir 4 messages.

Voici encore une union pour la manipulation des éléments 16 bits :

```
typedef union {
    uint16_t val;
    struct {uint8_t lsb;
            uint8_t msb;} shl;
} U_manip16;
// Définitions pour les messages
#define MESS_SIZE 6
#define STX_code (0xAA)

// Structure décrivant le message
typedef struct {
    uint8_t Start;
    uint8_t NoCh;
    uint16_t Data;
    uint8_t MsbCrc;
    uint8_t LsbCrc;
} StruMess;

// Type énuméré pour les étapes de traitement en réception
typedef enum { WaitSTX = 1, WaitNoCh, WaitLsbData, WaitMsbData ,
               WaitMsbCrc,   WaitLsbCrc
} E_MessSituation;

// Structure pour émission des messages
StruMess TxMess;
// Structure pour réception des messages
StruMess RxMess;

// Déclaration des FIFO pour réception et émission
#define FIFO_RX_SIZE ((4*MESS_SIZE) + 1) // 4 messages
#define FIFO_TX_SIZE ((4*MESS_SIZE) + 1) // 4 messages

int8_t fifoRX[FIFO_RX_SIZE];
// Déclaration du(descripteur du FIFO de réception
S_fifo descrFifoRX;

int8_t fifoTX[FIFO_TX_SIZE];
// Déclaration du(descripteur du FIFO d'émission
S_fifo descrFifoTX;
```

#### 7.16.4. LA FONCTION SENDMESSAGE

La fonction **SendMessage** détermine s'il y a la place pour déposer un message complet. Si c'est le cas, le message est composé avec le calcul du CRC16, puis le message est placé dans le FifoTX. A la fin de la fonction, on trouve la gestion du contrôle de flux.

```

void SendMessage(int8_t NoCh, int16_t Data)
{
    uint8_t FreeSize;
    uint16_t ValCrc16 = 0xFFFF;
    U_manip16 tmpCrc, TmpData;
    // Test si place Pour écrire 1 message
    FreeSize = GetWriteSpace ( &descrFifoTX);
    if (FreeSize >= (MESS_SIZE) ) {

        // Compose le message
        TxMess.Start = 0xAA; // start
        ValCrc16 = updateCRC16(ValCrc16, TxMess.Start);

        TxMess.NoCh = NoCh;
        ValCrc16 = updateCRC16(ValCrc16, TxMess.NoCh);

        // Traitement data
        TxMess.Data = Data;
        TmpData.val = Data;
        ValCrc16 = updateCRC16(ValCrc16, TmpData.sh1.lsb);
        ValCrc16 = updateCRC16(ValCrc16, TmpData.sh1.msb);

        // Traitement CRC
        tmpCrc.val = ValCrc16;
        TxMess.MsbCrc = tmpCrc.sh1.msb;
        TxMess.LsbCrc = tmpCrc.sh1.lsb;

        // Dépose le message dans le fifo
        PutCharInFifo ( &descrFifoTX, TxMess.Start);
        PutCharInFifo ( &descrFifoTX, TxMess.NoCh);
        PutCharInFifo ( &descrFifoTX, TmpData.sh1.lsb);
        PutCharInFifo ( &descrFifoTX, TmpData.sh1.msb);
        PutCharInFifo ( &descrFifoTX, TxMess.MsbCrc);
        PutCharInFifo ( &descrFifoTX, TxMess.LsbCrc);
    }

    // Gestion du control de flux
    // Si on a un caractère à envoyer et que CTS = 0
    FreeSize = GetReadSize(&descrFifoTX);
    if ((RS232_CTS == 0) && (FreeSize > 0))
    {
        // Autorise l'interruption d'émission
        PLIB_INT_SourceEnable(INT_ID_0,
                              INT_SOURCE_USART_1_TRANSMIT);
    }
} // End SendMessage

```

### 7.16.5. LA FONCTION GETMESSAGE

A chaque appel de la fonction GetMessage, on détermine si on peut traiter un message complet. Le traitement est réalisé par une boucle qui décompte le nombre de caractères obtenus du FIFO tout en surveillant la fin du message. Le mécanisme doit supporter un abandon de la boucle sans obtenir un message complet, ceci dans une situation de message incomplet. Pour savoir quel caractère est traité, on utilise une variable d'état dont la valeur définie par un type énuméré permet de progresser dans un Switch.

```

bool GetMessage(uint16_t *Ch1, uint16_t *Ch2)
{
    static E_MessSituation GetSituation = WaitSTX ;
    static uint16_t ValCrc;           // en cas d'abandon en cours
    static U_manip16 TmpData;        // en cas d'abandon en cours

    uint8_t EndMess;
    uint8_t RxC;
    uint8_t NbCharToRead = 0;
    bool MessReady = false;          // indique validité message reçu

    // Détermine le nombre de caractères à lire
    NbCharToRead = GetReadSize ( &descrFifoRX);

    // Si >= taille message alors traite
    if (NbCharToRead >= MESS_SIZE)  {

        EndMess = 0;
        while ( (NbCharToRead >= 1) && ( EndMess == 0) ) {

            // Lis un caractère sans gestion du status
            GetCharFromFifo ( &descrFifoRX, &RxC );
            NbCharToRead--;

            // Traitement selon état du message
            switch (GetSituation)  {

                case WaitSTX :
                    if ( RxC == STX_code) {
                        RxMess.Start = RxC;
                        ValCrc = 0xFFFF;
                        ValCrc = updateCRC16(ValCrc, RxC);
                        GetSituation = WaitNoCh;
                    } else {
                        BSP_LEDToggle(BSP_LED_1); // pour test
                    }
                    break;

                case WaitNoCh :
                    RxMess.NoCh = RxC;
                    ValCrc = updateCRC16(ValCrc, RxC);
                    GetSituation = WaitLsbData;
                    break;

                case WaitLsbData :
                    ValCrc = updateCRC16(ValCrc, RxC);
                    TmpData.sh1.lsb = RxC;
                    break;
            }
        }
    }
}

```

```

        GetSituation = WaitMsbData;
        break;

    case WaitMsbData :
        ValCrc = updateCRC16(ValCrc, RxC);
        TmpData.shl.msb = RxC;
        RxMess.Data = TmpData.val;
        GetSituation = WaitMsbCrc;
        break;

    case WaitMsbCrc :
        RxMess.MsbCrc = RxC;
        ValCrc = updateCRC16(ValCrc, RxC);
        GetSituation = WaitLsbCrc;
        break;

    case WaitLsbCrc :
        RxMess.LsbCrc = RxC;
        ValCrc = updateCRC16(ValCrc, RxC);
        if (ValCrc == 0 ) {
            // Contenu OK
            if (RxMess.NoCh == 1) {
                *Ch1 = RxMess.Data;
            }
            if (RxMess.NoCh == 2) {
                *Ch2 = RxMess.Data;
            }
            MessReady = true;
        } else {
            BSP_LEDToggle(BSP_LED_7); // pour test
        }
        GetSituation = WaitSTX;
        EndMess = 1; // pour ne traiter qu'un seul
                      // message à la fois
        break;
    } // end switch
} // end while
}

// Gestion control de flux de la réception
// 12 correspond à 2 paquets de 6 (3/4 RxBuffer de 8)
if(GetWriteSpace ( &descrFifoRX) >= 12) {
    // Autorise émission par l'autre
    RS232_RTS = 0;
}
return MessReady;
} // End GetMessage

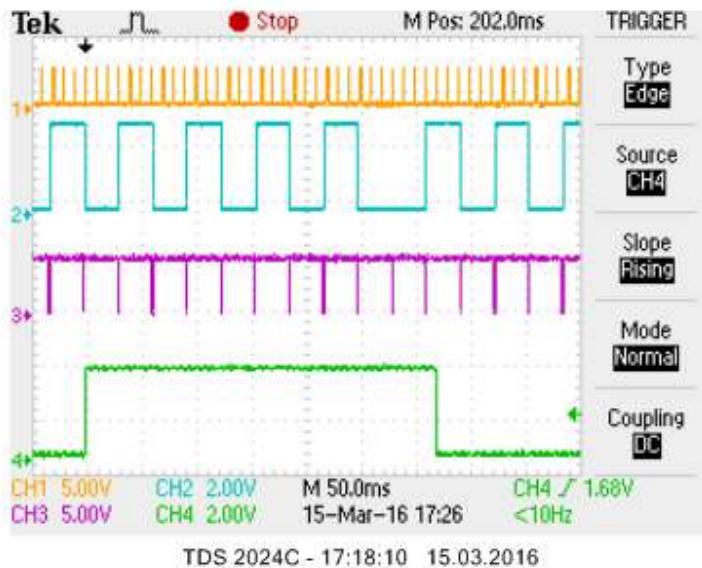
```

### 7.16.6. RÉACTION DU SYSTÈME À DES MESSAGES CORROMPUS

Pour vérifier si le système de réception supporte des messages corrompus, il est nécessaire d'introduire depuis l'application la possibilité d'injecter, par exemple une fois sur 10, un message incomplet. Nous choisissons un message pour lequel les 2 octets de CRC manquent. Sa taille sera donc de 4 octets au lieu de 6.

#### 7.16.6.1. VUE GÉNÉRALE RÉACTION À MAUVAIS MESSAGE

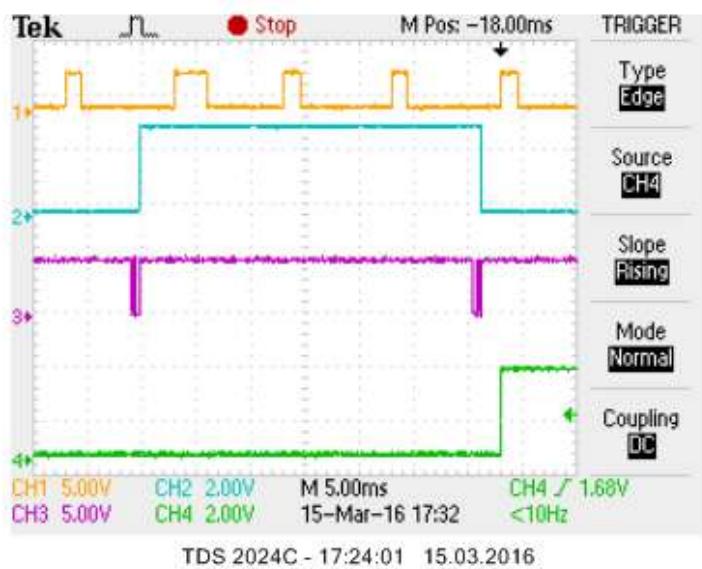
- canal 1: LED\_0  
(marque traitement dans l'application)
- canal 2: LED\_3  
(inversion à chaque int RX)
- canal 3 : broche 52 U1RX
- canal 4: LED\_7  
(inversion si erreur CRC)



On remarque l'erreur CRC tous les 10 messages.

#### 7.16.6.2. VUE RÉACTION À MAUVAIS MESSAGE

- canal 1: LED\_0  
(marque traitement dans l'application)
- canal 2: LED\_3  
(inversion à chaque int RX)
- canal 3 : broche 52 U1RX
- canal 4: LED\_7  
(inversion si erreur CRC)



Il est difficile de repérer le message plus court.

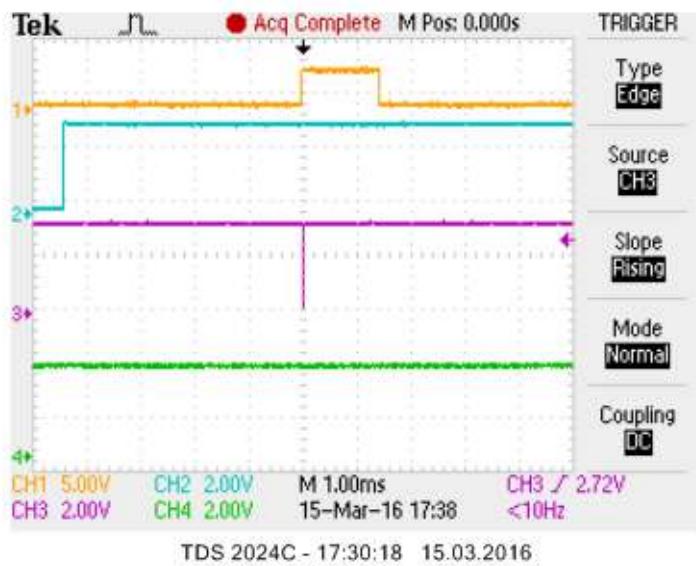
### 7.16.6.3. OBSERVATION ATTENTE STX

canal 1: LED\_0  
(marque traitement dans l'application)

canal 2: LED\_3  
(inversion à chaque int RX)

canal 3 : LED\_1  
inversion dans attente STX

canal 4: LED\_7  
(inversion si erreur CRC)



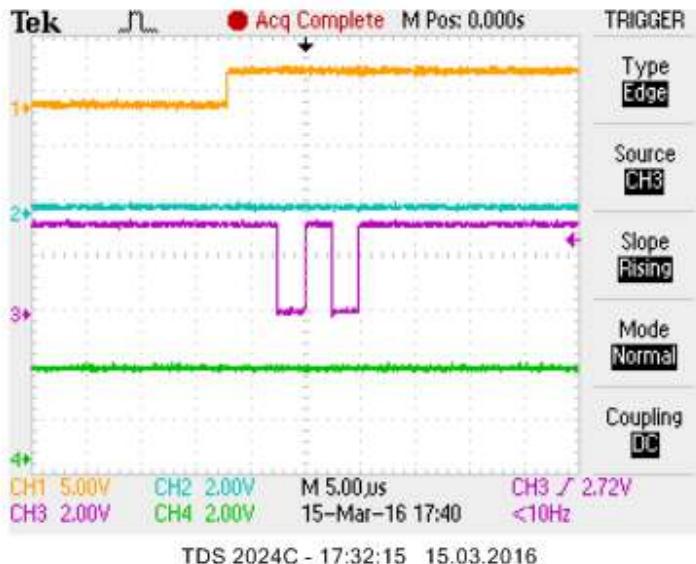
### 7.16.6.1. OBSERVATION ATTENTE STX DÉTAIL

canal 1: LED\_0  
(marque traitement dans l'application)

canal 2: LED\_3  
(inversion à chaque int RX)

canal 3 : LED\_1  
inversion dans attente STX

canal 4: LED\_7  
(inversion si erreur CRC)



On peut observer les 4 flancs qui nous indiquent la consommation de 4 caractères n'étant pas le STX.

### 7.16.6.2. CONCLUSION SUR LE TRAITEMENT DE MAUVAIS MESSAGE

Le système se comporte correctement grâce à l'attente du STX, ce qui a pour effet de consommer une partie de message pour se recaler sur un bon message.

### 7.16.7. CONCLUSION SUR L'EXEMPLE

Cet exemple a montré comment composer un message avec le calcul et la mise en place d'un CRC. Le principe est facilement adaptable à d'autres structures, il est aussi possible de remplacer la structure décrivant le message par un tableau.

En ce qui concerne la fonction GetMessage, la solution proposée combine une structure avec une avance étape par étape dans le switch. Cette solution un peu lourde présente l'avantage de pouvoir être facilement adaptée à d'autres organisations du message. Il est aussi possible d'utiliser un tableau à la place de la structure, ce qui permettrait de réduire le switch, voire de le supprimer puisqu'il suffit d'attendre le début du message, puis de compter les caractères traités, le critère de fin étant la bonne taille.

## 7.17. CONCLUSION GÉNÉRALE

Ce chapitre a présenté différents aspects de la communication sérielle asynchrone, dans le but de permettre à l'étudiant de disposer d'une méthode pour réaliser une telle communication adaptée aux contraintes d'un projet spécifique.

Il est important d'introduire des moyens pour vérifier le bon comportement de la communication au niveau des interruptions et du contrôle de flux, car des changements de taille de message, de FIFO ou autre peuvent modifier le comportement.

La combinaison de FIFO software avec les interruptions de réception et d'émission est le meilleur moyen d'obtenir des communications sérieelles efficaces en découplant les traitements. Les queues hardware de l'USART, bien utilisées, permettent de réduire la fréquence des interruptions, ce qui réduit la charge du processeur.

Suivant le type de transmission, le contrôle de l'intégrité des données transmises peut s'avérer nécessaire.

Il faut garder à l'esprit que, sans être complexe, une communication sérielle demande un certain soin dans sa réalisation et sa mise au point.

## 7.18. HISTORIQUE DES VERSIONS

### 7.18.1. VERSION 1.5 JANVIER 2015

Création de ce document par transformation du chapitre 12 du PIC18F. Version 1.5 pour indiquer l'utilisation des PLIB de Harmony. Mise à jour du traitement par CRC. Cette version n'est pas entièrement adaptée au PIC32MX.

### 7.18.2. VERSION 1.6 MARS 2015

Introduction émission et réception sans interruptions. Adaptation des mécanismes d'émission et réception avec interruptions et FIFO au PIC32MX. Remplacement de l'exemple de réception par un exemple complet avec les fonctions SendMessage et GetMessage.

### 7.18.3. VERSION 1.7 MARS 2016

Reprise des exemples et complément pour les modes du FIFO de réception. Ajout observation situation erreur CRC.

### 7.18.4. VERSION 1.7\_1 MARS 2016

Retouche 7.15.3.3 page 75 et retouche exemple et résultat calcul CRC par table.

### 7.18.5. VERSION 1.8 FÉVRIER 2017

Reprise par SCA. Relecture générale. Elucidation et nettoyage câbles (link et null-modem) p. 7 et 12. Enlevé version de code d'émission non amélioré (avec interruption parasite après une transmission). Simplification partie CRC (gardé que le nouvel algorithme, qui semble être le bon).

### 7.18.1. VERSION 1.9 JANVIER 2018

Ajouts documents de référence. Corrections mineures.

### 7.18.1. VERSION 1.91 JANVIER 2019

Adaptation exemple §7.13 "Exemple utilisation des FIFOs et des interruptions" à Harmony 2.05 (configurateur graphique et code init. généré ont changé).

Ajout §7.15.3 exemple de schéma logique pour génération hardware d'un CRC.

### 7.18.2. VERSION 1.92 SEPTEMBRE 2022

Mise à jour du code de la librairie GesFifoTh32.

**MINF**  
**Programmation des PIC32MX**

# **Chapitre 8**

## **Gestion du bus SPI**



### **Théorie PIC32MX**

**Christian HUBER (CHR)**  
**Serge CASTOLDI (SCA)**  
**Version 1.91 mars 2019**



## CONTENU DU CHAPITRE 8

<b>8. Gestion du bus SPI avec le PIC32MX</b>	<b>8-1</b>
<b>8.1. Bus SPI</b>	<b>8-2</b>
8.1.1. Les lignes du bus SPI	8-2
8.1.2. Connexions entre master et slaves	8-2
8.1.3. Connexions multi-master	8-3
8.1.4. Les modes master et slave	8-3
<b>8.2. Réalisation du bus SPI avec le PIC32MX</b>	<b>8-4</b>
8.2.1. Câblage du SPI sur le KITPIC32MX	8-4
8.2.2. Schéma-bloc du module SPI	8-5
8.2.3. Particularité du SPIxSR	8-5
8.2.4. Les possibilités du module SPI	8-6
8.2.5. Timing en mode master (8 bits)	8-7
8.2.6. Timing en mode Slave	8-8
8.2.6.1. Slave Select pin disabled	8-8
8.2.6.2. Slave Select pin enabled	8-9
<b>8.3. Les fonctions SPI de la PLIB_SPI</b>	<b>8-10</b>
8.3.1. Les fonctions de configuration	8-10
8.3.1.1. La fonction PLIB_SPI_BaudRateClockSelect	8-10
8.3.1.2. La fonction PLIB_SPI_BaudRateSet	8-10
8.3.1.3. La fonction PLIB_SPI_CommunicationWidthSelect	8-11
8.3.1.4. La fonction PLIB_SPI_ClockPolaritySelect	8-11
8.3.1.5. La fonction PLIB_SPI_InputSamplePhaseSelect	8-11
8.3.1.6. La fonction PLIB_SPI_OutputDataPhaseSelect	8-12
8.3.1.7. La fonction PLIB_SPI_PinEnable	8-12
8.3.1.8. La fonction PLIB_SPI_IsBusy	8-12
8.3.1.9. La fonction PLIB_SPI_FIFOInterruptModeSelect	8-13
8.3.2. Les fonctions du transmetteur	8-13
8.3.3. Les fonctions du récepteur	8-13
8.3.4. Les fonctions de "Data Transfer"	8-14
8.3.4.1. La fonction PLIB_SPI_BufferClear	8-14
8.3.4.2. La fonction PLIB_SPI_BufferRead	8-14
8.3.4.3. La fonction PLIB_SPI_BufferWrite	8-14
<b>8.4. Etude du driver SPI fourni par MHC</b>	<b>8-15</b>
8.4.1. Configuration du driver	8-15
8.4.2. La fonction DRV_SPI0_Initialize	8-15
8.4.3. La fonction DRV_SPI0_ReceiverBufferIsFull	8-16
8.4.4. La fonction DRV_SPI0_TransmitterBufferIsFull	8-16
8.4.5. La fonction DRV_SPI0_BufferAddWriteRead	8-16
<b>8.5. Réalisation de l'utilitaire SPI</b>	<b>8-18</b>
8.5.1. Réalisation de la fonction spi_write1	8-18
8.5.2. Réalisation de la fonction spi_read1	8-18
8.5.2.1. Séquence des actions de lecture	8-19
8.5.2.2. Observation du moment de lecture du tampon	8-20

<b>8.7. Périphériques SPI du Kit PIC32MX</b>	<b>8-21</b>
<b>8.8. Communication avec le Dac LTC2604</b>	<b>8-22</b>
8.8.1. Chip Select du LTC2604	8-22
8.8.2. Configuration SPI nécessaire au LTC2604	8-22
8.8.2.1. Fonction de configuration du SPI pour le DAC	8-23
8.8.3. Détail du SPIxCON	8-25
8.8.4. Sélection de la fréquence de SCK	8-27
8.8.5. Initialisation du LTC2604	8-27
8.8.6. Ecriture d'une valeur sur le LTC2604	8-28
8.8.7. Observation des signaux du DAC	8-29
8.8.7.1. Détail du 3ème octet	8-29
8.8.7.2. Signal sur le DAC	8-30
<b>8.10. Communication avec le LM70</b>	<b>8-31</b>
8.10.1. Connexions entre le PIC32 et le LM70	8-31
8.10.2. Configuration SPI nécessaire au LM70	8-31
8.10.2.1. Fonction de configuration du SPI pour le LM70	8-32
8.10.2.2. Vérification de la configuration avec SPI1CON	8-32
8.10.3. Initialisation du LM70	8-33
8.10.4. Lecture du LM70	8-34
8.10.5. La fonction SPI_ReadRawTempLM70	8-34
8.10.6. Ecriture vers le LM70	8-35
8.10.6.1. Exemple d'écriture	8-35
8.10.7. Observation des signaux du LM70	8-36
8.10.7.1. Vue de détail	8-37
<b>8.11. Utilisation combinée des 2 slaves SPI</b>	<b>8-38</b>
8.11.1. Fonction lecture LM70 avec reconfiguration	8-38
8.11.2. Fonction écriture LTC2604 avec reconfiguration	8-39
8.11.3. Utilisation et obtention des résultats	8-39
8.11.3.1. Contenu de la réponse à l'interruption du Timer1	8-39
8.11.3.2. Affichage température dans l'application	8-40
8.11.3.3. Problème avec les variables float dans l'interruption	8-40
8.11.4. Remarque sur les reconfigurations	8-41
<b>8.12. Fichiers à disposition</b>	<b>8-41</b>
<b>8.13. Conclusion</b>	<b>8-41</b>
<b>8.14. Historique des versions</b>	<b>8-42</b>
8.14.1. Version 1.0 mai 2014	8-42
8.14.2. Version 1.1 mai 2014	8-42
8.14.3. Version 1.5 mars 2015	8-42
8.14.4. Version 1.7 mai 2016	8-42
8.14.5. Version 1.8 mars 2017	8-42
8.14.6. Version 1.9 février 2018	8-42
8.14.7. Version 1.91 mars 2019	8-42

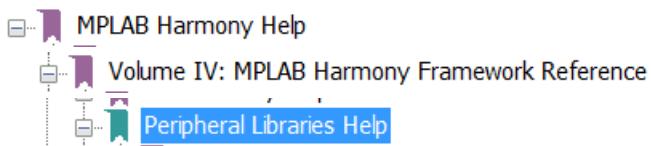
## 8. GESTION DU BUS SPI AVEC LE PIC32MX

Ce chapitre traite du bus SPI et de sa gestion avec le microcontrôleur PIC32MX795F512L ainsi que de la mise en pratique avec les composants LM70 et LTC2604.

Le PIC32MX dispose de modules de communication sérielle synchrone dédiés à la communication SPI (Serial Peripheral Interface).

Les documents de référence sont :

- La documentation "PIC32 Family Reference Manual" :  
Section 23 : Serial Peripheral Interface
- Pour les détails spécifiques au PIC32MX795F512L, il faut se référer au document "PIC32MX5XX/6XX/7XX Family Data Sheet" :  
Section 18 : Serial Peripheral Interface (SPI)
- La documentation d'Harmony, qui se trouve dans <Répertoire Harmony>\v<n>\doc :  
Section MPLAB Harmony Framework Reference > Peripheral Libraries Help,  
sous-section SPI Peripheral Library



Ce document a été établi sur la base de Harmony v1.06.

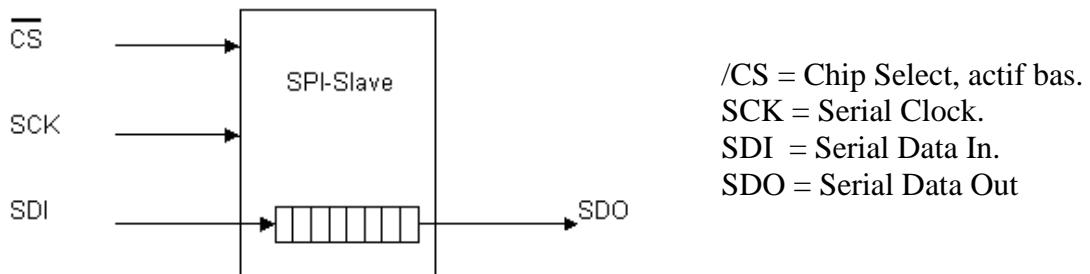
## 8.1. BUS SPI

Avant d'étudier comment gérer le bus SPI avec les microcontrôleurs PIC, il est nécessaire de connaître le bus SPI.

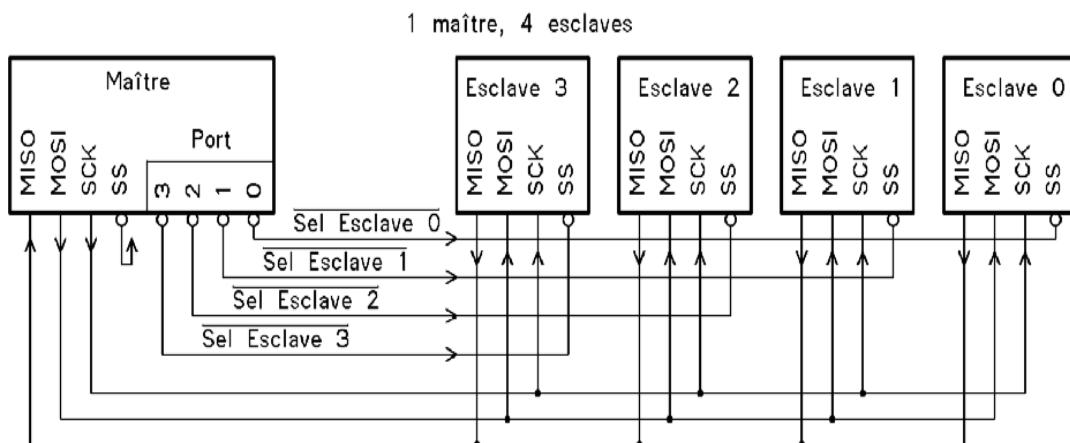
Nous allons présenter le rôle des lignes du bus, les modes et les signaux.

### 8.1.1. LES LIGNES DU BUS SPI

Voici les connexions standards sur un composant SPI, donc esclave.



### 8.1.2. CONNEXIONS ENTRE MASTER ET SLAVES

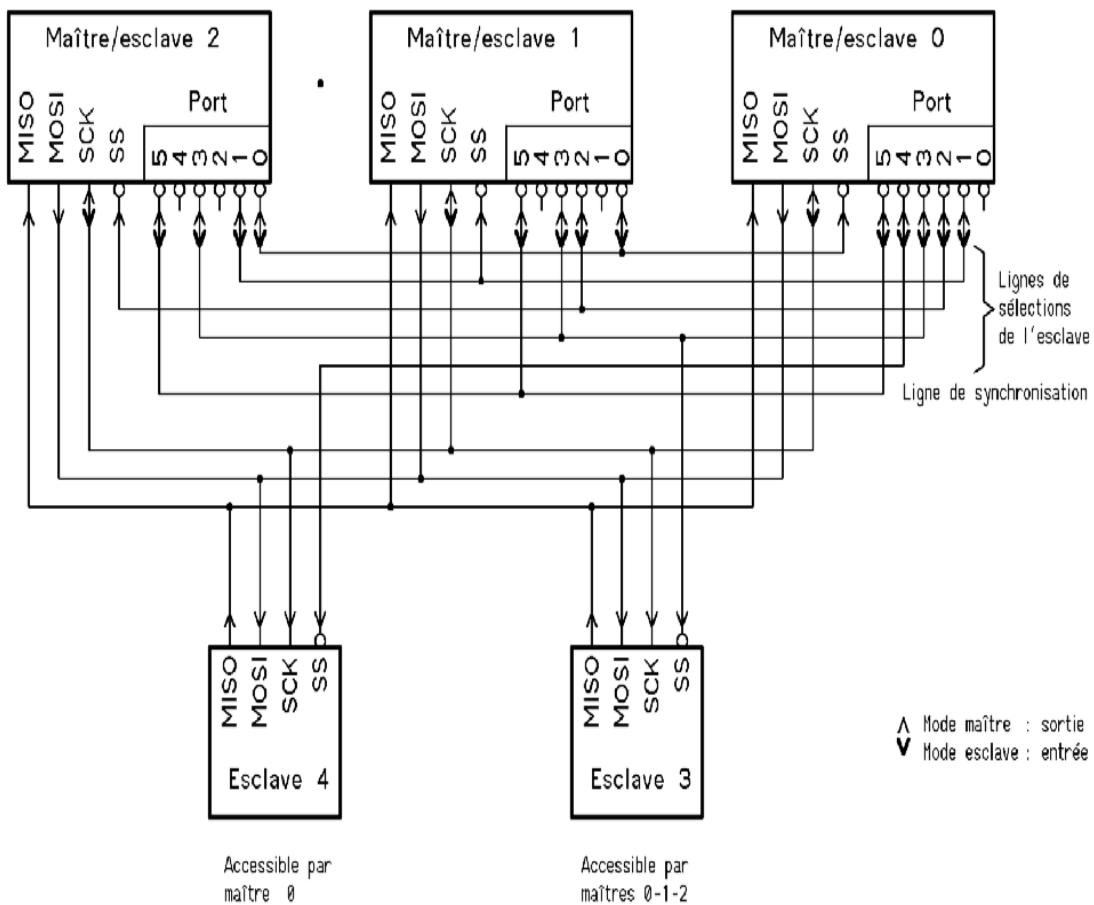


Motorola définit les lignes de la manière suivante :

- MISO = Master In, Slave Out.
- MOSI = Master Out, Slave In
- SCK = Serial ClocK
- SS = Slave Select

### 8.1.3. CONNEXIONS MULTI-MASTER

Il ne doit y avoir qu'un seul maître à la fois, mais les rôles peuvent être permutés. Cela implique le passage pour le SCK de sortie à entrée lorsque le maître devient esclave.



### 8.1.4. LES MODES MASTER ET SLAVE

En mode maître, le composant (en général le microcontrôleur) doit fournir l'horloge et activer la sélection de l'esclave.

En mode esclave, le composant (en général un périphérique) reçoit l'horloge et réagit à l'entrée de sélection.

En général un microcontrôleur peut être configuré en maître ou en esclave, c'est le cas du PIC.

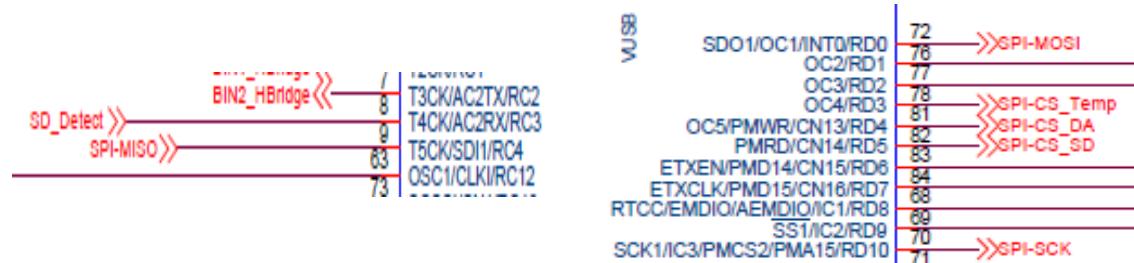
## 8.2. RÉALISATION DU BUS SPI AVEC LE PIC32MX

Le PIC32MX795F512L possède 4 modules SPI. Voici l'attribution des broches.

Pin Name	Pin Number <sup>(1)</sup>			Pin Type	Buffer Type	Description
	64-Pin QFN/TQFP	100-Pin TQFP	121-Pin XBGA			
SCK1	—	70	D11	I/O	ST	Synchronous serial clock input/output for SPI1
SDI1	—	9	E1	I	ST	SPI1 data in
SDO1	—	72	D9	O	—	SPI1 data out
SS1	—	69	E10	I/O	ST	SPI1 slave synchronization or frame pulse I/O
SCK3	49	48	K9	I/O	ST	Synchronous serial clock input/output for SPI3
SDI3	50	52	K11	I	ST	SPI3 data in
SDO3	51	53	J10	O	—	SPI3 data out
SS3	43	47	L9	I/O	ST	SPI3 slave synchronization or frame pulse I/O
SCK2	4	10	E3	I/O	ST	Synchronous serial clock input/output for SPI2
SDI2	5	11	F4	I	ST	SPI2 data in
SDO2	6	12	F2	O	—	SPI2 data out
SS2	8	14	F3	I/O	ST	SPI2 slave synchronization or frame pulse I/O
SCK4	29	39	L6	I/O	ST	Synchronous serial clock input/output for SPI4
SDI4	31	49	L10	I	ST	SPI4 data in
SDO4	32	50	L11	O	—	SPI4 data out
SS4	21	40	K6	I/O	ST	SPI4 slave synchronization or frame pulse I/O

### 8.2.1. CÂBLAGE DU SPI SUR LE KITPIC32MX

Comme on peut le voir dans les éléments de schéma ci-dessous, le kit utilise le module SPI 1 :

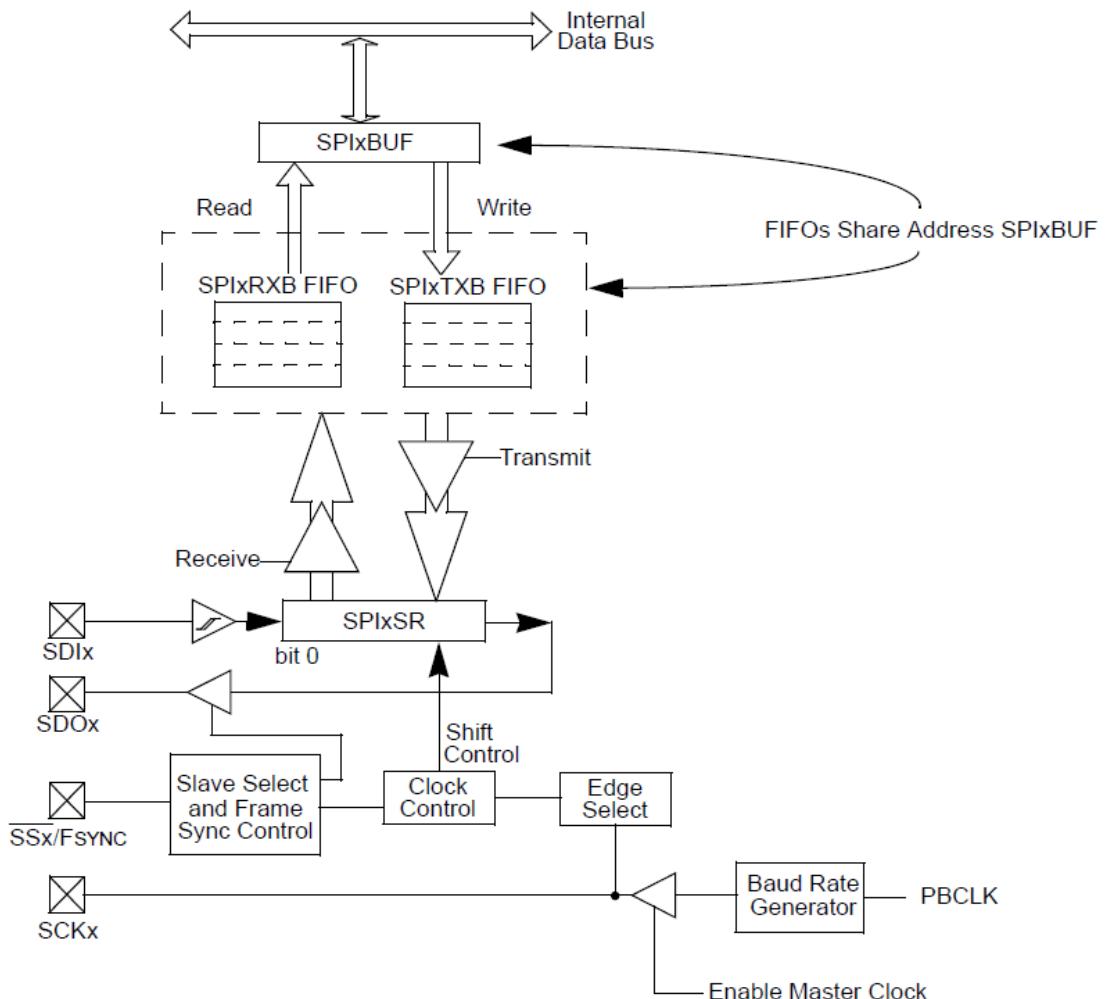


Broche du PIC32MX795F512L (100 pin TQFP)	Nom schéma	No broche boîtier 100
SCK1/RD10	SPI-SCK	70
SDO1/RD0	SPI-MOSI	72
SDI1/RC4	SPI-MISO	9
SS1/RD9	DAC_CLR	69

⚠ Remarque : Le slave select 1 \_SS1 est utilisé pour le DAC !

### 8.2.2. SCHÉMA-BLOC DU MODULE SPI

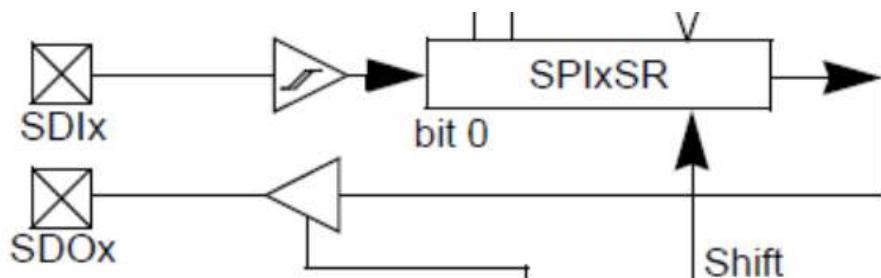
Voici le schéma bloc du module SPI :



**Note:** Access SPIxTXB and SPIxRXB FIFOs via SPIxBUF register.

### 8.2.3. PARTICULARITÉ DU SPIxSR

Comme le montre l'extrait ci-dessous, le registre à décalage SPIxSR va sortir un bit du registre pour chaque bit reçu. Cela signifie que lors de l'écriture, le slave fournit une info. Une transmission SPI est full-duplex.



Cette particularité est à gérer en relation avec les spécificités de chaque slave SPI.

### 8.2.4. LES POSSIBILITÉS DU MODULE SPI

Le tableau ci-dessous résume les possibilités du module SPI.

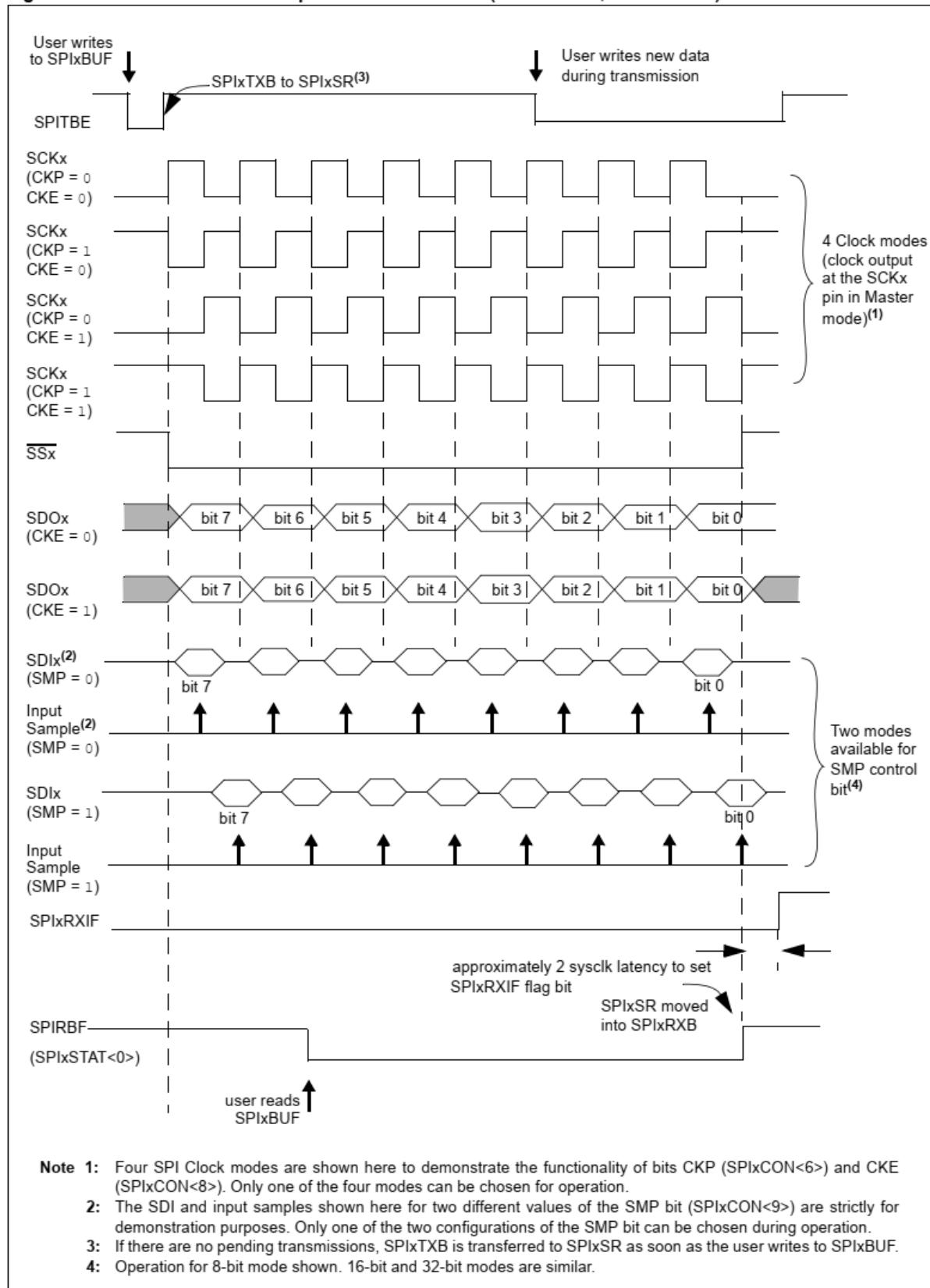
**Table 23-1: SPI Features**

Available SPI Modes	SPI Master	SPI Slave	Frame Master	Frame Slave	8-Bit, 16-Bit, and 32-Bit Modes	Selectable Clock Pulses and Edges	Selectable Frame Sync Pulses and Edges	Slave Select Pulse
Normal Mode	Yes	Yes	—	—	Yes	Yes	—	Yes
Framed Mode	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No

On en déduit la possibilité de travailler en mode 8, 16 ou 32 bits. Le mode "Framed" ne sera pas traité dans ce chapitre.

### 8.2.5. TIMING EN MODE MASTER (8 BITS)

Figure 23-9: SPI Master Mode Operation in 8-bit Mode (MODE32 = 0, MODE16 = 0)

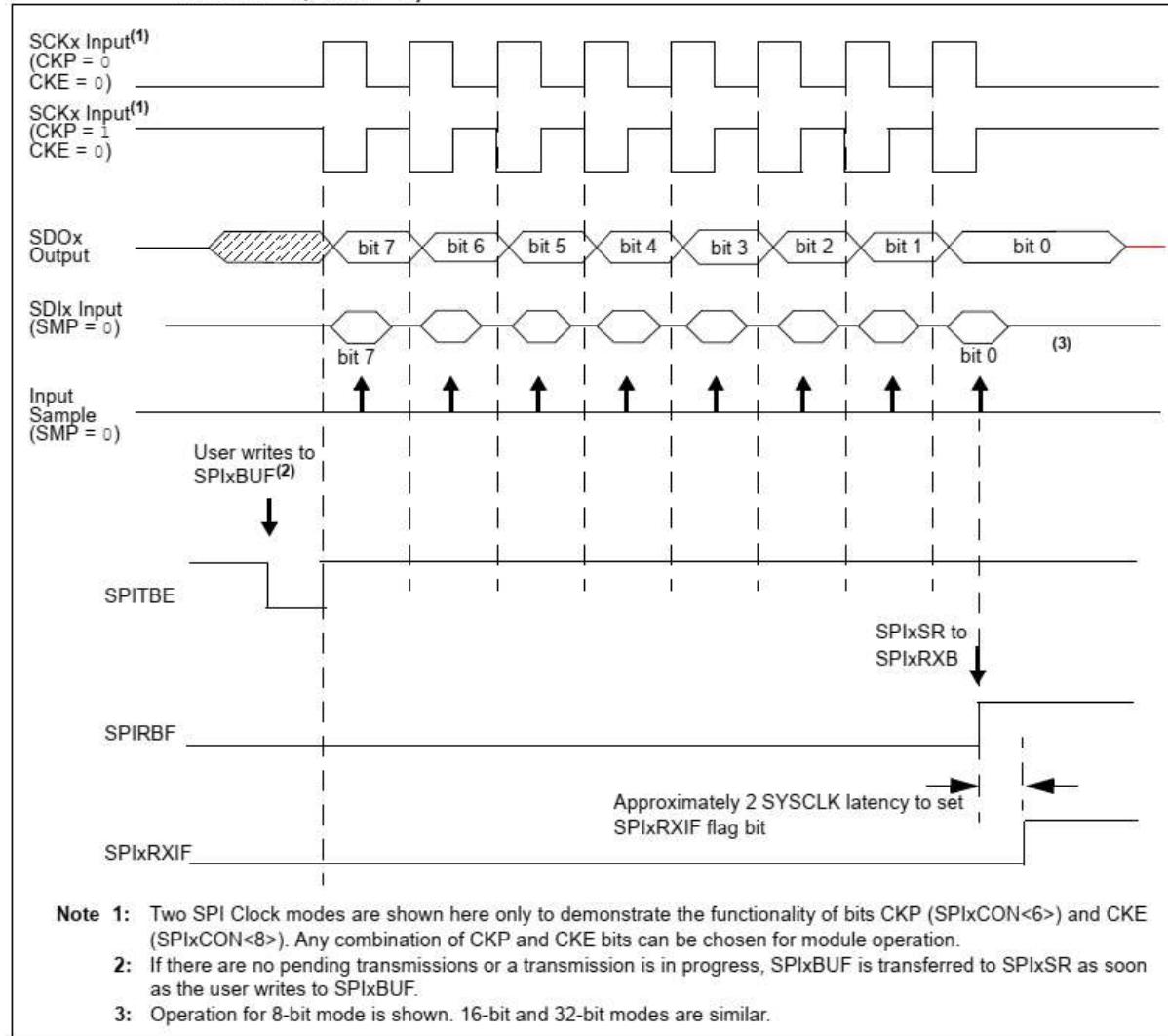


On peut observer les possibilités de configuration : les flancs de l'horloge en relation avec la donnée, ce qui permet d'adapter le master à un slave demandant un des modes. En mode 8 bits, le module SPI sérialise/dé-sérialise un octet à la fois.

## 8.2.6. TIMING EN MODE SLAVE

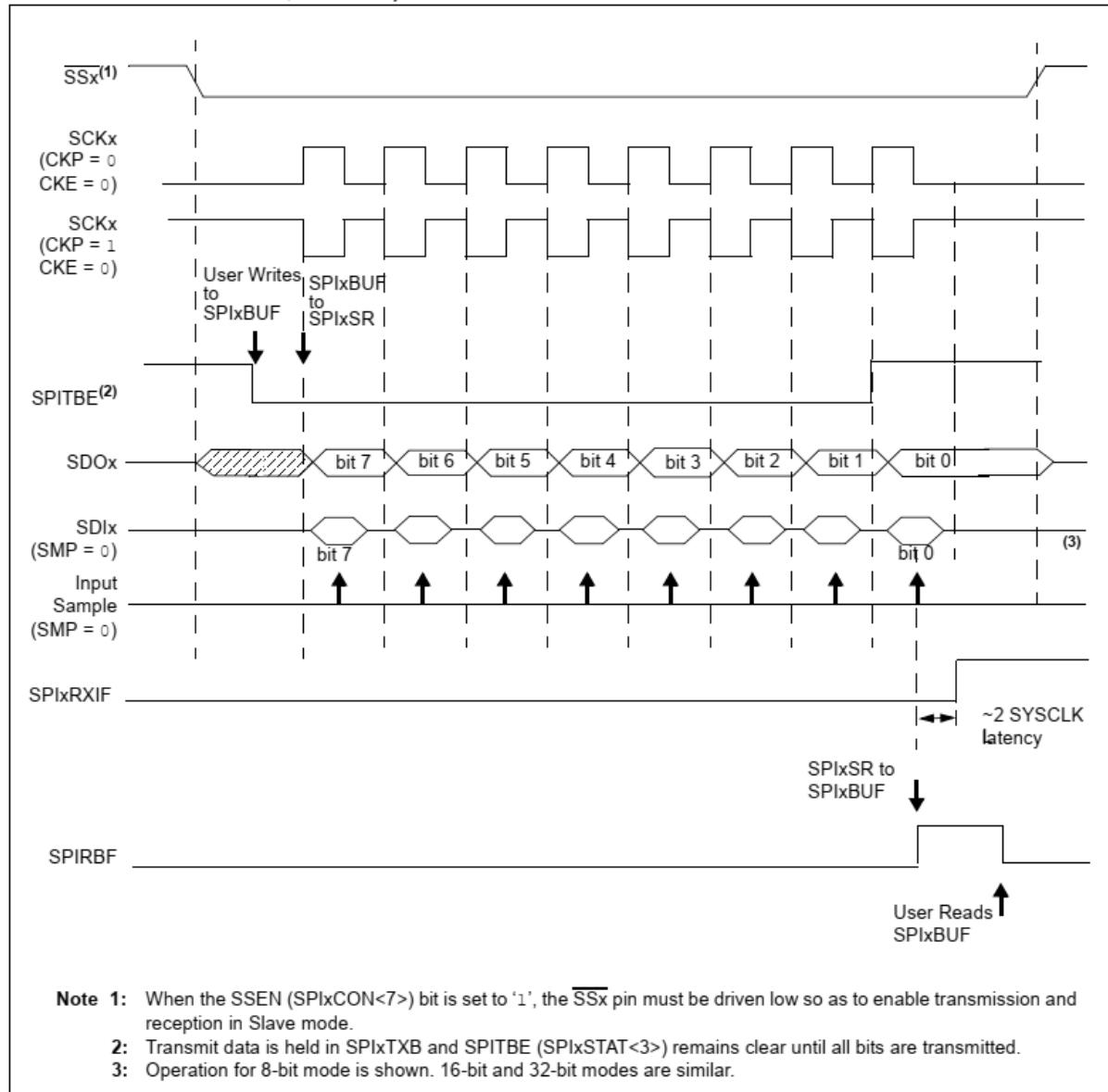
### 8.2.6.1. SLAVE SELECT PIN DISABLED

Figure 23-10: SPI Slave Mode Operation in 8-bit Mode with Slave Select Pin Disabled (MODE32 = 0, MODE16 = 0, SSEN = 0)



### 8.2.6.2. SLAVE SELECT PIN ENABLED

Figure 23-11: SPI Slave Mode Operation in 8-bit Mode with Slave Select Pin Enabled (MODE32 = 0, MODE16 = 0, SSEN = 1)



## 8.3. LES FONCTIONS SPI DE LA PLIB\_SPI

Avec l'introduction de Harmony, les fonctions des librairies périphériques ne font plus partie du compilateur. La PLIB\_SPI est décrite au paragraphe SPI Peripheral Library, dans la documentation de Harmony (version 1.06 ci-dessous).

### 8.3.1. LES FONCTIONS DE CONFIGURATION

Nous disposons de nombreuses fonctions de configuration dont la description sera limitée à celles demandant des explications.

Name	Description
<a href="#">PLIB_SPI_BaudRateClockSelect</a>	Selects the type of clock is used by the Baud Rate Generator.
<a href="#">PLIB_SPI_BaudRateSet</a>	Sets the baud rate to the desired value.
<a href="#">PLIB_SPI_ClockPolaritySelect</a>	Enables clock polarity.
<a href="#">PLIB_SPI_CommunicationWidthSelect</a>	Selects the data width for the SPI communication.
<a href="#">PLIB_SPI_Disable</a>	Disables the SPI module.
<a href="#">PLIB_SPI_Enable</a>	Enables the SPI module.
<a href="#">PLIB_SPI_ErrorInterruptDisable</a>	Enables SPI error interrupts.
<a href="#">PLIB_SPI_ErrorInterruptEnable</a>	Enables SPI error interrupts
<a href="#">PLIB_SPI_FIFOCountGet</a>	Reads the SPI Buffer Element Count bits for either receive or transmit.
<a href="#">PLIB_SPI_FIFODisable</a>	Disables the SPI enhanced buffer.
<a href="#">PLIB_SPI_FIFOEnable</a>	Enables the SPI enhanced buffer.
<a href="#">PLIB_SPI_FIFOInterruptModeSelect</a>	Selects the SPI buffer interrupt mode.
<a href="#">PLIB_SPI_FIFOShiftRegisterIsEmpty</a>	Returns the current status of the SPI shift register.
<a href="#">PLIB_SPI_InputSamplePhaseSelect</a>	Selects the SPI data input sample phase.
<a href="#">PLIB_SPI_IsBusy</a>	Returns the current SPI module activity status.
<a href="#">PLIB_SPI_MasterEnable</a>	Enables the SPI in Master mode.
<a href="#">PLIB_SPI_OutputDataPhaseSelect</a>	Selects serial output data change.
<a href="#">PLIB_SPI_PinDisable</a>	Enables the selected SPI pins.
<a href="#">PLIB_SPI_PinEnable</a>	Enables the selected SPI pins.
<a href="#">PLIB_SPI_PrescalePrimarySelect</a>	Selects the primary prescale for SPI Master mode.
<a href="#">PLIB_SPI_PrescaleSecondarySelect</a>	Selects the secondary prescale for SPI Master mode.
<a href="#">PLIB_SPI_ReadDatasSignExtended</a>	Returns the current status of the receive (RX) FIFO sign-extended data.
<a href="#">PLIB_SPI_SlaveEnable</a>	Enables the SPI in Slave mode.
<a href="#">PLIB_SPI_SlaveSelectDisable</a>	Disables Master mode slave select.
<a href="#">PLIB_SPI_SlaveSelectEnable</a>	Enables Master mode slave select.
<a href="#">PLIB_SPI_StopInIdleDisable</a>	Continues module operation when the device enters Idle mode.
<a href="#">PLIB_SPI_StopInIdleEnable</a>	Discontinues module operation when the device enters Idle mode.

#### 8.3.1.1. LA FONCTION PLIB\_SPI\_BAUDRATECLOCKSELECT

⌚ Cette fonction n'est pas supportée par le PIC32MX.

#### 8.3.1.2. LA FONCTION PLIB\_SPI\_BAUDRATESET

La fonction PLIB\_SPI\_BaudRateSet, permet d'établir la fréquence du SCK. Voici son prototype:

```
void PLIB_SPI_BaudRateSet(SPI_MODULE_ID index,
                           uint32_t clockFrequency,
                           uint32_t baudRate)
```

Il faut fournir la fréquence de l'horloge (en principe PB\_CLOCK) ainsi que la fréquence voulue. Il faut cependant veiller à ce que le rapport des fréquences soit un nombre entier pair (2, 4, 6, 8 , ...).

Par exemple avec l'horloge à 80 MHz pour obtenir SCK à 20 MHz le rapport est de 4, on écrira :

```
PLIB_SPI_BaudRateSet(KitSpi1, SYS_CLK_FREQ, 20000000);
```

### 8.3.1.3. LA FONCTION PLIB\_SPI\_COMMUNICATIONWIDTHSELECT

La fonction **PLIB\_SPI\_CommunicationWidthSelect** permet d'établir si la communication est effectuée par paquets de 8, 16 ou 32 bits. Voici son prototype :

```
void PLIB_SPI_CommunicationWidthSelect(SPI_MODULE_ID index,
                                         SPI_COMMUNICATION_WIDTH width)
```

Le type énuméré SPI\_COMMUNICATION\_WIDTH permet le choix.

```
typedef enum {
    SPI_COMMUNICATION_WIDTH_8BITS = 0,
    SPI_COMMUNICATION_WIDTH_16BITS = 1,
    SPI_COMMUNICATION_WIDTH_32BITS = 2
} SPI_COMMUNICATION_WIDTH;
```

### 8.3.1.4. LA FONCTION PLIB\_SPI\_CLOCKPOLARITYSELECT

La fonction **PLIB\_SPI\_ClockPolaritySelect** permet de sélectionner la polarité de l'horloge.

```
void PLIB_SPI_ClockPolaritySelect(SPI_MODULE_ID index,
                                   SPI_CLOCK_POLARITY polarity)
```

Le type énuméré SPI\_CLOCK\_POLARITY permet le choix entre 2 polarités :

```
typedef enum {
    SPI_CLOCK_POLARITY_IDLE_LOW = 0,
    SPI_CLOCK_POLARITY_IDLE_HIGH = 1
} SPI_CLOCK_POLARITY;
```

Cette fonction agit sur le bit CKP du registre SPIxCON.

bit 6	<b>CKP:</b> Clock Polarity Select bit
	1 = Idle state for clock is a high level; active state is a low level
	0 = Idle state for clock is a low level; active state is a high level

### 8.3.1.5. LA FONCTION PLIB\_SPI\_INPUTSAMPLEPHASESELECT

La fonction **PLIB\_SPI\_InputSamplePhaseSelect** permet d'établir à quel moment les bits en entrée sont échantillonnés. Son prototype est le suivant :

```
void PLIB_SPI_InputSamplePhaseSelect(SPI_MODULE_ID index,
                                      SPI_INPUT_SAMPLING_PHASE phase)
```

Le type énuméré SPI\_INPUT\_SAMPLING\_PHASE permet le choix entre 2 variantes :

```
typedef enum {
    SPI_INPUT_SAMPLING_PHASE_IN_MIDDLE = 0,
    SPI_INPUT_SAMPLING_PHASE_AT_END = 1
} SPI_INPUT_SAMPLING_PHASE;
```

Cette fonction agit sur le bit SMP du registre SPIxCON.

**bit 9 SMP: SPI Data Input Sample Phase bit**

Master mode (MSTEN = 1):

1 = Input data sampled at end of data output time  
0 = Input data sampled at middle of data output time

Slave mode (MSTEN = 0):

SMP value is ignored when SPI is used in Slave mode. The module always uses SMP = 0.

### **8.3.1.6. LA FONCTION PLIB\_SPI\_OUTPUTDATAPHASESELECT**

La fonction **PLIB\_SPI\_OutputDataPhaseSelect** permet d'établir sur quelle transition de l'horloge les données sont émises. Son prototype est le suivant :

Le type énuméré SPI\_OUTPUT\_DATA\_PHASE permet d'indiquer le choix.

```
typedef enum {
    SPI_OUTPUT_DATA_PHASE_ON_IDLE_TO_ACTIVE_CLOCK = 0,
    SPI_OUTPUT_DATA_PHASE_ON_ACTIVE_TO_IDLE_CLOCK = 1
} SPI_OUTPUT_DATA_PHASE;
```

Cette fonction agit sur le bit CKE du registre SPIxCON.

**bit 8 CKE: SPI Clock Edge Select bit**  
1 = Serial output data changes on transition from active clock state to Idle clock state (see CKP bit)  
0 = Serial output data changes on transition from Idle clock state to active clock state (see CKP bit)

### **8.3.1.7. LA FONCTION PLIB\_SPI\_PINENABLE**

La fonction PLIB\_SPI\_PinEnable permet d'activer les lignes SPI mentionnées.

```
void PLIB_SPI_PinEnable(SPI_MODULE_ID index, SPI_PIN pin)
```

Le type énuméré SPI\_PIN permet d'indiquer la ligne à activer.

```
typedef enum {
    SPI_PIN_DATA_OUT,
    SPI_PIN_DATA_IN,
    SPI_PIN_SLAVE_SELECT
} SPI_PIN;
```

### **8.3.1.8. LA FONCTION PLIB\_SPI\_IsBusy**

La fonction **PLIB\_SPI\_IsBusy** n'est pas une fonction de configuration, elle permet de savoir si le system SPI est actif (sérialisation/desérialisation ou au repos).

```
bool PLIB_SPI_IsBusy(SPI_MODULE_ID index);
```

## Returns

- true - SPI module is currently busy with some transactions
  - false - SPI module is currently idle

### 8.3.1.9. LA FONCTION PLIB\_SPI\_FIFOINTERRUPTMODESELECT

La fonction **PLIB\_SPI\_FIFOInterruptModeSelect** permet de configurer le comportement de l'interruption SPI en relation avec la situation des tampons de réception et d'émission, ceci pour autant que l'on ait activé l'utilisation du buffer avancé (à l'aide de la fonction **PLIB\_SPI\_FIFONable**).

Son prototype est le suivant :

```
void PLIB_SPI_FIFOInterruptModeSelect(SPI_MODULE_ID index,
                                      SPI_FIFO_INTERRUPT mode)
```

Le type énuméré SPI\_FIFO\_INTERRUPT permet les choix suivants :

```
typedef enum {
    SPI_FIFO_INTERRUPT_WHEN_TRANSMISSION_IS_COMPLETE = 3,
    SPI_FIFO_INTERRUPT_WHEN_BUFFER_IS_EMPTY = 7,
    SPI_FIFO_INTERRUPT_WHEN_TRANSMIT_BUFFER_IS_NOT_FULL = 0,
    SPI_FIFO_INTERRUPT_WHEN_TRANSMIT_BUFFER_IS_1HALF_EMPTY_OR_MORE
                                = 1,
    SPI_FIFO_INTERRUPT_WHEN_TRANSMIT_BUFFER_IS_COMpletely_EMPTY
                                = 2,
    SPI_FIFO_INTERRUPT_WHEN_RECEIVE_BUFFER_IS_FULL = 4,
    SPI_FIFO_INTERRUPT_WHEN_RECEIVE_BUFFER_IS_1HALF_FULL_OR_MORE
                                = 5,
    SPI_FIFO_INTERRUPT_WHEN_RECEIVE_BUFFER_IS_NOT_EMPTY = 6
} SPI_FIFO_INTERRUPT;
```

L'utilisation de l'interruption est particulièrement utile en mode SLAVE pour réagir rapidement aux données reçues afin de répondre.

### 8.3.2. LES FONCTIONS DU TRANSMETTEUR

Les 3 fonctions retournent un booléen indiquant la situation du tampon de transmission.

Name	Description
<a href="#">PLIB_SPI_TransmitBufferIsEmpty</a>	Returns the current status of the transmit buffer.
<a href="#">PLIB_SPI_TransmitBufferIsFull</a>	Returns the current transmit buffer status of the SPI module.
<a href="#">PLIB_SPI_TransmitUnderRunStatusGet</a>	Returns the current status of the transmit underrun.

### 8.3.3. LES FONCTIONS DU RÉCEPTEUR

Les 3 premières fonctions retournent un booléen indiquant la situation du tampon de réception. Il est à remarquer un manque de cohérence dans les noms des fonctions puisque le "Buffer" devient "FIFO" lorsque l'on cherche à savoir s'il est vide.

Name	Description
<a href="#">PLIB_SPI_ReceiverBufferIsFull</a>	Returns the current status of the SPI receive buffer.
<a href="#">PLIB_SPI_ReceiverFIFOIsEmpty</a>	Returns the current status of the SPI receive FIFO.
<a href="#">PLIB_SPI_ReceiverHasOverflowed</a>	Returns the current status of the SPI receiver overflow.
<a href="#">PLIB_SPI_ReceiverOverflowClear</a>	Clears the SPI receive overflow flag.

### 8.3.4. LES FONCTIONS DE "DATA TRANSFER"

Le système SPI dispose d'un tampon pour la transmission et la réception, d'où l'existence de fonction pour la gestion de ce tampon (Buffer).

Name	Description
<code>PLIB_SPI_BufferClear</code>	Clears the SPI buffer.
<code>PLIB_SPI_BufferRead</code>	Returns the SPI buffer value.
<code>PLIB_SPI_BufferAddressGet</code>	Returns the address of the SPIxBUF (Transmit(SPIxTXB) and Receive (SPIxRXB)) register.
<code>PLIB_SPI_BufferRead16bit</code>	Returns 16-bit SPI buffer value.
<code>PLIB_SPI_BufferRead32bit</code>	Returns 32-bit SPI buffer value.
<code>PLIB_SPI_BufferWrite</code>	Write the data to the SPI buffer.
<code>PLIB_SPI_BufferWrite16bit</code>	Writes 16-bit data to the SPI buffer.
<code>PLIB_SPI_BufferWrite32bit</code>	Write 32-bit data to the SPI buffer.

Nous limitons l'étude aux fonctions de transfert 8 bits ainsi qu'à la fonction d'effacement du buffer.

#### 8.3.4.1. LA FONCTION `PLIB_SPI_BUFFERCLEAR`

La fonction `PLIB_SPI_BufferClear` permet d'effacer le contenu du tampon (émission et réception). Exemple :

```
PLIB_SPI_BufferClear(KitSpi1);
```

#### 8.3.4.2. LA FONCTION `PLIB_SPI_BUFFERREAD`

La fonction `PLIB_SPI_BufferRead` permet d'obtenir un élément 8 bits du tampon de réception. Son prototype est le suivant :

```
uint8_t PLIB_SPI_BufferRead(SPI_MODULE_ID index)
```

⚠ Avant de lire il faut s'assurer que le tampon ne soit pas vide !

#### 8.3.4.3. LA FONCTION `PLIB_SPI_BUFFERWRITE`

La fonction `PLIB_SPI_BufferWrite` permet de déposer un élément 8 bits dans le tampon de transmission. Son prototype est le suivant :

```
void PLIB_SPI_BufferWrite(SPI_MODULE_ID index,
                           uint8_t data)
```

⚠ Avant d'écrire, il faut s'assurer que le tampon ne soit pas plein !

## 8.4. ETUDE DU DRIVER SPI FOURNI PAR MHC

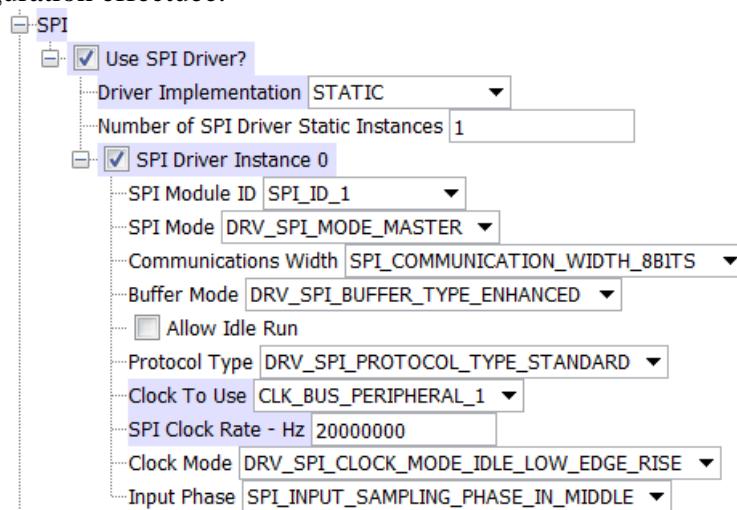
L'étude du driver SPI static généré par le MHC) nous fournit un exemple d'utilisation des fonctions de la PLIB\_SPI.

Cette partie a été réalisée avec les logiciels suivants :

- Harmony v1.06
- MPLABX IDE v3.10
- XC32 v1.40

### 8.4.1. CONFIGURATION DU DRIVER

Voici la configuration effectuée.



Les modifications apportées sont l'utilisation du CLK\_BUS\_PERIPHERAL\_1 au lieu du 2 et une réduction du SPI Clock Rate à 20 MHz.

Le choix des valeurs pour les sections Clock Mode et Input Phase sont à effectuer en relation avec le composant slave que l'on va utiliser.

### 8.4.2. LA FONCTION DRV\_SPI0\_INITIALIZE

Voici le contenu de la fonction **DRV\_SPI0\_Initialize**

```
void DRV_SPI0_Initialize(void)
{
    PLIB_SPI_Disable(SPI_ID_1);
    PLIB_SPI_MasterEnable(SPI_ID_1);
    PLIB_SPI_SlaveSelectEnable(SPI_ID_1);
    PLIB_SPI_StopInIdleDisable(SPI_ID_1);
    PLIB_SPI_ClockPolaritySelect(SPI_ID_1,
                                  SPI_CLOCK_POLARITY_IDLE_LOW);
    PLIB_SPI_OutputDataPhaseSelect(SPI_ID_1,
                                   SPI_OUTPUT_DATA_PHASE_ON_IDLE_TO_ACTIVE_CLOCK);
    PLIB_SPI_InputSamplePhaseSelect(SPI_ID_1,
                                    SPI_INPUT_SAMPLING_PHASE_IN_MIDDLE);
    PLIB_SPI_CommunicationWidthSelect(SPI_ID_1,
                                       SPI_COMMUNICATION_WIDTH_8BITS);
    PLIB_SPI_FramedCommunicationDisable( SPI_ID_1 );
    PLIB_SPI_FIFOEnable( SPI_ID_1 );
}
```

```

    PLIB_SPI_BaudRateSet(SPI_ID_1,
        SYS_CLK_PeripheralFrequencyGet
            (CLK_BUS_PERIPHERAL_1), 20000000);
    PLIB_SPI_Enable(SPI_ID_1);
}

```

Nous reprendrons le principe de la configuration effectuée dans DRV\_SPI0\_Initialize pour l'appliquer à des fonctions d'initialisation spécifiques au DAC et au LM70.

#### 8.4.3. LA FONCTION DRV\_SPI0\_RECEIVERBUFFERISFULL

La fonction DRV\_SPI0\_ReceiverBufferIsFull nous montre comment tester si le tampon de réception est plein :

```

bool DRV_SPI0_ReceiverBufferIsFull(void)
{
    return (PLIB_SPI_ReceiverBufferIsFull(SPI_ID_1));
}

```

#### 8.4.4. LA FONCTION DRV\_SPI0\_TRANSMITTERBUFFERISFULL

La fonction DRV\_SPI0\_TransmitterBufferIsFull nous montre comment tester si le tampon d'émission est plein :

```

bool DRV_SPI0_TransmitterBufferIsFull(void)
{
    return (PLIB_SPI_TransmitBufferIsEmpty(SPI_ID_1));
}

```

#### 8.4.5. LA FONCTION DRV\_SPI0\_BUFFERADDWRITEREAD

La fonction DRV\_SPI0\_BufferAddWriteRead nous montre comment gérer un échange bidirectionnel :

```

int32_t DRV_SPI0_BufferAddWriteRead(const void * txBuffer,
                                    void * rxBuffer, uint32_t size)
{
    bool continueLoop;
    int32_t txcounter = 0;
    int32_t rxcounter = 0;
    uint8_t unitsTxed = 0;
    const uint8_t maxUnits = 16;
    do {
        continueLoop = false;
        unitsTxed = 0;
        if (PLIB_SPI_TransmitBufferIsEmpty(SPI_ID_1))
        {
            while (!PLIB_SPI_TransmitBufferIsFull(SPI_ID_1)
                   && (txcounter < size)
                   && unitsTxed != maxUnits)
            {
                PLIB_SPI_BufferWrite(SPI_ID_1,
                    ((uint8_t*)txBuffer)[txcounter]);
                txcounter++;
                continueLoop = true;
            }
        }
    }
}

```

```
    unitsTxed++;
}
}

while (txcounter != rxcounter)
{
    while (PLIB_SPI_ReceiverFIFOIsEmpty(SPI_ID_1));
    ((uint8_t*) rxBuffer)[rxcounter] =
        PLIB_SPI_BufferRead(SPI_ID_1);
    rxcounter++;
    continueLoop = true;
}
if (txcounter > rxcounter)
{
    continueLoop = true;
}
}while(continueLoop);
return txcounter;
}
```

Cette fonction transmet un certain nombre de bytes, avec une limite fixée à 16, puis elle lit ce que le slave a fourni.

## 8.5. RÉALISATION DE L'UTILITAIRE SPI

Les fonctions utilitaires SPI s'inspirent des fonctions SPI du compilateur CCS. L'utilitaire ne s'occupe pas de l'initialisation car elle est spécifique pour chaque composant.

### 8.5.1. RÉALISATION DE LA FONCTION SPI\_WRITE1

Cette fonction effectue l'écriture de 8 bits de données sur le SPI canal 1. L'utilisateur doit gérer le Chip Select avant et après l'utilisation de la fonction.

Cette fonction est fournie dans les fichiers Mc32SpiUtil.c et Mc32SpiUtil.h.

```
void spi_write1( uint8_t Val) {
    bool SpiBusy;
    PLIB_SPI_BufferWrite(SPI_ID_1, Val);
    // Attente de la fin de la transmission
    do {
        SpiBusy = PLIB_SPI_IsBusy(SPI_ID_1) ;
    } while (SpiBusy == true);
}
```

La transmission s'effectue en déposant une valeur dans le tampon d'émission (TransmitBuffer) avec la fonction **PLIB\_SPI\_BufferWrite**.

☞ On aurait pu s'assurer que le tampon d'émission ne soit pas plein avant d'y déposer une nouvelle valeur (à l'aide de la fonction **PLIB\_SPI\_TransmitBufferIsFull**). Ceci n'est toutefois pas nécessaire car on attend la fin de la transmission avant de quitter la fonction. Ainsi, il y aura forcément de la place disponible dans le tampon d'émission au prochain appel de **spi\_write1**.

Le dépôt d'une valeur dans le tampon d'émission déclenche la transmission. Il est nécessaire d'attendre la fin de cette transmission pour gérer le /CS en synchronisation avec la transmission. Utilisation de la fonction **PLIB\_SPI\_IsBusy**.

### 8.5.2. RÉALISATION DE LA FONCTION SPI\_READ1

Cette fonction effectue la lecture de 8 bits de données sur le SPI canal 1. L'utilisateur doit gérer le Chip Select avant et après l'utilisation de la fonction.

Remarque : la fonction a en paramètre la valeur à transmettre et elle retourne la valeur lue.

```
uint8_t spi_read1( uint8_t Val) {
    int SpiBusy;
    uint32_t lu;

    PLIB_SPI_BufferWrite(SPI_ID_1, Val);
    // Attend fin transmission
    do {
        SpiBusy = PLIB_SPI_IsBusy(SPI_ID_1) ;
    } while (SpiBusy == 1);

    // Attend arrivée dans fifo
    while (PLIB_SPI_ReceiverFIFOIsEmpty(SPI_ID_1));
}
```

```
#ifdef MARKER_READ
    LED3_W = 1;
#endif
    lu = PLIB_SPI_BufferRead(SPI_ID_1);
#endif MARKER_READ
    LED3_W = 0;
#endif
    return lu;
}
```

Pour lire, il faut générer des coups d'horloge, ce qui s'effectue avec une fonction d'écriture. Au fur et à mesure que les coups d'horloge sont envoyés, le slave fournit les bits de données.

Il faut attendre la fin de la transmission avec **PLIB\_SPI\_IsBusy**.

Utilisation de la fonction **PLIB\_SPI\_ReceiverFIFOIsEmpty** pour attendre tant que le tampon de réception est vide. Ceci est important car si on lit trop vite on manque la valeur.

(?) Si les conditions de départ ne sont pas correctes (tampon vide, pas d'erreur Receive Overflow), il y a risque d'être bloqué ou de ne pas lire la bonne valeur.

L'action **PLIB\_SPI\_BufferClear(KitSpi1)** est nécessaire dans la configuration pour cela.

#### 8.5.2.1. SÉQUENCE DES ACTIONS DE LECTURE

Il s'agit d'une séquence de lecture de la température du LM70. La séquence est précédée de la reconfiguration, car l'application gère les 2 composants SPI.

```
SPI_ConfigureLM70();

CS_LM70 = 0;
MSB = spi_read1(0xFF);
LSB = spi_read1(0xFF);
//Fin de transmission
CS_LM70 = 1;
```

### 8.5.2.2. OBSERVATION DU MOMENT DE LECTURE DU TAMON

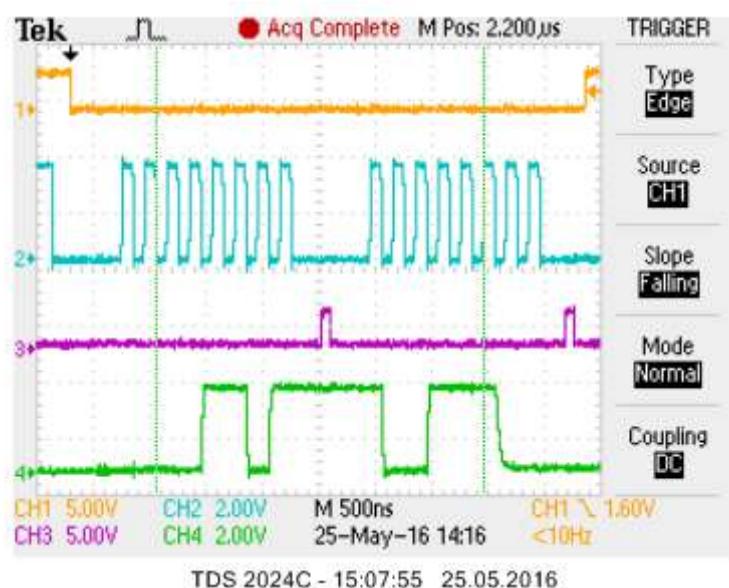
Pour vérifier à quel moment de la transmission s'effectue la lecture du tampon de réception, nous ajoutons un action sur la LED\_3 pour observer cela.

Canal 1 : CS\_LM70  
RD3  
broche 78

Canal 2 : SPI-SCK  
SCK1/RD10  
broche 70

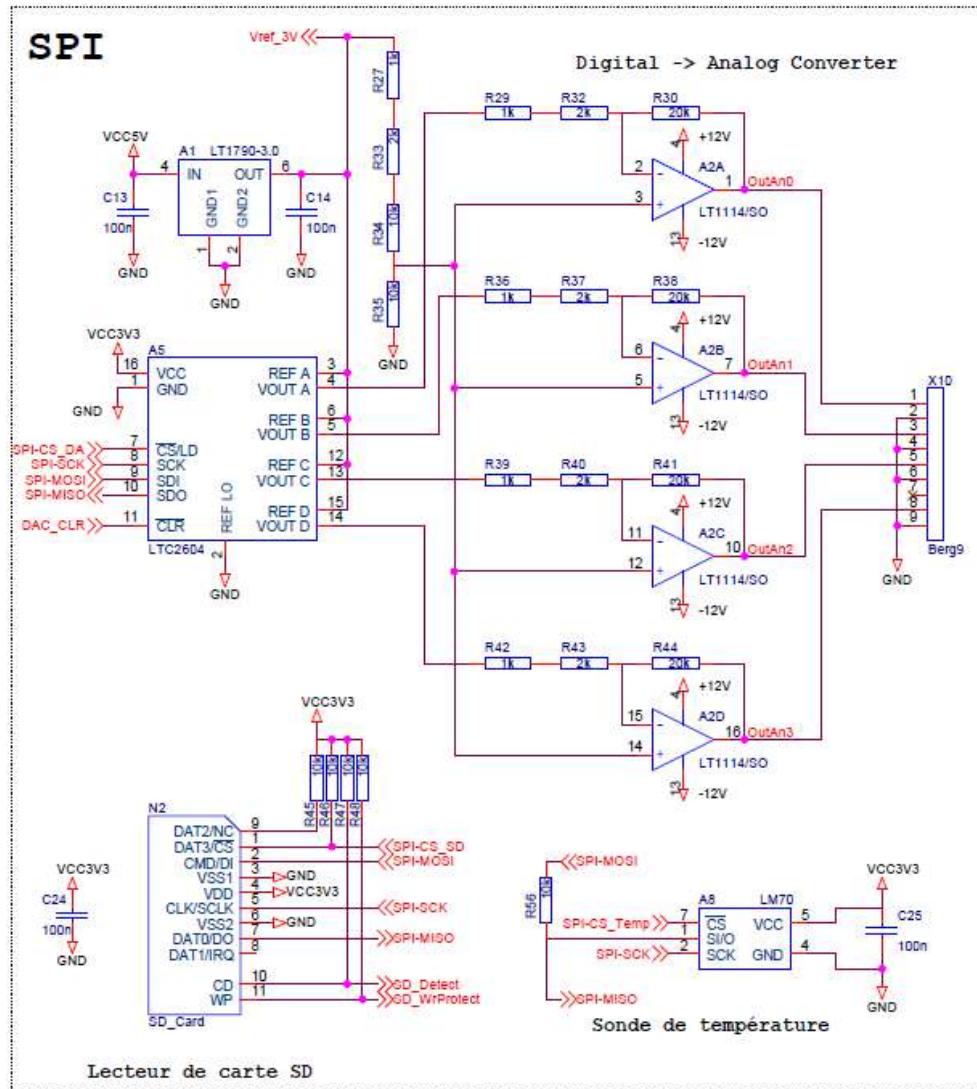
Canal 3 : LED\_3

Canal 4 : SPI-MISO  
SDI1/RC4  
broche 9



On peut observer que l'impulsion sur le canal 3 a lieu à la fin de la série de coups d'horloge. Donc les datas sont lues au bon moment, ce qui se confirme par le résultat en température.

## 8.7. PÉRIPHÉRIQUES SPI DU KIT PIC32MX



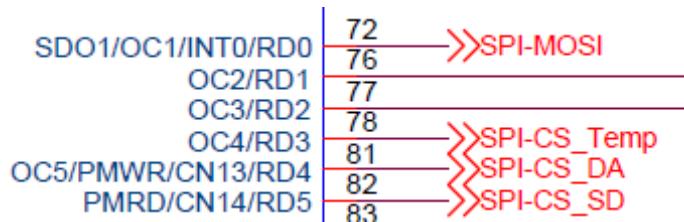
## 8.8. COMMUNICATION AVEC LE DAC LTC2604

Le dac LTC2604 utilise le module SPI 1.

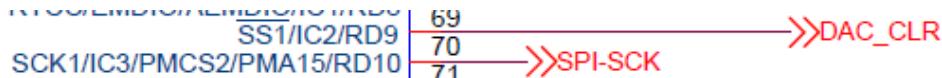
Broche du PIC32MX795F512L (100 pin TQFP)	Nom schéma	No broche boitier 100
SCK1/IC3/PMCS2/PMA15/RD10	SPI-SCK	70
SDO1/OC1/INT0/RD0	SPI-MOSI	72
T5CK/SDI1/RC4	SPI-MISO	9

### 8.8.1. CHIP SELECT DU LTC2604

Utilisation d'une ligne de port standard (RD4) pour le signal  $\overline{CS}$



On dispose aussi de RD9 pour effectuer un reset du composant.



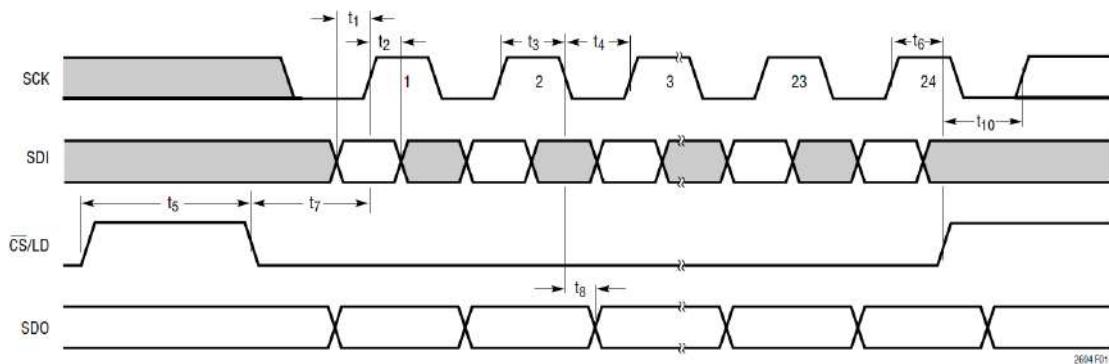
### 8.8.2. CONFIGURATION SPI NÉCESSAIRE AU LTC2604

Au niveau du timing, voici les caractéristiques :

SYMBOL	PARAMETER	CONDITIONS	LTC2604/LTC2614/LTC2624			UNITS
			MIN	Typ	MAX	
$V_{CC} = 2.5V \text{ to } 5.5V$						
$t_1$	SDI Valid to SCK Setup		●	4		ns
$t_2$	SDI Valid to SCK Hold		●	4		ns
$t_3$	SCK High Time		●	9		ns
$t_4$	SCK Low Time		●	9		ns
$t_5$	$\overline{CS}/LD$ Pulse Width		●	10		ns
$t_6$	LSB SCK High to $\overline{CS}/LD$ High		●	7		ns
$t_7$	$\overline{CS}/LD$ Low to SCK High		●	7		ns
$t_8$	SDO Propagation Delay from SCK Falling Edge	$C_{LOAD} = 10\text{pF}$ $V_{CC} = 4.5V \text{ to } 5.5V$ $V_{CC} = 2.5V \text{ to } 5.5V$	●		20 45	ns
$t_9$	CLR Pulse Width		●	20		ns
$t_{10}$	$\overline{CS}/LD$ High to SCK Positive Edge		●	7		ns
	SCK Frequency	50% Duty Cycle	●		50	MHz

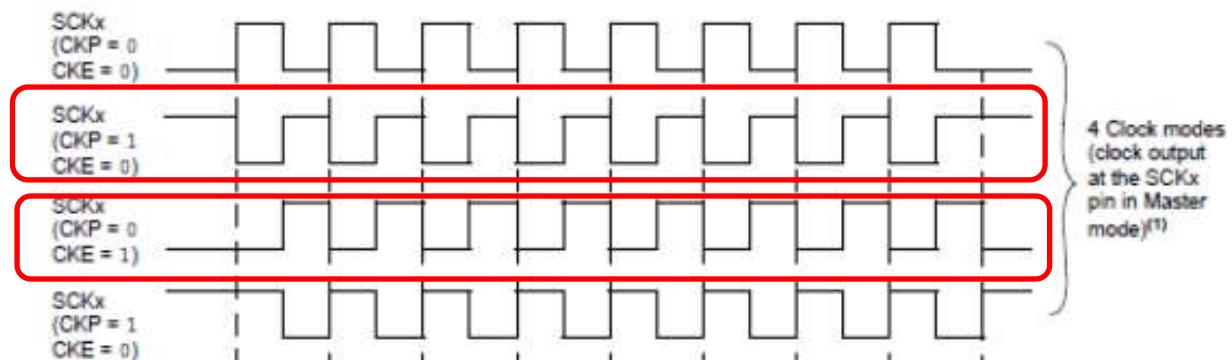
Avec la division du PB\_CLOCK par 2, 4, 6, 8, etc., il est possible de travailler à 40 MHz. En pratique, on travaillera à 20 MHz.

Pour le choix du mode d'horloge, le fabricant fournit un diagramme :



On doit donc avoir le flanc montant de l'horloge au milieu des datas.

Ce qui correspond à deux situations : CKP = 1 et CKE = 0 OU CKP = 0 et CKE = 1



### 8.8.2.1. FONCTION DE CONFIGURATION DU SPI POUR LE DAC

Voici le contenu de la fonction SPI\_ConfigureLTC2604 qui est fournie dans le fichier Mc32gestSPiDac.c. Cette fonction reprend les mêmes éléments que la fonction DRV\_SPI0\_Initialize, mais avec l'ajout d'une action PLIB\_SPI\_BufferClear.

```
void SPI_ConfigureLTC2604(void)
{
    PLIB_SPI_Disable(KitSpi1);
    PLIB_SPI_BufferClear(KitSpi1);
    PLIB_SPI_StopInIdleDisable(KitSpi1);
    PLIB_SPI_PinEnable(KitSpi1, SPI_PIN_DATA_OUT);
    PLIB_SPI_CommunicationWidthSelect(KitSpi1,
        SPI_COMMUNICATION_WIDTH_8BITS);
    // Config SPI clock à 20 MHz
    PLIB_SPI_BaudRateSet(KitSpi1,
        SYS_CLK_PeripheralFrequencyGet(CLK_BUS_PERIPHERAL_1),
        20000000);
    // Config polarité traitement des signaux SPI
    // pour input à confirmer
    // Polarité clock OK
    // Phase output à confirmer
    PLIB_SPI_InputSamplePhaseSelect(KitSpi1,
        SPI_INPUT_SAMPLING_PHASE_IN_MIDDLE );
}
```

```

PLIB_SPI_ClockPolaritySelect(KitSpi1,
                             SPI_CLOCK_POLARITY_IDLE_HIGH);
PLIB_SPI_OutputDataPhaseSelect(KitSpi1,
                             SPI_OUTPUT_DATA_PHASE_ON_IDLE_TO_ACTIVE_CLOCK);
PLIB_SPI_MasterEnable(KitSpi1);
PLIB_SPI_FramedCommunicationDisable(KitSpi1);
PLIB_SPI_FIFOEnable(KitSpi1); // Enhanced buffer mode
PLIB_SPI_Enable(KitSpi1);
// Contrôle de la configuration
ConfigReg = SPI1CON;
BaudReg = SPI1BRG;
}

```

Comme on peut le constater, les fonctions sont assez nombreuses, car elles correspondent aux groupes de bits du registre SPIxCON.

Pour vérifier la configuration on peut lire le registre de configuration.

[SPI1CON]	31	30	29	28	27	24	17
FRMEN	0	0	0	0	0	000	-
FRMSYNC	0	0	0	0	0	000	-
FRMPOL	0	0	0	0	0	000	-
MSSEN	0	0	0	0	0	000	-
FRMSYPW	0	0	0	0	0	000	-
FRMCNT	0	0	0	0	0	000	-
SPIFE	0	0	0	0	0	000	-
	16	15	13	12	11	10	9
ENHBUF	1	1	-	0	0	0	0
ON	1	1	-	0	0	0	0
SIDL	0	0	0	0	0	0	0
DISSDO	0	0	0	0	0	0	0
MODE32	0	0	0	0	0	0	0
MODE16	0	0	0	0	0	0	0
SMP	0	0	0	0	0	0	0
	8	7	6	5	2	0	0
CKE	0	0	1	1	-	00	00
SSEN	0	0	1	1	-	00	00
CKP	0	0	1	1	-	00	00
MSTEN	0	0	1	1	-	00	00
STXISEL	0	0	1	1	-	00	00
SRXISEL	0	0	1	1	-	00	00

On obtient la valeur 0x00018020, ce qui nous donne CKP = 1 et CKE = 0. Cela correspond au premier choix avec l'horloge qui démarre au niveau haut.

Si on modifie :

```

PLIB_SPI_ClockPolaritySelect(KitSpi1,
                             SPI_CLOCK_POLARITY_IDLE_HIGH);

```

En :

```

PLIB_SPI_ClockPolaritySelect(KitSpi1,
                             SPI_CLOCK_POLARITY_IDLE_LOW);

```

On obtient CKP = 0 et CKE = 0 et on n'obtient plus de signal sur la sortie du DAC !

### 8.8.3. DÉTAIL DU SPIxCON

 Register 23-1: SPIxCON: SPI Control Register<sup>(1,2,3)</sup>

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
FRMEN	FRMSYNC	FRMPOL	MSSEN <sup>(4)</sup>	FRMSYPW <sup>(4)</sup>	FRMCNT<2:0> <sup>(4)</sup>		
bit 31							bit 24

r-x	r-x	r-x	r-x	r-x	r-x	R/W-0	R/W-0
—	—	—	—	—	—	SPIFE	ENHBUF <sup>(4)</sup>
bit 23							bit 16

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ON	FRZ	SIDL	DISSDO	MODE32	MODE16	SMP	CKE
bit 15							bit 8

R/W-0	R/W-0	R/W-0	r-x	R/W-0	R/W-0	R/W-0	R/W-0
SSEN	CKP	MSTEN	—	STXISEL<1:0> <sup>(4)</sup>	SRXISEL<1:0> <sup>(4)</sup>		
bit 7							bit 0

**Legend:**

R = Readable bit      W = Writable bit      P = Programmable bit      r = Reserved bit  
 U = Unimplemented bit      -n = Bit Value at POR: ('0', '1', x = Unknown)

bit 31	<b>FRMEN:</b> Framed SPI Support bit 1 = Framed SPI support is enabled ( $\overline{SS}_x$ pin used as FSYNC input/output) 0 = Framed SPI support is disabled
bit 30	<b>FRMSYNC:</b> Frame Sync Pulse Direction Control on $\overline{SS}_x$ pin bit (Framed SPI mode only) 1 = Frame sync pulse input (Slave mode) 0 = Frame sync pulse output (Master mode)
bit 29	<b>FRMPOL:</b> Frame Sync Polarity bit (Framed SPI mode only) 1 = Frame pulse is active-high 0 = Frame pulse is active-low
bit 28	<b>MSSEN:</b> Master Mode Slave Select Enable bit <sup>(4)</sup> 1 = Slave select SPI support enabled. The $\overline{SS}$ pin is automatically driven during transmission in Master mode. Polarity is determined by the FRMPOL bit. 0 = Slave select SPI support is disabled.
bit 27	<b>FRMSYPW:</b> Frame Sync Pulse Width bit <sup>(4)</sup> 1 = Frame sync pulse is one character wide 0 = Frame sync pulse is one clock wide

**Register 23-1: SPIxCON: SPI Control Register<sup>(1,2,3)</sup> (Continued)**

bit 26-24	<b>FRMCNT&lt;2:0&gt;:</b> Frame Sync Pulse Counter bits. Controls the number of data characters transmitted per pulse. <sup>(4)</sup>  111 = Reserved; do not use 110 = Reserved; do not use 101 = Generate a frame sync pulse on every 32 data characters 100 = Generate a frame sync pulse on every 16 data characters 011 = Generate a frame sync pulse on every 8 data characters 010 = Generate a frame sync pulse on every 4 data characters 001 = Generate a frame sync pulse on every 2 data characters 000 = Generate a frame sync pulse on every data character
-----------	--

Note: This bit is only valid in FRAMED\_SYNC mode.

bit 23-18	Reserved: Write '0'; ignore read
bit 17	<b>SPIFE:</b> Frame Sync Pulse Edge Select bit (Framed SPI mode only) 1 = Frame synchronization pulse coincides with the first bit clock 0 = Frame synchronization pulse precedes the first bit clock
bit 16	<b>ENHBUF:</b> Enhanced Buffer Enable bit <sup>(4)</sup> 1 = Enhanced Buffer mode is enabled 0 = Enhanced Buffer mode is disabled

bit 15	<b>ON:</b> SPI Peripheral On bit 1 = SPI Peripheral is enabled 0 = SPI Peripheral is disabled  Note: When using the 1:1 PBCLK divisor, the user's software should not read or write the peripheral's SFRs in the SYSCLK cycle immediately following the instruction that clears the module's ON bit.
bit 14	<b>FRZ:</b> Freeze in Debug Exception Mode bit 1 = Freeze operation when CPU enters Debug Exception mode 0 = Continue operation when CPU enters Debug Exception mode  Note: FRZ is writable in Debug Exception mode only, it is forced to '0' in Normal mode.
bit 13	<b>SIDL:</b> Stop in Idle Mode bit 1 = Discontinue operation when CPU enters in Idle mode 0 = Continue operation in Idle mode
bit 12	<b>DISSDO:</b> Disable SDOx pin bit 1 = SDOx pin is not used by the module. Pin is controlled by associated PORT register 0 = SDOx pin is controlled by the module
bit 11-10	<b>MODE&lt;32,16&gt;:</b> 32/16-Bit Communication Select bits 1x = 32-bit data width 01 = 16-bit data width 00 = 8-bit data width

- Note 1:** This register has an associated Clear register (SPIxCONCLR) at an offset of 0x4 bytes. Writing a '1' to any bit position in the Clear register will clear valid bits in the associated register. Reads from the Clear register should be ignored.
- 2:** This register has an associated Set register (SPIxCONSET) at an offset of 0x8 bytes. Writing a '1' to any bit position in the Set register will set valid bits in the associated register. Reads from the Set register should be ignored.
- 3:** This register has an associated Invert register (SPIxCONINV) at an offset of 0xC bytes. Writing a '1' to any bit position in the Invert register will invert valid bits in the associated register. Reads from the Invert register should be ignored.
- 4:** These bits are not available on all devices. Refer to the specific device data sheet for availability.

**Register 23-1: SPIxCON: SPI Control Register<sup>(1,2,3)</sup> (Continued)**

bit 9	<b>SMP:</b> SPI Data Input Sample Phase bit <u>Master mode (MSTEN = 1):</u> 1 = Input data sampled at end of data output time 0 = Input data sampled at middle of data output time <u>Slave mode (MSTEN = 0):</u> SMP value is ignored when SPI is used in Slave mode. The module always uses SMP = 0.
bit 8	<b>CKE:</b> SPI Clock Edge Select bit 1 = Serial output data changes on transition from active clock state to Idle clock state (see CKP bit) 0 = Serial output data changes on transition from Idle clock state to active clock state (see CKP bit)  Note: The CKE bit is not used in the Framed SPI mode. The user should program this bit to '0' for the Framed SPI mode (FRMEN = 1).
bit 7	<b>SSEN:</b> Slave Select Enable (Slave mode) bit 1 = SS <sub>x</sub> pin used for Slave mode 0 = SS <sub>x</sub> pin not used for Slave mode, pin controlled by port function.
bit 6	<b>CKP:</b> Clock Polarity Select bit 1 = Idle state for clock is a high level; active state is a low level 0 = Idle state for clock is a low level; active state is a high level
bit 5	<b>MSTEN:</b> Master Mode Enable bit 1 = Master mode 0 = Slave mode
bit 4	Reserved: Write '0'; ignore read
bit 3-2	<b>TXISEL&lt;1:0&gt;:</b> SPI Transmit Buffer Empty Interrupt Mode bits <sup>(4)</sup> 11 = SPI_TBE_EVENT is set when the buffer is not full (has one or more empty elements) 10 = SPI_TBE_EVENT is set when the buffer is empty by one-half or more 01 = SPI_TBE_EVENT is set when the buffer is completely empty 00 = SPI_TBE_EVENT is set when the last transfer is shifted out of SPISR and transmit operations are complete
bit 1-0	<b>RTXISEL&lt;1:0&gt;:</b> SPI Receive Buffer Full Interrupt Mode bits <sup>(4)</sup> 11 = SPI_RBF_EVENT is set when the buffer is full 10 = SPI_RBF_EVENT is set when the buffer is full by one-half or more 01 = SPI_RBF_EVENT is set when the buffer is not empty 00 = SPI_RBF_EVENT is set when the last word in the receive buffer is read (i.e., buffer is empty)

#### 8.8.4. SÉLECTION DE LA FRÉQUENCE DE SCK

Voici la formule qui détermine la fréquence de SCK en fonction de la fréquence de PB\_CLOCK et du registre SPIxBRG.

$$F_{SCK} = \frac{F_{PB}}{2 \cdot (SPIxBRG + 1)}$$

Ce qui donne des divisions possibles par 2, 4, 6, 8, etc.

Avec PB\_CLOCK = 80 MHz et SPIxBRG = 0, la fréquence maximum de SCK sera de 40 MHz.

Lors de la configuration, nous avons établi :

```
PLIB_SPI_BaudRateSet(KitSpi1,
                      SYS_CLK_PeripheralFrequencyGet(CLK_BUS_PERIPHERAL_1),
                      20000000);
```

Pour obtenir 20 MHz, il faut diviser par 4 donc SPIxBRG doit valoir 1. Ce que nous vérifions en lisant la valeur de SPI1BRG.

```
PLIB_SPI_Enable(KitSpi1);
// Contrôle 1
ConfigReg = S[SPI1BRG] 0
              SPI1BRG   on
              000000001
BaudReg = SPI1BRG;
```

#### 8.8.5. INITIALISATION DU LTC2604

Voici la fonction d'initialisation complète avec le reset du LTC2604.

```
void SPI_InitLTC2604(void)
{
    //Initialisation SPI DAC
    CS_DAC = 1;
    // Impulsion reset du DAC
    DAC_CLEAR = 0;
    delay_us(500);
    DAC_CLEAR = 1;

    // LTC2604 MAX 50 MHz choix 20 MHz
    SPI_ConfigureLTC2604();
}
```

### 8.8.6. ECRITURE D'UNE VALEUR SUR LE LTC2604

Voici la fonction d'écriture sur le LTC2604. Dans le but de partager le bus SPI avec un autre composant, la fonction d'écriture reconfigure le SPI avant l'action.

```

// Envoi d'une valeur sur le DAC LTC2604
// Avec reconfiguration du SPI
// Indication du canal 0 à 3
void SPI_CfgWriteToDac(uint8_t NoCh, uint16_t DacVal)
{
    uint8_t MSB;
    uint8_t LSB;

    // Reconfiguration du SPI
    SPI_ConfigureLTC2604();

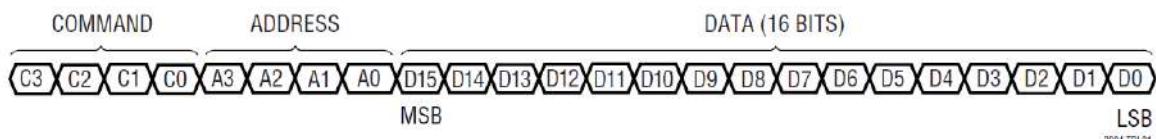
    // Sélection du canal
    // 3 -> Set and Update, 0/1/2/3 Sélection canal A/B/C/D,
    //                                F tous canaux
    NoCh = NoCh + 0x30;
    MSB = DacVal >> 8;
    LSB = DacVal;

    CS_DAC = 0;
    spi_write1(NoCh);
    spi_write1(MSB);
    spi_write1(LSB);

    // Fin de transmission
    CS_DAC = 1;
} // SPI_CfgWriteToDac

```

Ce qui correspond à l'indication du fabricant :



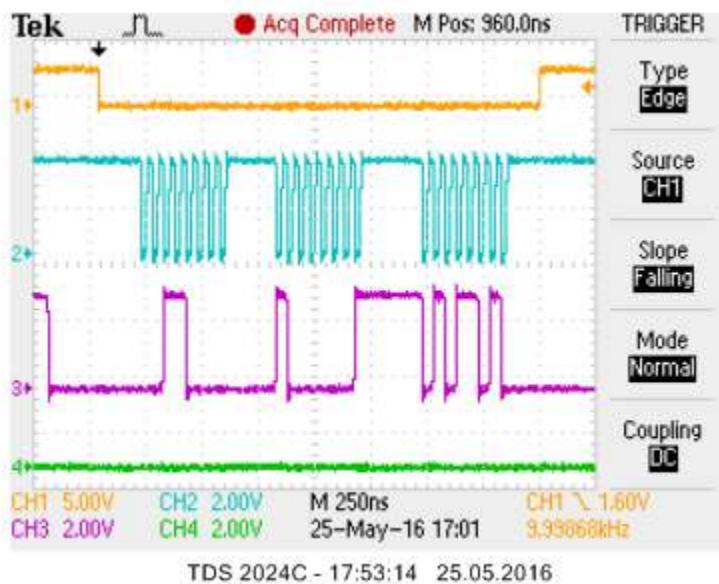
### 8.8.7. OBSERVATION DES SIGNAUX DU DAC

Canal 1 : CS\_DAC  
RD4  
broche 81

Canal 2 : SPI-SCK  
SCK1/RD10  
broche 70

Canal 3 : SPI-MOSI  
SDO1/RD0  
broche 72

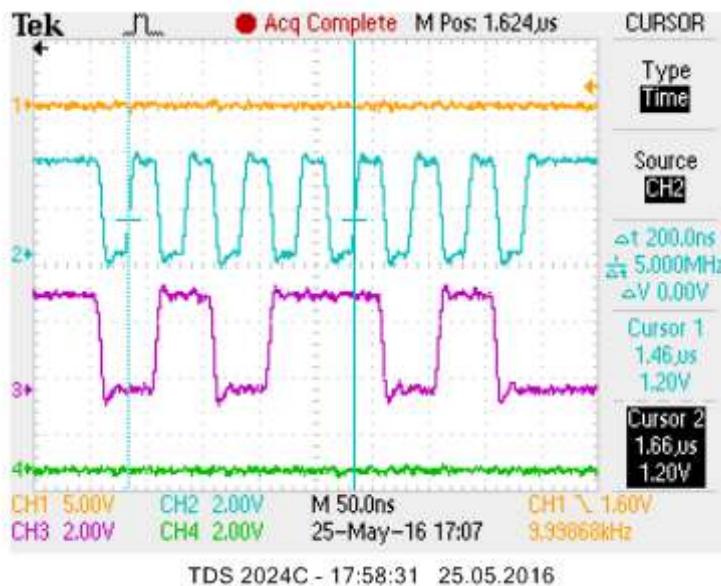
Canal 4 : sortie du  
DAC 0



On observe les 3 salves de 8 coups d'horloge. La valeur envoyée est 0x81 pour le MSB et 0x5A sur le LSB (3<sup>ème</sup> data).

#### 8.8.7.1. DÉTAIL DU 3ÈME OCTET

La valeur du 3<sup>ème</sup> octet (LSB) est imposée à 0x5A. On peut donc vérifier que le flanc montant du clock est au milieu du data. Au 1<sup>er</sup> flanc montant on a bien un 0 et au 5<sup>ème</sup> flanc un 1.



Les 4 périodes d'horloge valent 200 ns, ce qui nous donne 50 ns pour une période donc une fréquence de 20 MHz comme prévu.

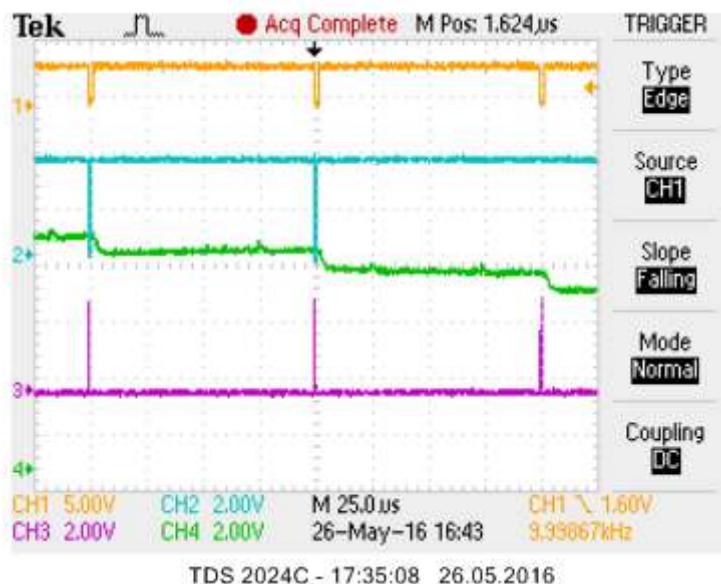
Le fait que l'horloge démarre sur un état haut ne pose pas de problème.

### 8.8.7.2. SIGNAL SUR LE DAC

Si on transmet des valeurs qui se suivent, comme par exemple :

```
SPI_CfgWriteToDac(0, DacVal);  
DacVal += 2048;
```

On peut observer les changements de valeur du signal, ce qui prouve la bonne interprétation par le DAC.



## 8.10. COMMUNICATION AVEC LE LM70

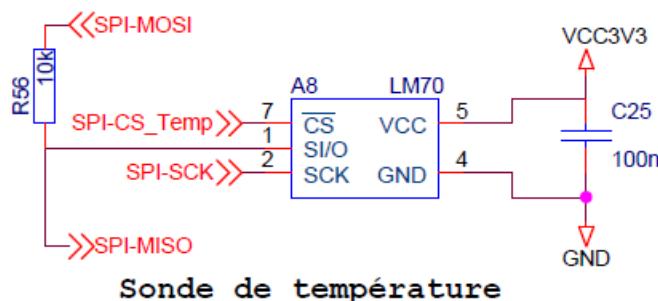
Nous allons illustrer principalement l'utilisation des fonctions de lecture du SPI par la communication avec le composant LM70 qui est un capteur de température.

### 8.10.1. CONNEXIONS ENTRE LE PIC32 ET LE LM70

Utilisation du SPI1 et d'une ligne de port standard (RD3) pour le signal  $\overline{CS}$

Broche du PIC32MX795F512L (100 pin TQFP)	Nom schéma	No broche boîtier 100
SCK1/IC3/PMCS2/PMA15/RD10   70 71	SPI-SCK	70
SDO1/OC1/INT0/RD0   72 76	SPI-MOSI	72
SPI-MISO   9 63	SPI-MISO	9
OC3/RD2   78 81	SPI-CS_Temp	78

Comme le LM70 ne possède qu'une ligne bidirectionnelle, il est nécessaire d'utiliser une résistance pour combiner les lignes SPI-MOSI et SPI-MISO SDO du PIC.

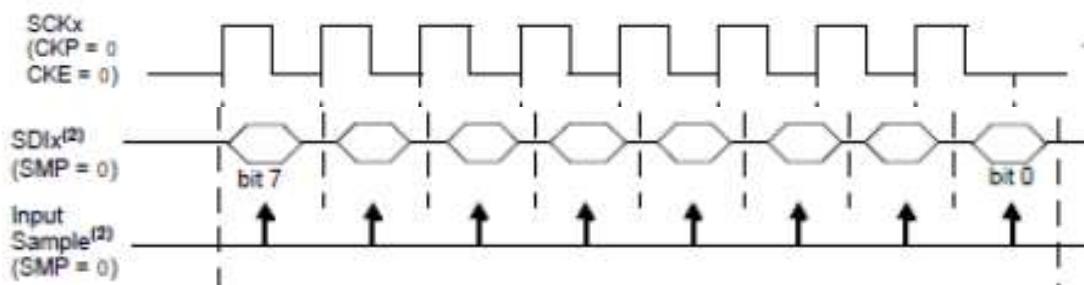


### 8.10.2. CONFIGURATION SPI NÉCESSAIRE AU LM70

Le fabricant du LM70 indique :

*The LM70 operates as a slave and is compatible with SPI or MICROWIRE bus specifications. Data is clocked out on the falling edge of the serial clock (SC), while data is clocked in on the rising edge of SC.*

Pour la lecture du LM70 par le maître, les données sont fournies par l'esclave au flanc descendant de SCK, tandis que pour l'écriture par le maître, les datas sont lues au flanc montant de l'horloge. Ce qui correspond à CKP = 0 et CKE = 0.



### 8.10.2.1. FONCTION DE CONFIGURATION DU SPI POUR LE LM70

Voici le contenu de la fonction SPI\_ConfigureLM70 qui est fournie dans les fichiers Mc32GestSpiLM70.h et Mc32GestSpiLM70.c.

```
void SPI_ConfigureLM70(void)
{
    PLIB_SPI_Disable(KitSpi1);

    PLIB_SPI_BufferClear(KitSpi1);
    PLIB_SPI_StopInIdleDisable(KitSpi1);
    PLIB_SPI_PinEnable(KitSpi1, SPI_PIN_DATA_OUT);
    PLIB_SPI_CommunicationWidthSelect(KitSpi1,
                                       SPI_COMMUNICATION_WIDTH_8BITS);
    // LM70 MAX 6.25 MHz choix 5 MHz
    PLIB_SPI_BaudRateSet(KitSpi1,
                         SYS_CLK_PeripheralFrequencyGet(CLK_BUS_PERIPHERAL_1),
                         5000000);

    // Config polarité traitement des signaux SPI
    // pour input à confirmer
    // Polarité clock OK
    // Phase output à confirmer
    PLIB_SPI_InputSamplePhaseSelect(KitSpi1,
                                    SPI_INPUT_SAMPLING_PHASE_IN_MIDDLE);
    PLIB_SPI_ClockPolaritySelect(KitSpi1,
                                SPI_CLOCK_POLARITY_IDLE_LOW);
    PLIB_SPI_OutputDataPhaseSelect(KitSpi1,
                                SPI_OUTPUT_DATA_PHASE_ON_IDLE_TO_ACTIVE_CLOCK);
    PLIB_SPI_MasterEnable(KitSpi1);
    PLIB_SPI_FramedCommunicationDisable(KitSpi1);
    PLIB_SPI_FIFOEnable(KitSpi1); // Enhanced buffer mode

    PLIB_SPI_Enable(KitSpi1);

    // Contrôle de la configuration
    ConfigReg = SPI1CON;
    BaudReg = SPI1BRG;
}
```

### 8.10.2.2. VÉRIFICATION DE LA CONFIGURATION AVEC SPI1CON

[SPI1CON]	31	30	29	28	27	24	17
FRMEN	FRMSYNC	FRMPOL	MSSEN	FRMSYPW	FRMCNT	-	SPIFE
0	0	0	0	0	000	-	0
16 15	13	12	11	10	9		
ENHBUF	ON	-	SIDL	DISSDO	MODE32	MODE16	SMP
1	1	-	0	0	0	0	0
8 7 6 5			2		0		
CKE	SSEN	CKP	MSTEN	-	STXISEL	SRXISEL	
0	0	0	1	-	00	00	

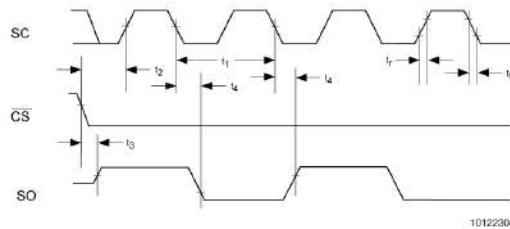
On a bien confirmation de CKE = 0 et CKP = 0.

### Serial Bus Digital Switching Characteristics

Unless otherwise noted, these specifications apply for  $V^+ = 2.65V$  to  $3.6V$  for the LM70-3 and  $V^+ = 4.5V$  to  $5.5V$  for the LM70-5,  $C_L$  (load capacitance) on output lines =  $100\text{ pF}$  unless otherwise specified. **Boldface limits apply for  $T_A = T_J = T_{MIN}$  to  $T_{MAX}$ ; all other limits  $T_A = T_J = +25^\circ\text{C}$ , unless otherwise noted.**

Symbol	Parameter	Conditions	Typical (Note 7)	Limits (Note 8)	Units (Limit)
$t_1$	SC (Clock) Period			<b>0.16</b> DC	$\mu\text{s}$ (min) (max)
$t_2$	CS Low to SC (Clock) High Set-Up Time			100	ns (min)
$t_3$	$\overline{\text{CS}}$ Low to Data Out (SO) Delay			70	ns (max)
$t_4$	SC (Clock) Low to Data Out (SO) Delay			70	ns (max)
$t_5$	$\overline{\text{CS}}$ High to Data Out (SO) TRI-STATE			200	ns (min)
$t_6$	SC (Clock) High to Data In (SI) Hold Time			60	ns (min)
$t_7$	Data In (SI) Set-Up Time to SC (Clock) High			30	ns (min)

### Timing Diagrams



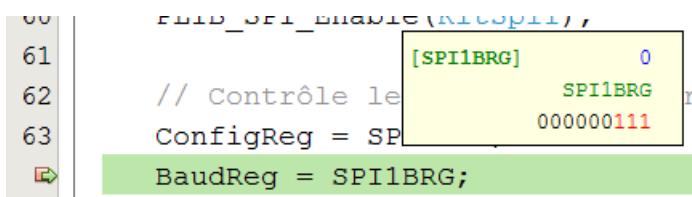
10122304

Le minimum de 0,16 us pour le serial clock correspond à un max de 6.25 MHz. Avec la division par 16 on obtient si la fréquence d'oscillateur est de 80 MHz,  $80/16 = 5\text{ MHz}$ .

Lors de la configuration nous avons établi :

```
PLIB_SPI_BaudRateSet(KitSpi1,
    SYS_CLK_PeripheralFrequencyGet(CLK_BUS_PERIPHERAL_1),
    5000000);
```

Pour obtenir 5 MHz, il faut diviser par 16 donc SPIxBRG doit valoir 7, car  $2(7+1) = 16$ . Ce que nous vérifions en lisant la valeur de SPI1BRG.



...ce qui correspond bien à 7.

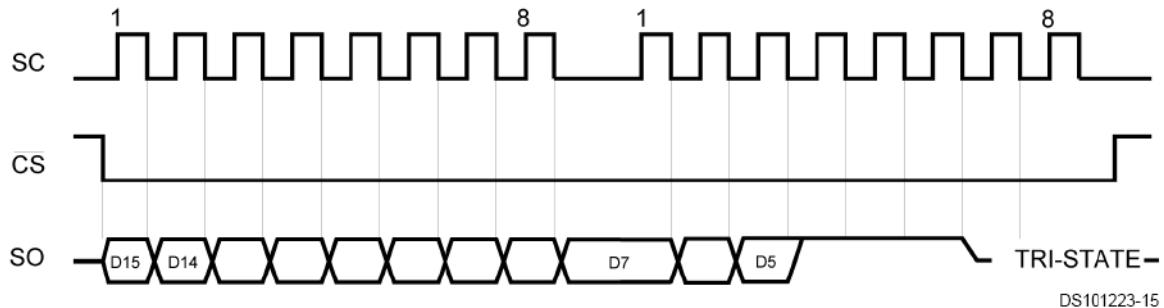
### 8.10.3. INITIALISATION DU LM70

La fonction **SPI\_InitLM70** effectue la configuration du SPI pour le LM70 et effectue une séquence permettant de configurer les registres du LM70.

```
void SPI_InitLM70(void) {
    SPI_ConfigureLM70();
    // action de configuration
    CS_LM70 = 0;
    spi_read1(0xFF);
    spi_read1(0xFF);
    spi_read1(0); // pour écrire 0
    spi_read1(0); // pour écrire 0
    //Fin de transmission
    CS_LM70 = 1;
} // SPI_InitLM70
```

### 8.10.4. LECTURE DU LM70

Pour réaliser une lecture, il faut respecter la séquence ci-dessous :



b) Reading Continuous Conversion - Two Eight-Bit Frames

On remarque que le signal  $\overline{CS}$  doit être activé au début de l'action et désactivé à la fin de l'action. La lecture s'effectue en deux trains de 8 bits, ce qui correspond bien à l'usage du MSSP.

### 8.10.5. LA FONCTION SPI\_READRAWMPLM70

La fonction SPI\_ReadRawTempLM70 effectue la lecture en appelant 2 fois la fonction **spi\_read1**. On obtient d'abord le poids fort et ensuite le poids faible. La valeurs des 2 octets envoyés n'a pas d'importance.

```
// Lecture du registre de température du LM70
// Version sans reconfiguration
int16_t SPI_ReadRawTempLM70(void)
{
    //Déclaration des variables
    uint8_t MSB;
    uint8_t LSB;
    int16_t RawTemp;

    CS_LM70 = 0;
    MSB = spi_read1(0xFF);
    LSB = spi_read1(0xFF);
    //Fin de transmission
    CS_LM70 = 1;

    RawTemp = MSB;
    RawTemp = RawTemp << 8;
    RawTemp = RawTemp | LSB;
    return RawTemp;
} // SPI_ReadRawTempLM70
```

Pour obtenir la température il faut combiner le msb et le lsb et manipuler la valeur en tenant compte des détails du registre de température :

#### 1.5.2 TEMPERATURE REGISTER

(Read Only):

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
MSB	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	LSB	1	1	1	X	X

D0-D1: Undefined. TRI-STATE will be output on SI/O.

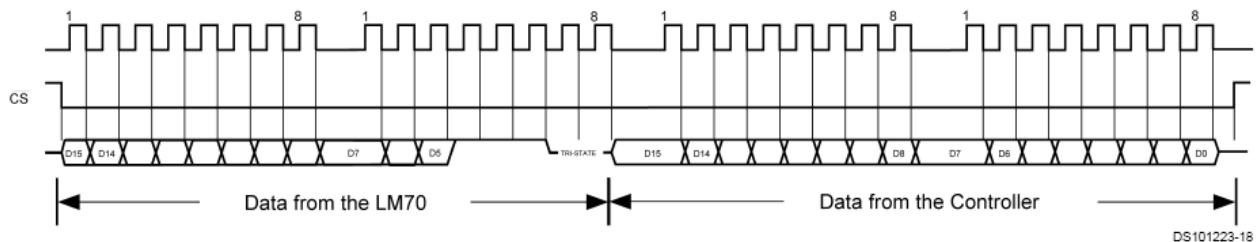
D2-D4: Always set high.

D5-D15: Temperature Data. One LSB = 0.25°C. Two's complement format.

complement format.

### 8.10.6. ECRITURE VERS LE LM70

Il y a écriture uniquement pour la configuration du mode de Shutdown. Il y a 2 cycles de lecture avant les 2 cycles d'écriture. C'est ce que nous déduisons du diagramme ci-dessous :



c) Writing Shutdown Control

On remarque que le signal  $\overline{CS}$  doit être activé au début de l'action et désactivé à la fin.

#### 1.5.1 CONFIGURATION REGISTER

(Selects shutdown or continuous conversion modes):

(Write Only):

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	Shutdown

D0-D15 set to XX FF hex enables shutdown mode.

D0-D15 set to XX 00 hex enables continuous conversion mode.

Note: setting D0-D15 to any other values may place the LM70 into a manufacturer's test mode, upon which the LM70

will stop responding as described. These test modes are to be used for National Semiconductor production testing only. See Section 1.2 Serial Bus Interface for a complete discussion.

L'écriture s'adresse uniquement au registre de configuration. Il faut écrire XX 00 pour obtenir le mode continu. Dans l'exemple d'écriture on utilise 00 00.

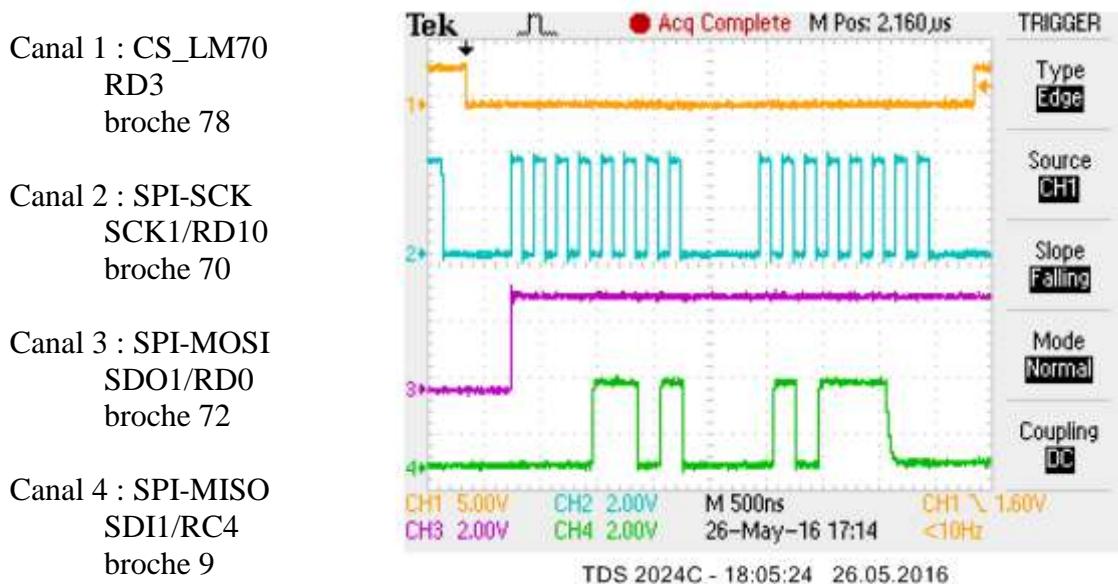
#### 8.10.6.1. EXEMPLE D'ÉCRITURE

Dans l'exemple ci-dessous, il y a écriture d'une valeur 16 bits valant 0. Pour éviter des problèmes avec le buffer de réception, il faut utiliser la fonction de lecture aussi pour écrire les 0. D'où :

```
// action de configuration
CS_LM70 = 0;
spi_read1(0xFF);
spi_read1(0xFF);
spi_read1(0); // pour écrire 0
spi_read1(0); // pour écrire 0
//Fin de transmission
CS_LM70 = 1;
```

### 8.10.7. OBSERVATION DES SIGNAUX DU LM70

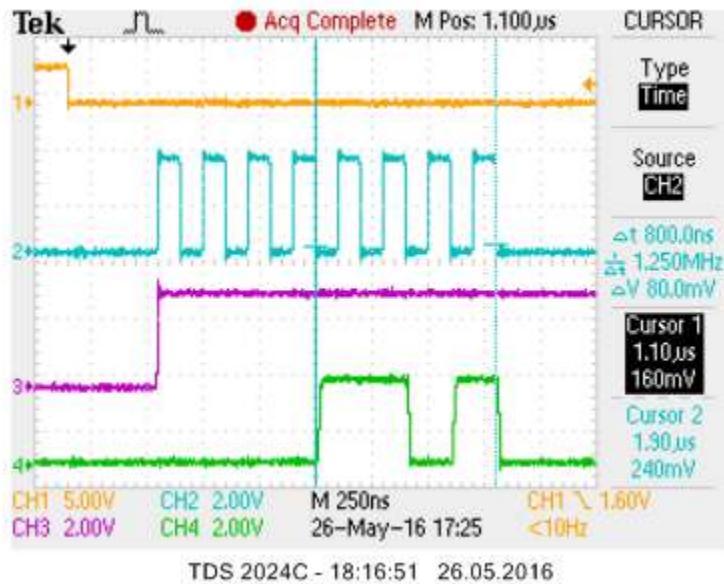
Voici la vue d'ensemble lors de la lecture du registre de température :



On peut observer que le MOSI (canal 3) est à l'état haut durant l'action. Le LM70 fournit une valeur dès réception des coups d'horloge.

### 8.10.7.1. VUE DE DÉTAIL

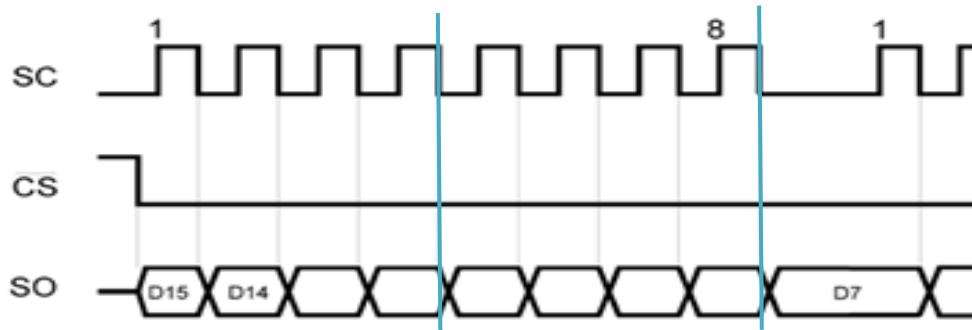
Voici la vue de détail, observation du traitement du 1<sup>er</sup> octet (MSB, le plus stable).



La fréquence d'horloge est bien de 5 MHz car 4 périodes donnent 800 ns et pour 5 MHz la période est de 200 ns.

La valeur de l'octet de poids fort transmis est 0x0D, soit 0000'1101, ce que le master peut lire au flancs montants de l'horloge. Le LM70 fournit un nouveau bit lors de chaque flanc descendant de l'horloge (sauf le tout premier bit, qui est fourni dès le /CS bas). Au 1<sup>er</sup> flanc descendant, on a donc le 2<sup>ème</sup> bit qui est mis en sortie du LM70. Lors du dernier flanc descendant de l'horloge, le data passe déjà à l'octet suivant.

La figure datasheet nous confirme tout cela :



## 8.11. UTILISATION COMBINÉE DES 2 SLAVES SPI

Pour permettre d'utiliser sur le même bus SPI du LM70 et du LTC2604, il est nécessaire de reconfigurer le SPI avant chaque transaction. Cela permet donc d'utiliser des fréquences d'horloge différentes et des modes d'horloge différents.

Au niveau de l'initialisation on exécutera une fois la fonction d'initialisation du LTC2604 pour s'assurer de son Reset.

### 8.11.1. FONCTION LECTURE LM70 AVEC RECONFIGURATION

Lecture du registre de température du LM70, version avec reconfiguration pour partage du bus SPI avec un autre composant.

```
// Lecture du registre de température du LM70
// Version avec reconfiguration
int16_t SPI_CfgReadRawTempLM70(void)
{
    uint8_t MSB;
    uint8_t LSB;
    int16_t RawTemp;

    SPI_ConfigureLM70();

    CS_LM70 = 0;
    MSB = spi_read1(0xFF);
    LSB = spi_read1(0xFF);
    //Fin de transmission
    CS_LM70 = 1;

    RawTemp = MSB;
    RawTemp = RawTemp << 8;
    RawTemp = RawTemp | LSB;
    return RawTemp;
} // SPI_CfgReadRawTempLM70
```

### 8.11.2. FONCTION ÉCRITURE LTC2604 AVEC RECONFIGURATION

Cette fonction appelle la fonction de reconfiguration avant d'effectuer la transaction.

```
void SPI_CfgWriteToDac(uint8_t NoCh, uint16_t DacVal)
{
    //Déclaration des variables
    uint8_t MSB;
    uint8_t LSB;

    // Reconfiguration du SPI
    SPI_ConfigureLTC2604();

    //Sélection du canal
    //3 -> Set and Update, 0/1/2/3 Sélection canal A/B/C/D,
    //                                              F tous canaux
    NoCh = NoCh + 0x30;

    // Selon canal
    MSB = DacVal >> 8;
    LSB = DacVal;

    CS_DAC = 0;
    spi_write1(NoCh);
    spi_write1(MSB);
    spi_write1(LSB);
    // Fin de transmission
    CS_DAC = 1;

} // SPI_CfgWriteToDac
```

### 8.11.3. UTILISATION ET OBTENTION DES RÉSULTATS

Dans cet exemple, on transmet les valeurs au DAC dans une routine d'interruption. Pour garantir que la séquence de lecture du LM70 ne soit pas interrompue par le dialogue du DAC, on la place également dans l'interruption du DAC en utilisant un compteur.

#### 8.11.3.1. CONTENU DE LA RÉPONSE À L'INTERRUPTION DU TIMER1

Voici le contenu de la réponse à l'interruption du Timer1.

```
// Cycle 100 us
// Envois échantillon sur DAC
// Lecture température LM70 et activation
// de l'application tous les 5000 cycles (500 ms)

void __ISR(__TIMER_1_VECTOR, ipl6AUTO)
    _IntHandlerDrvTmrInstance0(void)
{
    static int count = 0;
    static uint16_t DacVal = 0;
    static int16_t RawTemp;
    static float TempLm70;
```

```

PLIB_INT_SourceFlagClear(INT_ID_0, INT_SOURCE_TIMER_1);
count++;
// DacVal = 0x815A; // pour observation signaux
SPI_CfgWriteToDac(0, DacVal);
DacVal += 2048;
if (count >= 5000 ) {
    count = 0;
    delay_us(5); // pour séparation des signaux
    RawTemp = SPI_CfgReadRawTempLM70();
    LM70_ConvRawToDeg( RawTemp, &TempLm70);
    APP_UpdateTemp (RawTemp, TempLm70);
    APP_UpdateState(APP_STATE_SERVICE_TASKS);
}
} // end ISR

```

### 8.11.3.2. AFFICHAGE TEMPÉRATURE DANS L'APPLICATION

L'affichage est réalisé dans l'application. Les fonctions de mise à jour appelées depuis l'interruption permettent un accès propre aux variables de l'application.

```

APP_DATA appData;
int16_t APP_RawTemp = 225;
float APP_TempLm70 = 21.7;

case APP_STATE_SERVICE_TASKS:
    BSP_LEDToggle(BSP_LED_2);
    // Affichage temperature du LM70
    lcd_gotoxy(1,3);
    printf_lcd("RawTemp = %08X", APP_RawTemp);
    lcd_gotoxy(1,4);
    printf_lcd("Temp = %6.1f", APP_TempLm70);
    appData.state = APP_STATE_WAIT;
break;

```

### 8.11.3.3. PROBLÈME AVEC LES VARIABLES FLOAT DANS L'INTERRUPTION

Dans l'interruption, nous avons déclaré les 2 variables pour le LM70 de la manière suivante :

```

static int16_t RawTemp;
static float TempLm70;

```

En écrivant :

```
TempLm70 = LM70_ConvRawToDeg( RawTemp);
```

Le résultat dans TempLm70 était totalement incohérent.

En modifiant en :

```
LM70_ConvRawToDeg( RawTemp, &TempLm70);
```

Le résultat est correct.

(?) La manipulation d'une variable float implique un appel à une fonction système, il semble que cela soit la cause du problème, sans pour autant l'expliquer.

En utilisant le passage par référence, ce problème disparaît.

#### 8.11.4. REMARQUE SUR LES RECONFIGURATIONS

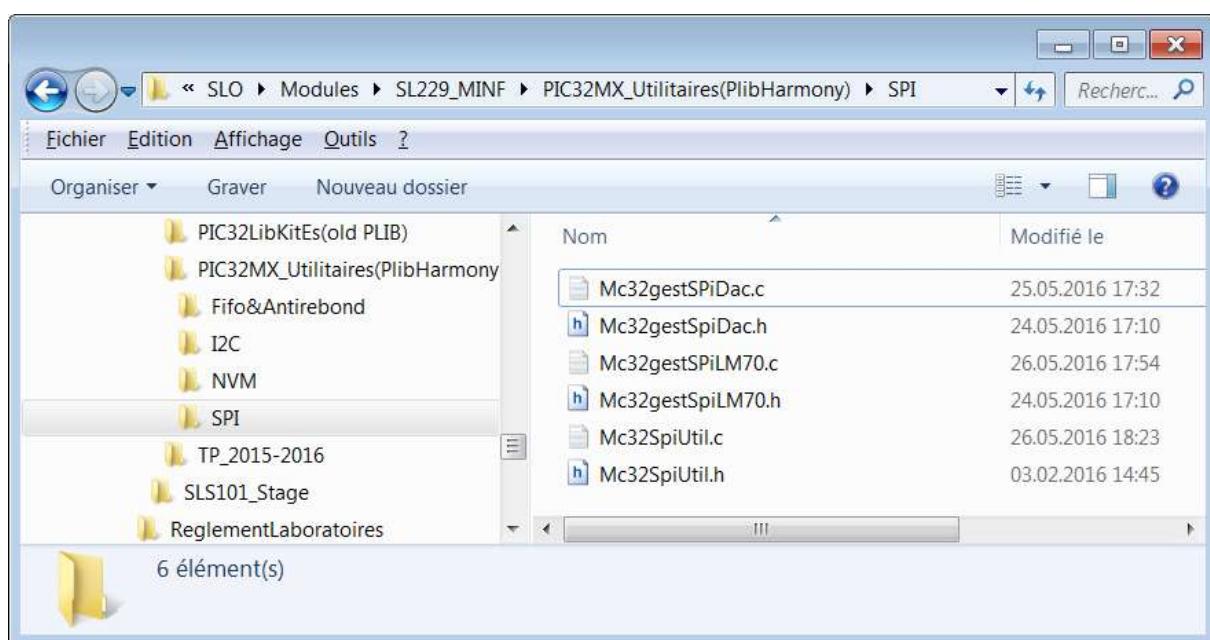
La reconfiguration avant chaque communication avec un des périphériques SPI permet de passer d'une horloge à 20 MHz pour le LTC2604 à une horloge à 5 MHz pour le LM70. Les modes d'horloge sont aussi adaptés.

La reconfiguration présente l'avantage d'effacer les erreurs, ce qui permet d'assurer une lecture propre du LM70, alors que l'on a utilisé la fonction d'écriture pour le LTC2604.

### 8.12. FICHIERS À DISPOSITION

Les librairies permettant l'utilisation des composants SPI sont fournies dans le répertoire suivant :

...\\Maitres-Eleves\\SLO\\Modules\\SL229\_MINF\\PIC32MX\_Utilitaires(PlibHarmony)\\SPI



### 8.13. CONCLUSION

Ce document devrait permettre, en s'inspirant des principes utilisés, de s'adapter à la gestion par le bus SPI d'autres composants que ceux présentés.

Il s'agira à chaque fois d'adapter la configuration SPI (notamment la fréquence du clock, sa polarité et le nombre de tranches de 8 bits à lire ou à écrire). Une étude détaillée des datasheets est indispensable.

## 8.14. HISTORIQUE DES VERSIONS

### 8.14.1. VERSION 1.0 MAI 2014

Transformation du chapitre 13 (théorie PIC18) pour obtenir la 1<sup>ère</sup> version de ce chapitre.

### 8.14.2. VERSION 1.1 MAI 2014

Remplacement des extraits de schéma PIC18 pour le LM70. Complément info sur le SPIICON. Ajout observation des signaux du LM70.

### 8.14.3. VERSION 1.5 MARS 2015

Redevient un chapitre de théorie. Adaptation à la PLIB\_SPI de Harmony V 1.00 et description d'une partie des fonctions.

### 8.14.4. VERSION 1.7 MAI 2016

Version 1.7 pour compatibilité avec l'ensemble des modules. Adaptation à la PLIB\_SPI de Harmony V 1.06 et ajout de l'étude du driver SPI fourni par le MHC. Correction de la fonction spi\_read et contrôle du bon fonctionnement.

### 8.14.5. VERSION 1.8 MARS 2017

Relecture générale par SCA.

### 8.14.6. VERSION 1.9 FÉVRIER 2018

Ajouts documents de référence. Corrections mineures.

### 8.14.7. VERSION 1.91 MARS 2019

Correction mineure fonction spi\_write1.

**MINF**  
**Programmation des PIC32MX**

# **Chapitre 9**

## **Gestion du bus I2C**



## **Théorie PIC32MX**

**Christian HUBER (CHR)**  
**Serge CASTOLDI (SCA)**  
**Version 1.91 février 2022**



## CONTENU DU CHAPITRE 9

<b>9. Gestion du bus I2C avec les PIC32MX</b>	<b>9-1</b>
<b>9.1. Bus I2C</b>	<b>9-2</b>
9.1.1. Généralités	9-2
9.1.2. Les lignes du bus I2C	9-3
9.1.2.1. Dimensionnement des résistances de pull-up	9-4
9.1.3. Principe de la communication I2C	9-5
9.1.3.1. La condition de départ	9-5
9.1.3.2. L'octet d'en-tête	9-6
9.1.3.3. Principe de la transmission des données	9-6
9.1.3.4. Notion d'acquittement	9-6
9.1.3.5. La condition d'arrêt	9-7
9.1.3.6. Détail de la transmission d'un bit	9-7
9.1.4. Notion de maître et d'esclaves	9-7
9.1.5. Situation multi-master	9-8
<b>9.2. Réalisation du bus I2C avec le PIC32MX795F512L</b>	<b>9-9</b>
9.2.1. Composant I2C du kit PIC32MX795F512L	9-9
9.2.1.1. Sonde de température LM92	9-9
9.2.1.2. Convertisseur 1-wire DS2482S-100	9-9
9.2.1.3. RTC, MAC adresse et EEPROM	9-10
<b>9.3. Driver I2C fourni par Harmony</b>	<b>9-11</b>
9.3.1. Config du driver I2C	9-11
9.3.2. Fonction d'initialisation obtenue	9-11
<b>9.4. Fonctions I2C souhaitées</b>	<b>9-12</b>
9.4.1. Les fonctions du compilateur CCS	9-12
9.4.2. Configuration de l'I2C	9-12
9.4.3. Contenu du fichier Mc32_I2cUtilCCS.h	9-12
9.4.4. Réalisation des fonctions	9-13
9.4.4.1. Vue d'ensemble des fonctions de la PLIB_I2C	9-13
9.4.4.2. #Include nécessaire et définitions	9-15
9.4.4.3. La fonction i2c_init	9-15
9.4.4.4. La fonction i2c_start	9-16
9.4.4.1. La fonction i2c_reStart	9-16
9.4.4.2. La fonction i2c_write	9-17
9.4.4.3. La fonction i2c_read	9-18
9.4.4.4. La fonction i2c_stop	9-19
<b>9.5. Communication avec le composant LM92</b>	<b>9-20</b>
9.5.1. Connexion entre le PIC32MX et le LM92	9-20
9.5.2. Lecture registre température du LM92	9-20
9.5.2.1. Réalisation de la lecture	9-21
9.5.2.2. Observation de la séquence	9-21
9.5.2.3. Contrôle de la période de SCL en fast	9-22
9.5.2.4. Contrôle de la période de SCL en slow	9-22
9.5.3. Set pointer et lecture température du LM92	9-23
9.5.3.1. Réalisation de la lecture avec config	9-23

9.5.3.2. Observation de la séquence	9-24
<b>9.5.4. Configuration du I2C pour le LM92</b>	<b>9-25</b>
9.5.4.1. La fonction I2C_WriteConfigLM92	9-25
9.5.4.2. Définitions de l'adresse du LM92	9-25
9.5.4.3. Définitions du pointeur de température	9-26
9.5.4.4. Organisation du registre de température	9-26
9.5.4.5. La fonction ConvRawToDeg	9-26
<b>9.6. Mise en œuvre du DS2482-100</b>	<b>9-27</b>
9.6.1. Vue d'ensemble de principe	9-27
9.6.2. Fonctionnement du DS2482-100	9-27
9.6.2.1. Registre de statut du DS2482-100	9-27
9.6.2.2. Command, Device Reset	9-28
9.6.2.3. Command, Set Read Pointer	9-29
9.6.2.4. Command, Write Configuration	9-29
9.6.2.5. Command, 1-Wire Reset	9-30
9.6.2.6. Signal issu de la command 1-Wire Reset	9-30
9.6.2.7. Command, 1-Wire Single Bit	9-31
9.6.2.8. Command, 1-Wire Write Byte	9-32
9.6.2.9. Command, 1-Wire Read Byte	9-32
9.6.2.10. Command, 1-Wire Triplet	9-33
9.6.3. Utilisation du DS2482 en relation avec DS18B20	9-34
9.6.3.1. Principe de la communication avec le DS18B20	9-34
9.6.3.2. Commandes du DS18B20	9-35
9.6.3.3. Ds18B20, Convert	9-35
9.6.3.4. Ds18B20, Read ScratchPad	9-35
9.6.3.5. Ds18B20, Résumé des commandes	9-35
9.6.3.6. Exemple de dialogue	9-36
9.6.3.7. Séquence commande ROM	9-37
9.6.3.8. Séquence traitement des commandes	9-38
9.6.4. Communication avec le composant DS18B20	9-39
9.6.4.1. Définitions pour le DS2482	9-39
9.6.4.2. Définitions 1-Wire et DS18B20	9-41
9.6.4.3. Structures et variables	9-42
9.6.4.4. Les fonctions de communications 1-Wire	9-43
9.6.4.5. La fonction de lecture de la température	9-45
9.6.4.6. Observation de la transaction	9-48
9.6.4.7. Détails du début la transaction	9-48
9.6.4.8. Détails de la fin de la transaction	9-49
9.6.4.9. Contenu du fichier app.c de l'application utilisée	9-49
<b>9.7. Fichiers à disposition</b>	<b>9-52</b>
<b>9.8. Conclusion</b>	<b>9-52</b>
<b>9.9. Historique des versions</b>	<b>9-53</b>
9.9.1. Version 1.5 avril 2015	9-53
9.9.2. Version 1.7 avril 2016	9-53
9.9.3. Version 1.7.1 avril 2016	9-53
9.9.4. Version 1.8 avril 2017	9-53
9.9.5. Version 1.9 février 2018	9-53
9.9.6. Version 1.91 février 2022	9-53

## 9. GESTION DU BUS I2C AVEC LES PIC32MX

Ce chapitre traite du bus I2C et de sa gestion avec le microcontrôleur PIC32MX795F512L en utilisant le compilateur XC32 et le framework Harmony. Il présente la mise en œuvre des composants LM92 et DS2482-100. Ce dernier étant un convertisseur I2C-OneWire, ce chapitre contiendra une introduction à ce bus.

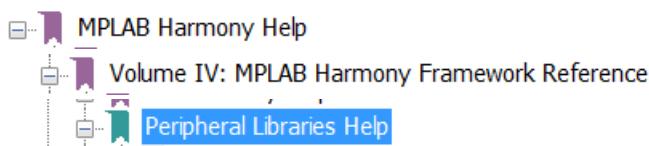
La librairie de gestion du bus I2C proposée dans ce document reprend le principe de décomposition d'une transaction I2C en actions de base, qui sont : start, restart, read, write et stop. Ce principe est issu de l'utilisation anciennement d'un autre compilateur (CCS), et a l'avantage de simplifier la gestion du bus I2C. Ainsi, il suffira au programmeur de décomposer la transaction I2C voulue en une série d'actions de base et d'appeler les fonctions adéquates dans l'ordre.

¶ Un désavantage de cette approche est le fait que ces fonctions sont bloquantes (elles incluent des boucles d'attente), ce qui est peu compatible avec un traitement cyclique rapide.

Le chapitre T.P. correspondant traite lui des fonctions I2C réalisées sous forme de machine d'état. Ce fonctionnement est donc non bloquant.

Les documents de référence sont :

- La documentation "PIC32 Family Reference Manual" :  
Section 24 : Inter-Integrated Circuit
- Pour les détails spécifiques au PIC32MX795F512L, il faut se référer au document "PIC32MX5XX/6XX/7XX Family Data Sheet" :  
Section 19 : Inter-Integrated Circuit (I2C)
- La documentation d'Harmony, qui se trouve dans <Répertoire Harmony>\v<n>\doc :  
Section MPLAB Harmony Framework Reference > Peripheral Libraries Help,  
sous-section I2C Peripheral Library



Ce document a été établi sur la base de Harmony v1.06.

## 9.1. BUS I2C

Avant d'étudier comment gérer le bus I2C avec les microcontrôleurs PIC32MX, il est nécessaire de connaître le bus I2C.

Le bus I2C, dont le sigle signifie *Inter Integrated Circuit*, ce qui donne IIC et par contraction I2C, a été proposé initialement par Philips mais est adopté de nos jours par de très nombreux fabricants. C'est un bus de communication de type série présent sur divers circuits d'interface spécialisés (convertisseurs A/D ou D/A, horloges temps réel, etc.) ainsi que sur de nombreuses mémoires EEPROM à accès série.

### 9.1.1. GENERALITES

Le bus I2C n'utilise que deux lignes de signal (et la masse correspondante bien sûr) et permet d'échanger des informations sous forme série. Ses points forts sont les suivants :

- C'est un bus série bifilaire utilisant une ligne de données appelée SDA (Serial DAta) et une ligne d'horloge appelée SCL (Serial CLock).
- Les données peuvent être échangées dans les deux sens. Etant donné qu'il n'y a que 2 lignes (data et clock), le bus reste half-duplex.
- Le bus est multi-maître.
- Dans la variante de base, chaque esclave dispose d'une adresse codée sur 7 bits. On peut donc en connecter simultanément 128 sur le même bus, avec des adresses différentes (sous réserve de ne pas le surcharger électriquement bien sûr). Une possibilité d'adressage 10 bits, cohabitant avec l'adressage 7 bits, existe.
- Une quittance (acknowledge) est générée pour chaque octet de donnée transféré.
- Selon les périphériques connectés, le bus peut travailler à une vitesse maximum de (état de la norme I2C version 3, 2007) 100 kb/s (standard mode), 400 kb/s (fast mode), 1 Mb/s (fast mode plus) ou encore 3,4 Mb/s (high speed mode). Un procédé automatique permet de ralentir l'équipement le plus rapide pour s'adapter à la vitesse de l'élément le plus lent lors d'un transfert.
- Le nombre maximum d'esclaves n'est limité que par la charge capacitive maximale du bus qui peut être de 400 pF. Ce nombre ne dépend donc que de la technologie des circuits et du mode de câblage employés.
- Les niveaux électriques permettent l'utilisation de circuits en technologies CMOS, NMOS ou TTL.

### 9.1.2. LES LIGNES DU BUS I2C

La Figure 9-1 montre le principe adopté au niveau des étages d'entrée/sortie des circuits d'interface au bus I2C. La partie entrée est représentée par un simple tampon, tandis que la partie sortie fait appel à une configuration à drain ouvert (l'équivalent en MOS du classique collecteur ouvert bipolaire). Ceci permet de réaliser des ET câblés par simple connexion des sorties SDA et SCL de tous les circuits.

Aucune charge n'étant prévue dans ces derniers, une résistance de rappel à une tension positive doit être mise en place. Le niveau électrique n'est pas précisé pour l'instant car il dépend de cette tension. Nous parlerons donc de niveaux logiques hauts ou "1" ou encore de niveaux logiques bas ou "0" étant entendu que l'on travaille en logique positive c'est-à-dire qu'un niveau haut correspond à une tension plus élevée qu'un niveau bas.

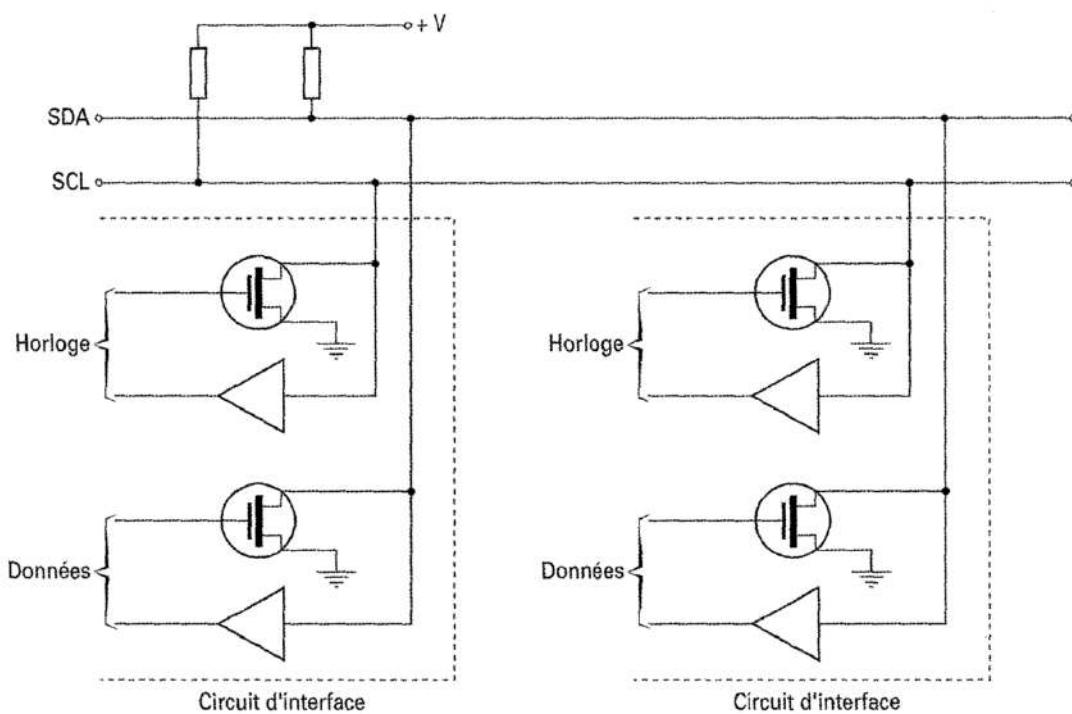


Figure 9-1

Compte tenu de ce mode de connexion en ET câblé, lorsque aucun périphérique I2C n'émet sur le bus, les lignes SDA et SCL sont au niveau haut qui est leur état de repos.

### 9.1.2.1. DIMENSIONNEMENT DES RESISTANCES DE PULL-UP

La Figure 9-2 montre la possibilité d'introduire des résistances série  $R_s$  pour protéger les composants des pics de tension.

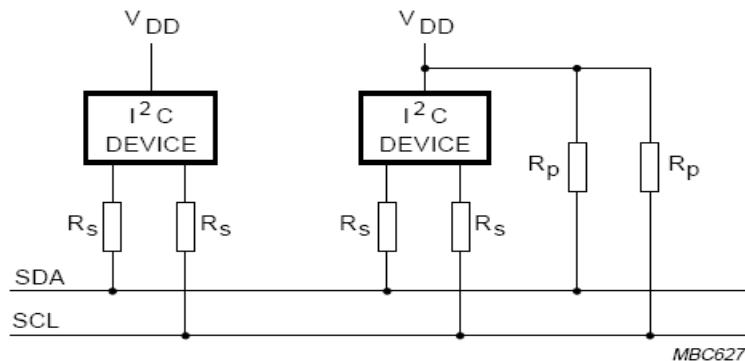
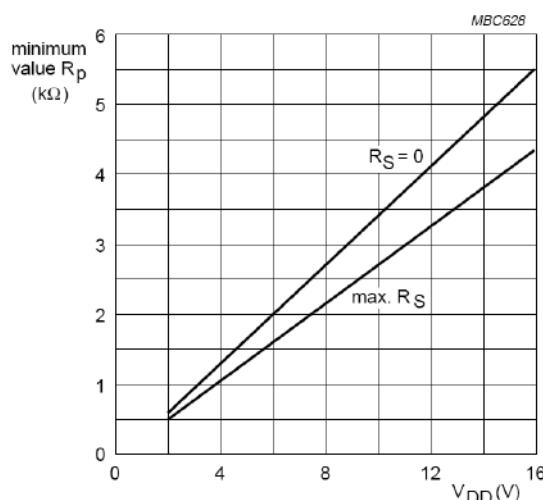
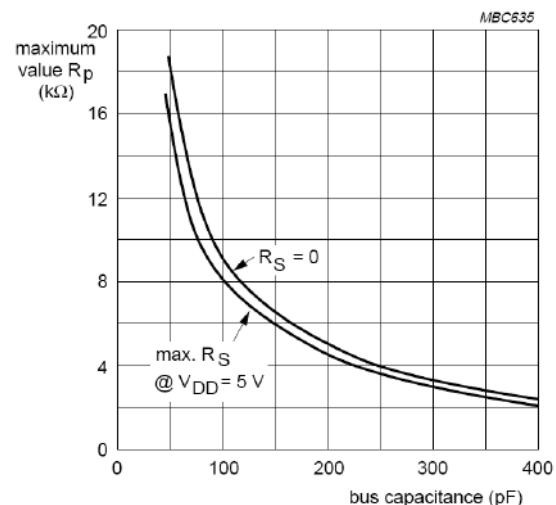


Figure 9-2

Les diagrammes ci-dessous permettent de déterminer la valeur de  $R_p$  en fonction de la tension d'alimentation et de la capacité du bus.



Valeur minimum de  $R_p$  en fonction de la tension d'alimentation et de  $R_s$ .



Valeur maximum de  $R_p$  en fonction de la capacité du bus de  $R_s$

Remarque : au niveau du kit PIC32MX, la valeur de résistance utilisée est de  $2.2\text{ k}\Omega$ , ce qui permet une capacité de bus maximum d'environ  $400\text{ pF}$ . La tension utilisée est de  $3.3\text{ V}$ , donc il serait possible de mettre des résistances de plus faible valeur (min  $1\text{ k}\Omega$ ).

### 9.1.3. PRINCIPE DE LA COMMUNICATION I2C

Pour illustrer le principe de la communication I2C, supposons que le maître veuille transmettre 2 octets de données à un esclave. La séquence de communication sera en principe la suivante :

Start	Adresse	W	ACK	Data 1	ACK	Data 2	ACK	Stop
-------	---------	---	-----	--------	-----	--------	-----	------

La transmission commence par la condition de départ (Start), il s'agit d'un changement d'état des deux lignes SDA et SCL (activation).

Au début de toute communication figure l'adresse du composant destinataire. L'adresse est sur 7 bits, le 8<sup>ème</sup> bit servant à indiquer s'il s'agit d'une lecture ou d'une écriture. Dans notre exemple il s'agit d'une écriture. Cet octet est appelé octet d'en-tête.

La transmission des données se réalise par tranche de 8 bits. Dans notre exemple : 2 octets. A la suite de chaque octet il y a un bit d'acquittement provenant du destinataire.

La fin de transmission est signalée par la condition d'arrêt (Stop). Il s'agit d'un changement d'état des deux lignes SDA et SCL (retour à l'état de repos).

#### 9.1.3.1. LA CONDITION DE DEPART

Une condition de départ est réalisée lorsque la ligne SDA passe du niveau haut au niveau bas alors que SCL est au niveau haut.

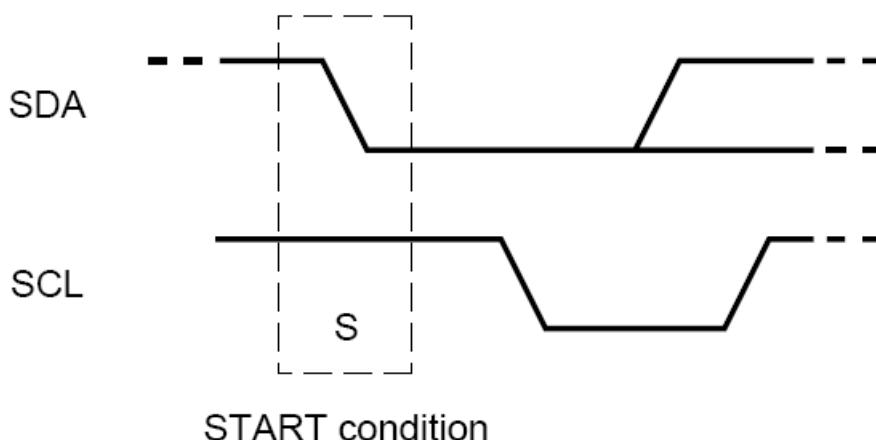
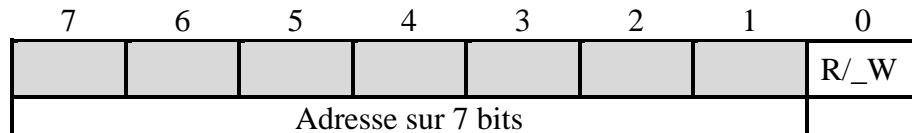


Figure 9-3 : condition de départ

### 9.1.3.2. L'OCTET D'EN-TETE

L'octet d'en-tête se compose de l'adresse (7 bits) et du bit de direction.



Valeur du bit R/\_W :

- 0 = écriture, transfert maître → esclave
- 1 = lecture, transfert esclave → maître

### 9.1.3.3. PRINCIPE DE LA TRANSMISSION DES DONNEES

La Figure 9-4 illustre le principe de la transmission des données. Après la condition de départ, un octet est transmis en commençant par son bit de poids fort. Cet octet est suivi du bit d'acquittement. Il en est ainsi pour chaque octet.

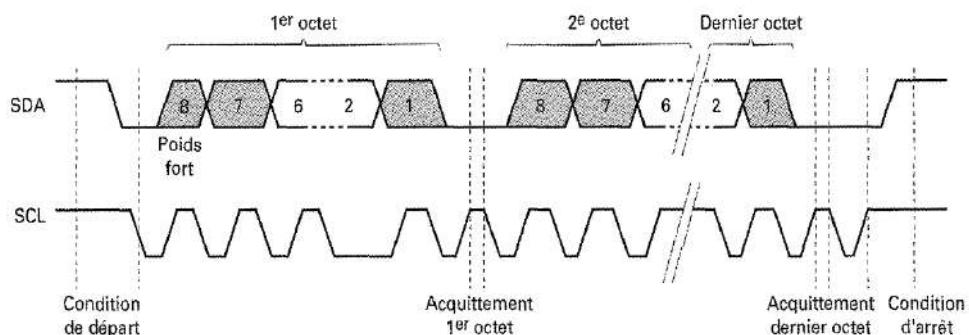


Figure 9-4

### 9.1.3.4. NOTION D'ACQUITTEMENT

La Figure 9-5 montre le détail de l'acquittement par le récepteur. A noter la nécessité du 9ème coup d'horloge.

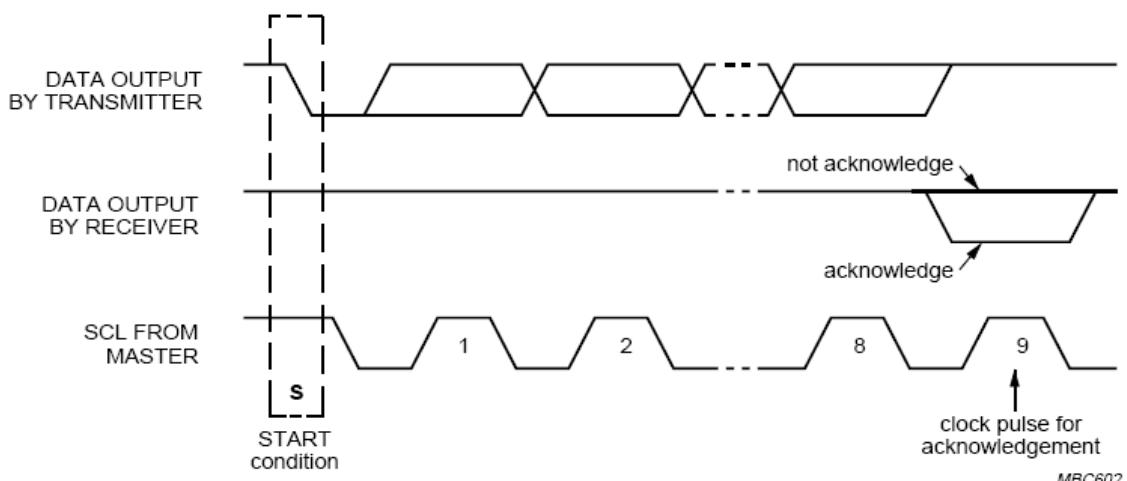


Figure 9-5

### 9.1.3.5. LA CONDITION D'ARRET

Une condition d'arrêt est réalisée lorsque SDA passe du niveau bas au niveau haut alors que SCL est au niveau haut.

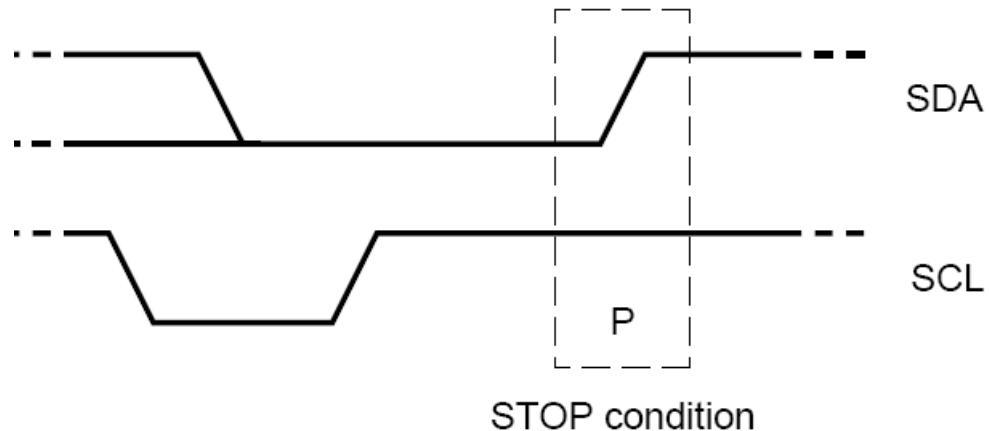


Figure 9-6 : condition d'arrêt

### 9.1.3.6. DETAIL DE LA TRANSMISSION D'UN BIT

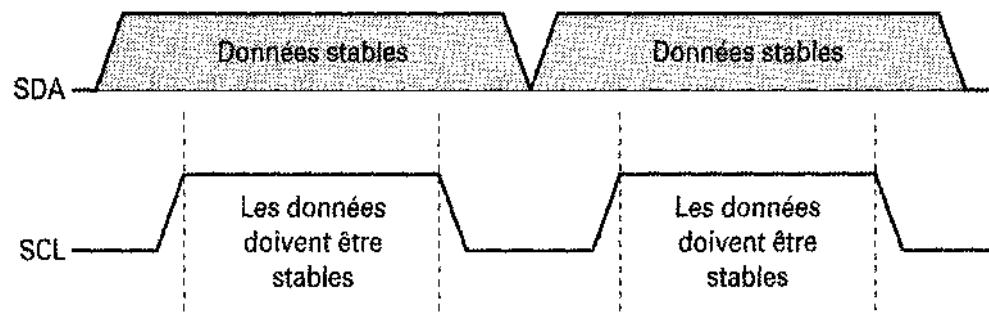


Figure 9-7

Comme le montre la Figure 9-7, une donnée n'est considérée comme valide sur le bus que lorsque le signal SCL est à l'état haut. L'émetteur doit donc positionner la donnée à émettre lorsque SCL est à l'état bas et la maintenir tant que SCL reste à l'état haut.

### 9.1.4. NOTION DE MAITRE ET D'ESCLAVES

Le maître est toujours celui qui initialise le transfert, fournit le signal d'horloge et termine le transfert. L'esclave est le composant adressé par le maître.

Dans une situation simple, le microcontrôleur est le maître et les différents composants des esclaves.

### 9.1.5. SITUATION MULTI-MASTER

Le bus I2C permet la présence de plusieurs maîtres. Cependant, un procédé d'arbitrage empêche la communication simultanée des maîtres.

La Figure 9-8 illustre la procédure d'arbitrage, qui est basée sur le niveau de la ligne SDA pendant que la ligne SCL est à 1. Si un des maîtres fournit un SDA à 1 et l'autre à 0, de par la situation de collecteur ouvert c'est la valeur 0 qui l'emporte. Le maître dont la valeur de donnée ne correspond pas à l'état de la ligne SDA perd la main et laissera la priorité à l'autre.

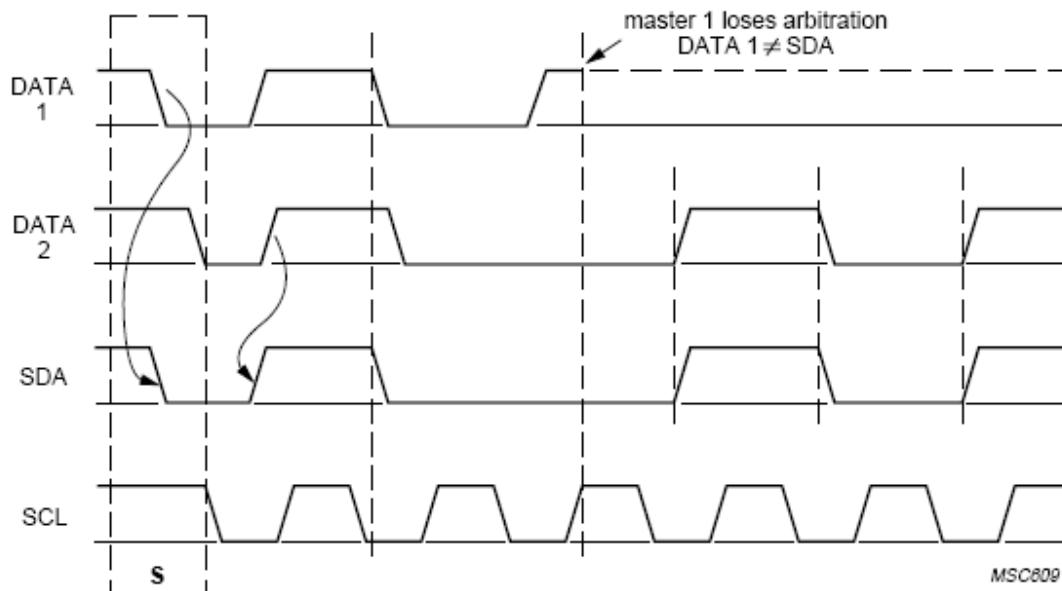


Figure 9-8 : arbitrage entre 2 maîtres

## 9.2. REALISATION DU BUS I2C AVEC LE PIC32MX795F512L

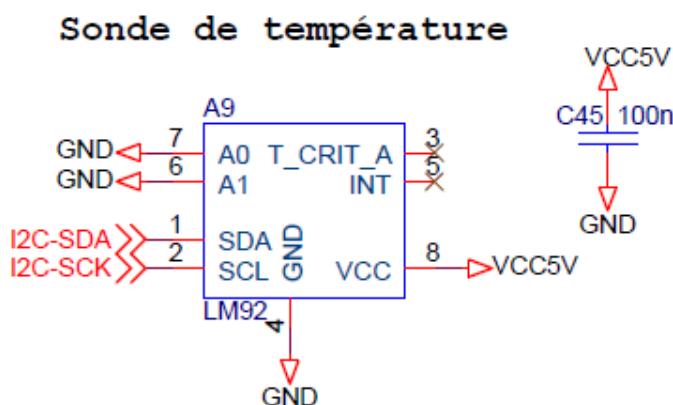
Le PIC32MX dispose de modules spécifiques pour la gestion du bus I2C. Dans le cadre du kit PIC32MX795F512L, c'est le module I2C no 2 qui est utilisé.



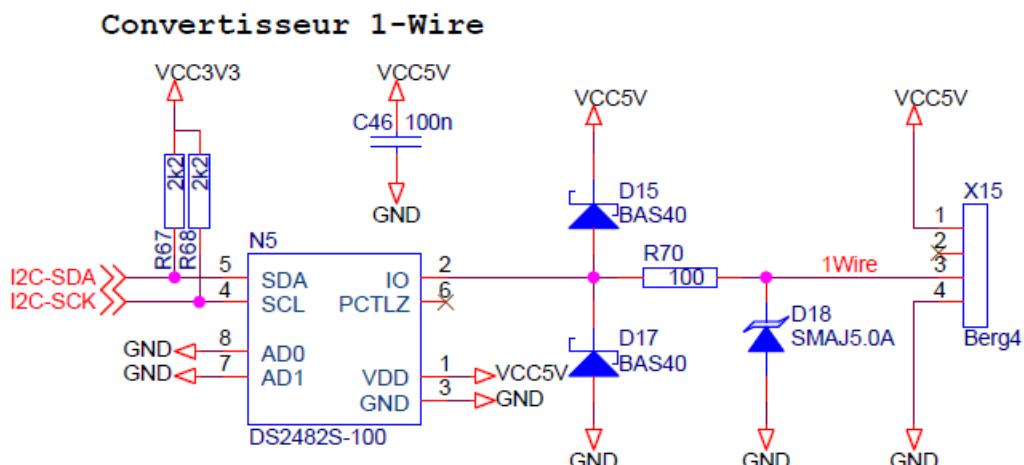
### 9.2.1. COMPOSANT I2C DU KIT PIC32MX795F512L

Il y a trois composants I2C sur le kit PIC32MX795F512L.

#### 9.2.1.1. SONDE DE TEMPERATURE LM92



#### 9.2.1.2. CONVERTISSEUR 1-WIRE DS2482S-100

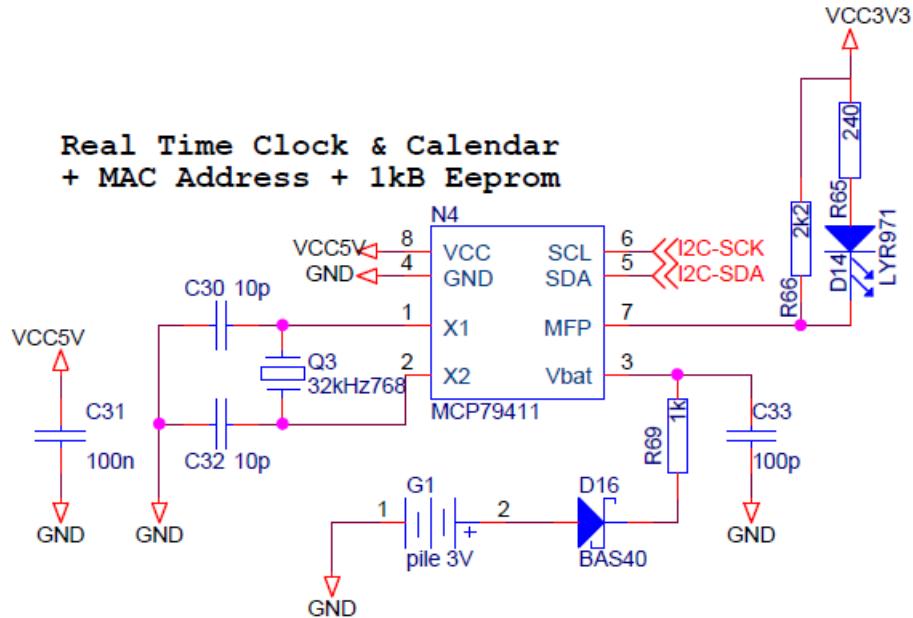


On remarque la paire de résistances de pull up pour le bus I2C.

### 9.2.1.3. RTC, MAC ADRESSE ET EEPROM

Le MCP79411 intègre :

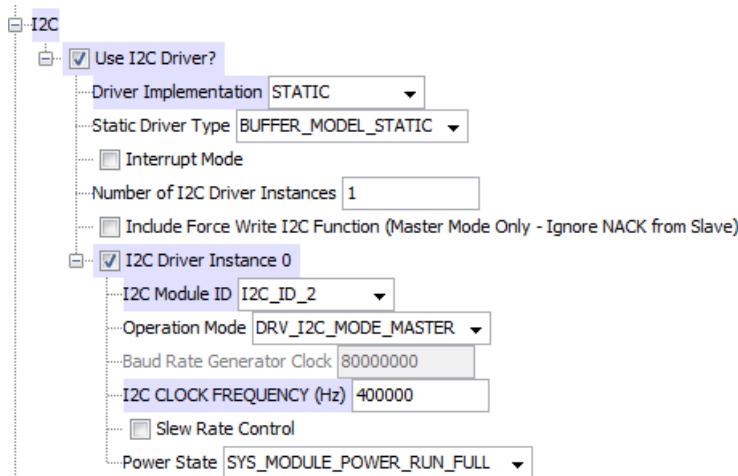
- un Real Time Clock/Calendar,
- une adresse MAC unique préprogrammée d'usine,
- une EEPROM de 1 Kbits, soit 128 octets.



## 9.3. DRIVER I2C FOURNI PAR HARMONY

Créons un driver I2C automatiquement via Harmony et observons le code généré. Cet exemple a été effectué avec Harmony 1.08.

### 9.3.1. CONFIG DU DRIVER I2C



### 9.3.2. FONCTION D'INITIALISATION OBTENUE

```
void DRV_I2C0_Initialize(void)
{
    /* Initialize I2C0 */
    PLIB_I2C_BaudRateSet(I2C_ID_2,
        SYS_CLK_PeripheralFrequencyGet(CLK_BUS_PERIPHERAL_1), 400000);
    PLIB_I2C_StopInIdleDisable(I2C_ID_2);

    /* Low frequency is enabled (**NOTE** PLIB function logic reverted) */
    PLIB_I2C_HighFrequencyEnable(I2C_ID_2); ②

    { i2c0Obj.i2cMode      = DRV_I2C_MODE_MASTER;
      i2c0Obj.transferSize = 0;
      i2c0State           = DRV_I2C_TASK_SEND_DEVICE_ADDRESS; } ①
    QueueInitialize_0();

    /* Enable I2C0 */
    PLIB_I2C_Enable(I2C_ID_2);
}
```

- 1) Comme nous ne nous intéressons ici qu'à l'initialisation du périphérique I2C et non au fonctionnement du driver, la partie d'initialisation des variables peut être ignorée.
- 2) Cette partie correspond à la couche "Slew Rate Control" du MHC. La couche n'est pas mise, donc low frequency.
  - \* Comme l'indique le commentaire, Harmony a une incohérence et c'est la fonction PLIB\_I2C\_HighFrequencyEnable (au lieu de Disable) qui est appelée. Ce comportement pourrait être appelé à évoluer avec les versions de Harmony.
  - \* D'après le datasheet I2C du PIC32, le "slope control" devrait être activé pour des fréquences de clock de 100 kHz et 400 kHz. Toutefois, il est désactivé ici dû

à des problèmes d'incompatibilité du capteur de température LM92 avec des flancs trop lents.

☝ Ce code est fonctionnel. Toutefois, un code d'initialisation plus robuste sera présenté plus loin dans ce chapitre (cf. §9.4.4.3)

## 9.4. FONCTIONS I2C SOUHAITEES

Dans le but de pouvoir facilement mettre en œuvre un nouveau composant I2C ou de migrer la gestion d'un composant I2C réalisée à l'aide du compilateur CCS, vous trouverez dans les fichiers Mc32\_I2cUtilCCS.h et Mc32\_I2cUtilCCS.c les fonctions compatibles.

### 9.4.1. LES FONCTIONS DU COMPILATEUR CCS

Voici la liste des fonctions les plus courantes nécessaires à la gestion de l'I2C (documentation issue du compilateur CCS) :

<code>i2c_start()</code>	Issues a start command when in the I2C master mode.
<code>i2c_write(data)</code>	Sends a single byte over the I2C interface.
<code>i2c_read()</code>	Reads a byte over the I2C interface.
<code>i2c_stop()</code>	Issues a stop command when in the I2C master mode.

- Ajout également d'une fonction `i2c_reStart()` pour la répétition du start.

### 9.4.2. CONFIGURATION DE L'I2C

L'initialisation du bus I2C est faite par un appel à `i2c_init()` (anciennement réalisée via une directive).

### 9.4.3. CONTENU DU FICHIER Mc32\_I2CUTILCCS.H

Voici les prototypes des fonctions I2C :

```
#include <stdbool.h>
#include <stdint.h>

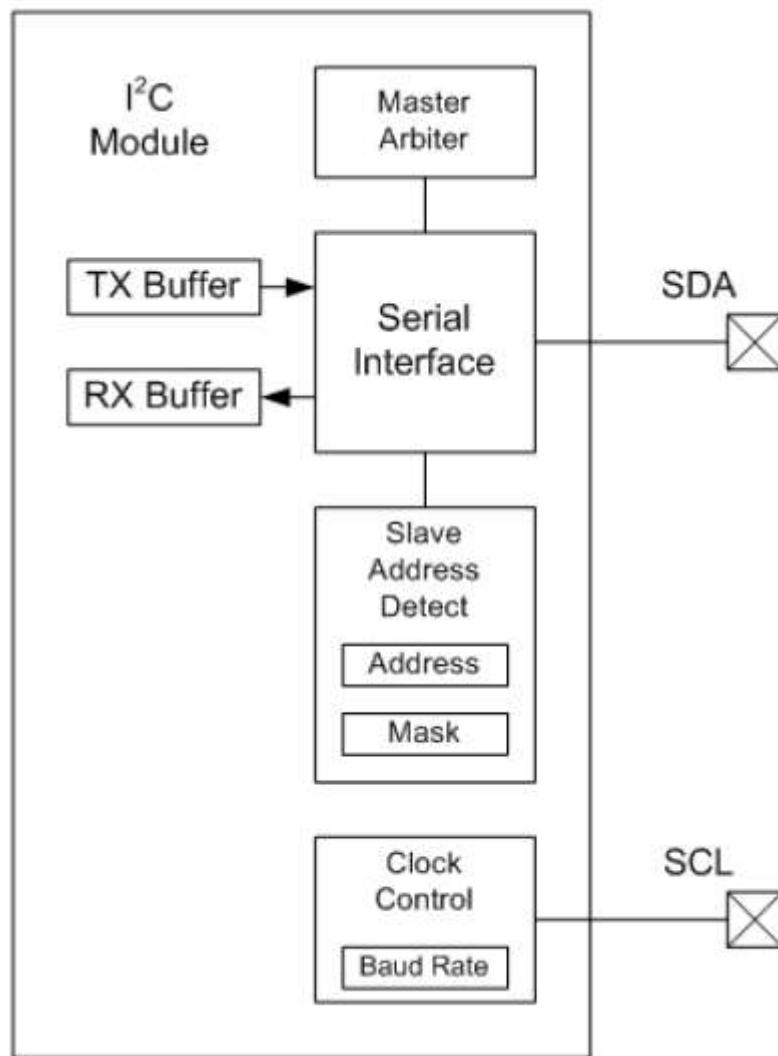
void i2c_init( bool Fast );

void i2c_start(void);
void i2c_reStart(void);
bool i2c_write( uint8_t data );
uint8_t i2c_read(bool ackTodo);
void i2c_stop( void );
```

#### 9.4.4. REALISATION DES FONCTIONS

Ces fonctions utilisent des fonctions élémentaires de gestion du bus I2C. Elles ont été réalisées en utilisant les fonctions I2C de la PLIB.

Dans l'aide de Harmony, section I2C Peripheral Library, on trouve le schéma de principe du module I2C.



##### 9.4.4.1. VUE D'ENSEMBLE DES FONCTIONS DE LA PLIB\_I2C

Voici une partie des fonctions de la plib\_i2c. Les fonctions d'existence et de gestion du slave ne sont pas présentées.

Baud Rate Generator Control Functions

	Name	Description
=♦	<a href="#">PLIB_I2C_BaudRateGet</a>	Calculates the I2C module's current SCL clock frequency.
=♦	<a href="#">PLIB_I2C_BaudRateSet</a>	Sets the desired baud rate.

**General Initialization Functions**

Name	Description
PLIB_I2C_Disable	Disables the specified I2C module.
PLIB_I2C_Enable	Enables the specified I2C module.
PLIB_I2C_GeneralCallDisable	Disables the I2C module from recognizing the general call address.
PLIB_I2C_GeneralCallEnable	Enables the I2C module to recognize the general call address.
PLIB_I2C_HighFrequencyDisable	Disables the I2C module from using high frequency (400 kHz or 1 MHz) signaling.
PLIB_I2C_HighFrequencyEnable	Enables the I2C module to use high frequency (400 kHz or 1 MHz) signaling.
PLIB_I2C_IPMIDisable	Disables the I2C module's support for the IPMI specification
PLIB_I2C_IPMIEnable	Enables the I2C module to support the Intelligent Platform Management Interface (IPMI) specification (see Remarks).
PLIB_I2C_ReservedAddressProtectDisable	Disables the I2C module from protecting reserved addresses, allowing it to respond to them.
PLIB_I2C_ReservedAddressProtectEnable	Enables the I2C module to protect (not respond to) reserved addresses.
PLIB_I2C_SlaveClockStretchingDisable	Disables the I2C module from stretching the slave clock.
PLIB_I2C_SlaveClockStretchingEnable	Enables the I2C module to stretch the slave clock.
PLIB_I2C_SMBDisable	Disable the I2C module support for SMBus electrical signaling levels.
PLIB_I2C_SMBEnable	Enables the I2C module to support System Management Bus (SMBus) electrical signaling levels.
PLIB_I2C_StopInIdleDisable	Disables the Stop-in-Idle feature.
PLIB_I2C_StopInIdleEnable	Enables the I2C module to stop when the processor enters Idle mode

**General Status Functions**

Name	Description
PLIB_I2C_ArbitrationLossClear	Clears the arbitration loss status flag
PLIB_I2C_ArbitrationLossHasOccurred	Identifies if bus arbitration has been lost.
PLIB_I2C_BusIsIdle	Determines whether the I2C bus is idle or busy.
PLIB_I2C_StartClear	Clears the start status flag
PLIB_I2C_StartWasDetected	Identifies when a Start condition has been detected.
PLIB_I2C_StopClear	Clears the stop status flag
PLIB_I2C_StopWasDetected	Identifies when a Stop condition has been detected

**Master Control Functions**

Name	Description
PLIB_I2C_MasterReceiverClock1Byte	Drives the bus clock to receive 1 byte of data from a slave device.
PLIB_I2C_MasterStart	Sends an I2C Start condition on the I2C bus in Master mode.
PLIB_I2C_MasterStartRepeat	Sends a repeated Start condition during an ongoing transfer in Master mode.
PLIB_I2C_MasterStop	Sends an I2C Stop condition to terminate a transfer in Master mode.

**Receiver Control Functions**

Name	Description
PLIB_I2C_ReceivedByteAcknowledge	Allows a receiver to acknowledge a that a byte of data has been received.
PLIB_I2C_ReceivedByteGet	Gets a byte of data received from the I2C bus interface.
PLIB_I2C_ReceivedBytesAvailable	Detects whether the receiver has data available.
PLIB_I2C_ReceiverByteAcknowledgeHasCompleted	Determines if the previous acknowledge has completed.
PLIB_I2C_ReceiverOverflowClear	Clears the receiver overflow status flag.
PLIB_I2C_ReceiverOverflowHasOccurred	Identifies if a receiver overflow error has occurred.
PLIB_I2C_MasterReceiverReadyToAcknowledge	Checks whether the hardware is ready to acknowledge.

**Transmitter Control Functions**

Name	Description
PLIB_I2C_TransmitterByteHasCompleted	Detects whether the module has finished transmitting the most recent byte.
PLIB_I2C_TransmitterByteSend	Sends a byte of data on the I2C bus.
PLIB_I2C_TransmitterByteWasAcknowledged	Determines whether the most recently sent byte was acknowledged.
PLIB_I2C_TransmitterIsBusy	Identifies if the transmitter of the specified I2C module is currently busy (unable to accept more data).
PLIB_I2C_TransmitterIsReady	Detects if the transmitter is ready to accept data to transmit.
PLIB_I2C_TransmitterOverflowClear	Clears the transmitter overflow status flag.
PLIB_I2C_TransmitterOverflowHasOccurred	Identifies if a transmitter overflow error has occurred.

#### 9.4.4.2. #INCLUDE NECESSAIRE ET DEFINITIONS

Nous avons besoin du fichier plib\_i2c.h pour la librairie I2C. L'autre fichier est nécessaire pour la fonction SYS\_CLK\_PeripheralFrequencyGet.

```
#include "app.h"
#include "Mc32_I2cUtilCCS.h"
#include "peripheral\i2c\plib_i2c.h"
#include "peripheral\osc\plib_osc.h"

// KIT PIC32MX795F512L Constants
#define KIT_I2C_BUS    I2C_ID_2
#define I2C_CLOCK_FAST 400000
#define I2C_CLOCK_SLOW 100000
```

¶ Sur le kit, les différents périphériques I2C sont connectés au module I2C no 2.

#### 9.4.4.3. LA FONCTION I2C\_INIT

Cette fonction permet de définir la fréquence de travail du module I2C, de configurer le comportement et finalement d'enclencher le module. Sa réalisation s'inspire du driver i2c généré par Harmony (cf. §9.3), mais avec une petite adaptation.

```
// Initialisation de l'I2C
//      si BOOL Fast = FALSE    LOW speed  100  kHz
//      si BOOL Fast = TRUE     HIGH speed  400  kHz
void i2c_init( bool Fast )
{
    PLIB_I2C_Disable(KIT_I2C_BUS);           // Ajout CHR

    // LOW frequency is enabled (**NOTE** PLIB function logic reverted)
    // A 100k et 400kHz, on devrait activer le "slope control"
    // (cf. § I2C datasheet PIC32). Toutefois, le LM92 a des problèmes
    // d'incompatibilité avec les flancs trop lents => désactivé
    // Voir application note
    // "AN-2113 Applying I2C Compatible Temperature Sensors in Systems with Slow
    // Clock Edges"
    PLIB_I2C_HighFrequencyEnable(KIT_I2C_BUS);
    if (Fast) {
        PLIB_I2C_BaudRateSet(KIT_I2C_BUS,
            SYS_CLK_PeripheralFrequencyGet(CLK_BUS_PERIPHERAL_1),
            I2C_CLOCK_FAST);
    } else {
        PLIB_I2C_BaudRateSet(KIT_I2C_BUS,
            SYS_CLK_PeripheralFrequencyGet(CLK_BUS_PERIPHERAL_1),
            I2C_CLOCK_SLOW);
    }

    PLIB_I2C_SlaveClockStretchingEnable(KIT_I2C_BUS); // ajout CHR
    PLIB_I2C_Enable(KIT_I2C_BUS);
}
```

Il est nécessaire de combiner les fonctions PLIB\_I2C\_HighFrequencyEnable et PLIB\_I2C\_BaudRateSet pour obtenir une fréquence correcte.

Cette solution a été testée avec Harmony 1.06 et 1.08 pour des fréquences d'horloges de 100 kHz et 400 kHz.

#### 9.4.4.4. LA FONCTION I2C\_START

Cette fonction génère la condition de départ en mode master. Elle a été réalisée en adaptant la fonction StartTransfer de l'ancien exemple Microchip.

```
void i2c_start(void)
{
    // Wait for the bus to be idle, then start the transfer
    while(PLIB_I2C_BusIsIdle(KIT_I2C_BUS) == 0);

    /* Check for receive overflow */
    if ( PLIB_I2C_ReceiverOverflowHasOccurred(KIT_I2C_BUS) )
    {
        PLIB_I2C_ReceiverOverflowClear(KIT_I2C_BUS);
    }

    /* Check for transmit overflow */
    if (PLIB_I2C_TransmitterOverflowHasOccurred(KIT_I2C_BUS))
    {
        PLIB_I2C_TransmitterOverflowClear(KIT_I2C_BUS);
    }

    PLIB_I2C_MasterStart(KIT_I2C_BUS);

    if (PLIB_I2C_ArbitrationLossHasOccurred(KIT_I2C_BUS));
    {
        // Handel bus collision
        PLIB_I2C_ArbitrationLossClear(KIT_I2C_BUS);
    }

    // Wait for the signal to complete
    while (PLIB_I2C_StartWasDetected(KIT_I2C_BUS) == false);
} // end i2c_start
```

#### 9.4.4.1. LA FONCTION I2C\_RESTART

Cette fonction permet de répéter le start. La différence est que l'on ne contrôle pas si le bus est en idle et que l'on utilise la fonction **PLIB\_I2C\_MasterStartRepeat**.

```
void i2c_reStart(void)
{
    // Pas d'attente bus en Idle
    /* Check for receive overflow */
    if ( PLIB_I2C_ReceiverOverflowHasOccurred(KIT_I2C_BUS) )
    {
        PLIB_I2C_ReceiverOverflowClear(KIT_I2C_BUS);
    }

    /* Check for transmit overflow */
    if (PLIB_I2C_TransmitterOverflowHasOccurred(KIT_I2C_BUS))
    {
        PLIB_I2C_TransmitterOverflowClear(KIT_I2C_BUS);
    }

    PLIB_I2C_MasterStartRepeat(KIT_I2C_BUS);

    if (PLIB_I2C_ArbitrationLossHasOccurred(KIT_I2C_BUS));
    {
        // Handel bus collision
        PLIB_I2C_ArbitrationLossClear(KIT_I2C_BUS);
    }
}
```

```
// Wait for the signal to complete
while (PLIB_I2C_StartWasDetected(KIT_I2C_BUS) == false);
} // end i2c_reStart
```

#### 9.4.4.2. LA FONCTION I2C\_WRITE

Cette fonction effectue l'envoi d'un octet sur le bus I2C.

Syntaxe :      **i2c\_write** (data);

Le paramètre data est la valeur 8 bits à transmettre.

En mode maître, la fonction génère le signal d'horloge. En outre, cette fonction retourne le bit ACK (0 = ACK, 1 = NO\_ACK). Réalisation sur la base de la fonction TransmitOneByte de l'ancien exemple Microchip.

```
bool i2c_write( uint8_t data )
{
    bool AckBit;

    // Wait for the bus to be idle (nécessaire après un reStart)
    while(PLIB_I2C_BusIsIdle(KIT_I2C_BUS) == false);

    // Wait for the transmitter to be ready
    while( PLIB_I2C_TransmitterIsBusy(KIT_I2C_BUS) == true);

    // Transmit the byte
    PLIB_I2C_TransmitterByteSend(KIT_I2C_BUS, data);

    // Wait as long as TBF = 1
    while(PLIB_I2C_TransmitterIsBusy(KIT_I2C_BUS));

    // Wait as long as TRSTAT == 1
    while(!PLIB_I2C_TransmitterByteHasCompleted(KIT_I2C_BUS));

    // Gestion du bit d'acknowledge
    AckBit = PLIB_I2C_TransmitterByteWasAcknowledged(KIT_I2C_BUS);

    return AckBit;
} // end i2c_write
```

#### 9.4.4.3. LA FONCTION I2C\_READ

Cette fonction effectue la lecture d'un octet sur le bus I2C.

Syntaxe :      data = i2c\_read (**ack**);

La fonction retourne l'octet lu.

Paramètre **ack** :

- 1 (true) signifie qu'il faut effectuer l'acquittement.
- 0 (false) signifie qu'il ne faut pas effectuer l'acquittement.

En mode maître, la fonction génère le signal d'horloge. Cette fonction est obtenue en partie à partir de l'ancien et du nouvel exemple Microchip.

```
uint8_t i2c_read(bool ackTodo)
{
    uint8_t i2cByte;
    BSP_LEDOn(BSP_LED_5); // provisoire : pour observation
// ajout idem driver statique I2C de Harmony 1_03
    if ( PLIB_I2C_ReceiverOverflowHasOccurred(KIT_I2C_BUS) )
    {
        i2cByte = PLIB_I2C_ReceivedByteGet(KIT_I2C_BUS);
        PLIB_I2C_ReceiverOverflowClear(KIT_I2C_BUS);
    }

    // en relation avec stretching
    PLIB_I2C_SlaveClockRelease(KIT_I2C_BUS);

    //Set Rx enable in MSTR which causes SLAVE to send data
    PLIB_I2C_MasterReceiverClock1Byte(KIT_I2C_BUS);

    // Wait till RBF = 1; Which means data is available in I2C2RCV reg
    while (!PLIB_I2C_ReceivedByteIsAvailable(KIT_I2C_BUS));

    //Read from I2CxRCV
    i2cByte = PLIB_I2C_ReceivedByteGet(KIT_I2C_BUS);
    while ( PLIB_I2C_MasterReceiverReadyToAcknowledge
                ( KIT_I2C_BUS ) == false );

    if (ackTodo) {
        PLIB_I2C_ReceivedByteAcknowledge ( KIT_I2C_BUS, true );
    } else {
        PLIB_I2C_ReceivedByteAcknowledge ( KIT_I2C_BUS, false );
    }

    // wait till ACK/NACK sequence is complete i.e ACKEN = 0
    while( PLIB_I2C_MasterReceiverReadyToAcknowledge
                ( KIT_I2C_BUS ) == false);

    BSP_LEDOff(BSP_LED_5); // provisoire : pour observation
    return i2cByte;
} // end i2c_read
```

#### 9.4.4.4. LA FONCTION I2C\_STOP

Cette fonction génère la condition d'arrêt en mode master.

Syntaxe : **i2c\_stop ()**;

Réalisation sur la base du nouvel exemple Microchip.

```
void i2c_stop( void )
{
    // Attente bus en idle
    while(PLIB_I2C_BusIsIdle(KIT_I2C_BUS) == false);
    PLIB_I2C_MasterStop(KIT_I2C_BUS);
    // Wait for the signal to complete
    while (PLIB_I2C_StopWasDetected(KIT_I2C_BUS) == false);
} // end i2c_stop
```

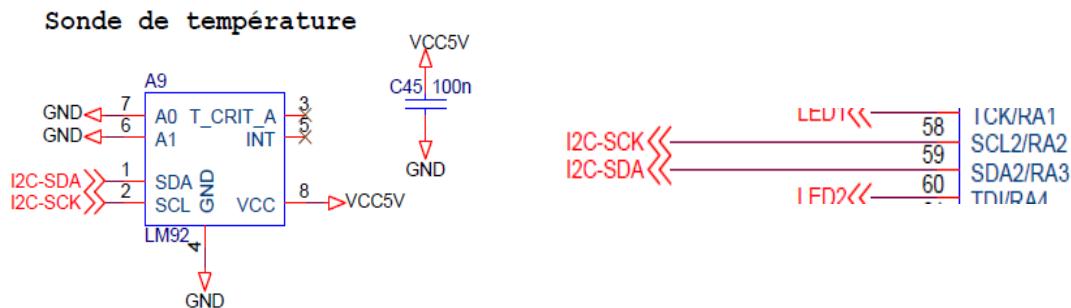
## 9.5. COMMUNICATION AVEC LE COMPOSANT LM92

Nous allons illustrer l'utilisation des fonctions I2C décrites précédemment en prenant comme exemple la communication avec le composant LM92, qui est un capteur de température avec bus I2C. Nous allons présenter deux façons de lire la température :

- Lecture simple (pointeur = registre température). Soit par défaut, soit suite à configuration.
- Etablissement du pointeur, puis lecture de la température. Cette méthode peut être adaptée pour lire une autre info que la température.

### 9.5.1. CONNEXION ENTRE LE PIC32MX ET LE LM92

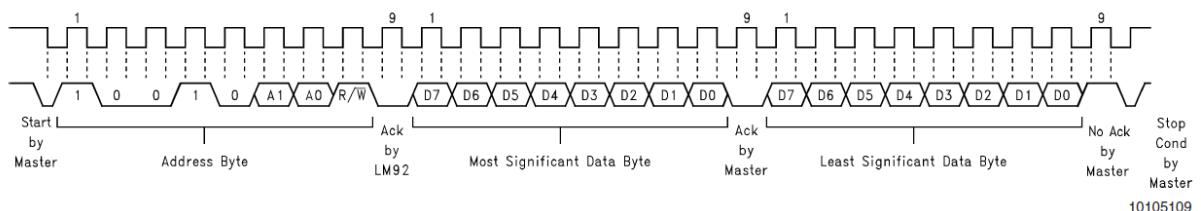
Utilisation du module I2C no 2, RA2 pour le signal SCL et RA3 pour le signal SDA.



Remarque : les résistances de pull-up de 2k2 sont présentes à un autre endroit du schéma.

### 9.5.2. LECTURE REGISTRE TEMPERATURE DU LM92

Pour lire le registre de température du LM92, il faut respecter la séquence indiquée par la Figure 9-9.



Typical 2-Byte Read From Preset Pointer Location Such as Temp or Comparison Registers

Figure 9-9

On remarque qu'il y a envoi de l'adresse avec le bit de poids faible à 1 pour indiquer la lecture. Il faut aussi noter que lors de la lecture du msb il y a acquittement par le maître, alors que pour la lecture du lsb, il n'y a pas d'acquittement.

### 9.5.2.1. REALISATION DE LA LECTURE

La fonction **I2C\_ReadRawTempLM92** sert à lire les données brutes représentant la température. Cette fonction est fournie dans les fichiers Mc32gestI2cLm92.h et Mc32gestI2cLm92.c.

```
int16_t I2C_ReadRawTempLM92(void)
{
    // Déclaration des variables
    uint8_t msb = 1;
    uint8_t lsb = 1;
    int16_t RawTemp;

    BSP_LEDToggle(BSP_LED_6); // provisoire : pour observation
    i2c_start();
    i2c_write(lm92_rd); // adresse + lecture
    msb = i2c_read(1); // ack
    lsb = i2c_read(0); // no ack
    i2c_stop();

    BSP_LEDToggle(BSP_LED_6); // provisoire : pour observation
    RawTemp = msb;
    RawTemp = RawTemp << 8;
    RawTemp = RawTemp | lsb;
    return RawTemp;
} // end I2C_ReadRawTempLM92
```

### 9.5.2.2. OBSERVATION DE LA SEQUENCE

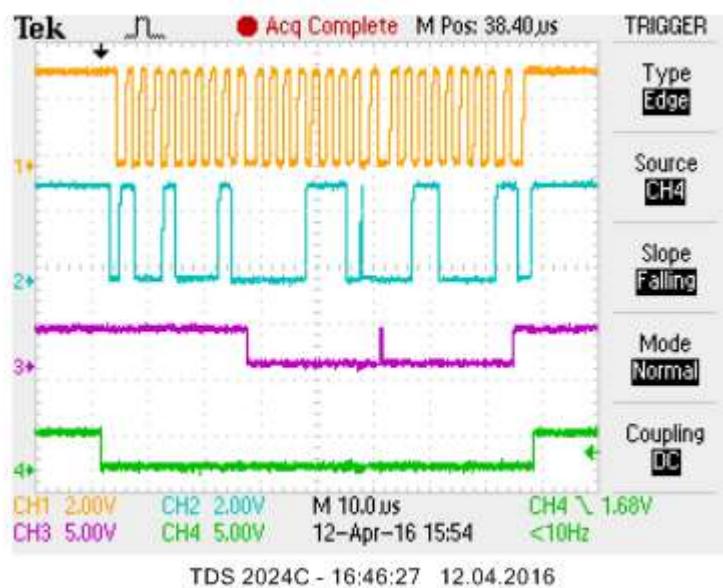
L'action sur la LED\_6 marque le début et la fin de la fonction **I2C\_ReadRawTempLM92**. La LED\_5 est activée dans la fonction **i2c\_read**.

Canal 1 : SCK  
I2C-SCK / SCL2/RA2  
PORTAbits.RA2 / pin 58

Canal 2 : SDA  
I2C-SDA / SDa2/RA3  
PORTAbits.RA3 / pin 59

Canal 3 : LED\_5  
marqueur de la fonction  
i2c\_read

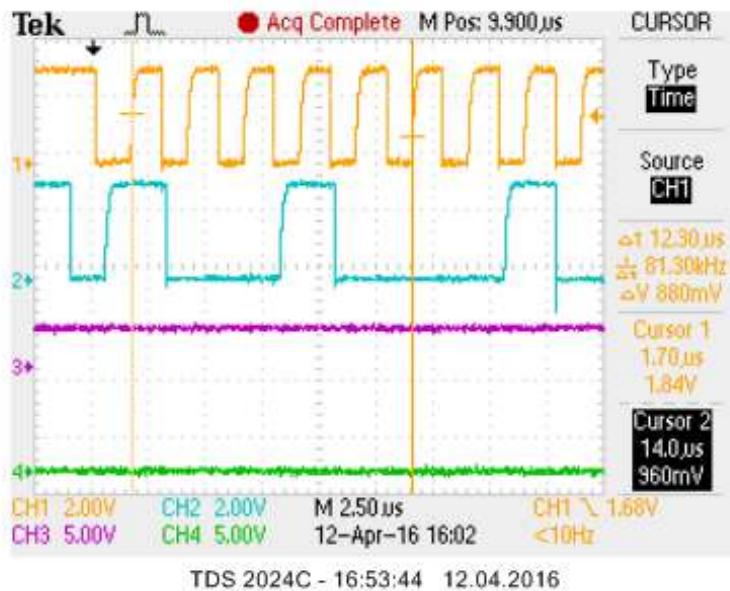
Canal 4 : LED\_6  
marqueur de la fonction  
I2C\_ReadRawTempLM92



On obtient bien 3 séries de 9 coups d'horloge. Et si on convertit la température brute en degré, on obtient un résultat cohérent.

### 9.5.2.3. CONTROLE DE LA PERIODE DE SCL EN FAST

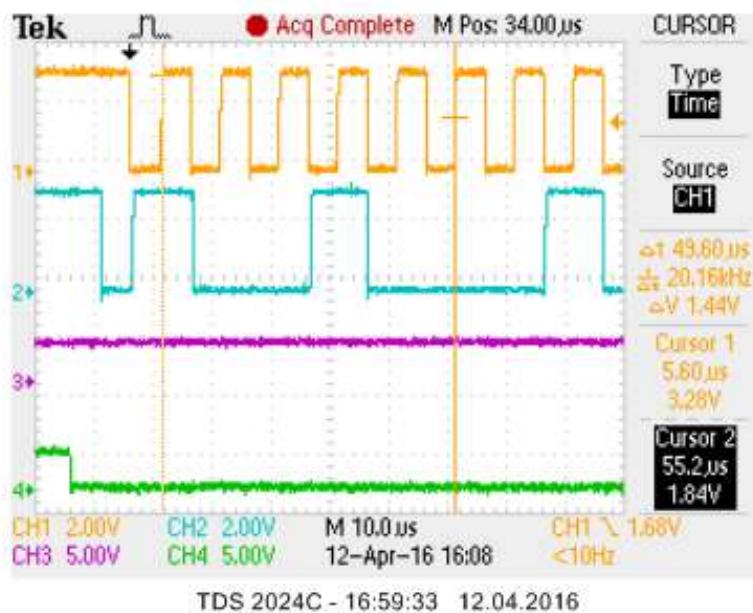
En modifiant la base de temps de la mesure précédente, il est possible avec les curseurs de mesurer la période du signal SCL.



La mesure de 5 périodes nous donne 12,3 us d'où 2,46 us pour une période, ce qui est assez proche des 2.5 us qui correspondent à 400 kHz. Cela nous donne une fréquence de 406,5 kHz que le LM92 semble supporter sans problème.

### 9.5.2.4. CONTROLE DE LA PERIODE DE SCL EN SLOW

En modifiant l'initialisation et en sélectionnant slow, on obtient :



La mesure de 5 périodes nous donne 49,6 us d'où 9,92 us pour une période, ce qui est assez proche des 10 us qui correspondent à 100 kHz. Cela nous donne une fréquence de 100,8 kHz.

### 9.5.3. SET POINTER ET LECTURE TEMPERATURE DU LM92

Pour lire la température, l'autre solution consiste à sélectionner le registre de température, puis à effectuer la lecture de la température. Voici la séquence à respecter pour cela.

La Figure 9-10 nous montre le début de la séquence (sélection du pointeur), tandis que la Figure 9-11 nous montre la partie lecture.

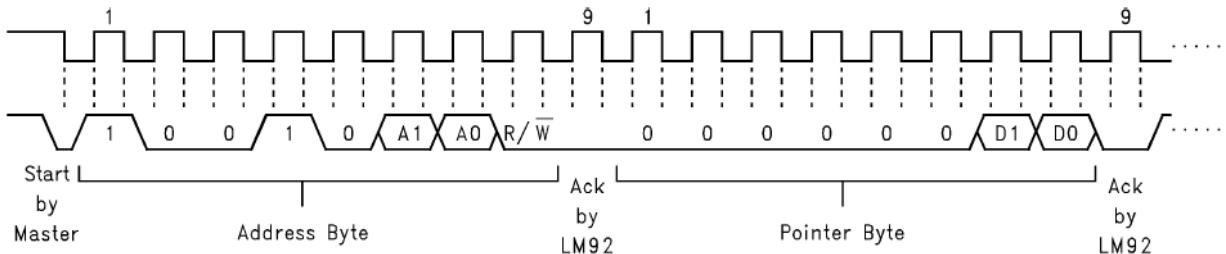


Figure 9-10

On observe que le bit R/\_W est à 0 pour indiquer une action d'écriture. Le deuxième octet contient l'adresse du pointeur.

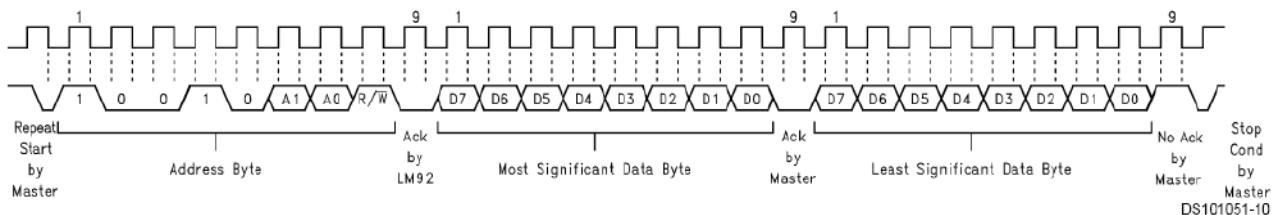


Figure 9-11

On remarque qu'il y a à nouveau envoi de l'adresse, mais cette fois-ci avec le bit de poids faible à 1 pour indiquer la lecture. Il faut aussi noter que lors de la lecture du msb, il y a acquittement par le maître, alors que pour la lecture du lsb il n'y a pas d'acquittement. Il y a répétition du start, raison de la réalisation d'une fonction i2c\_reStart.

#### 9.5.3.1. REALISATION DE LA LECTURE AVEC CONFIG

La fonction WriteCfgReadRawTempLM92 ci-dessous effectue la lecture en commençant par configurer le pointeur de lecture. On constate un 2<sup>ème</sup> start (réalisé par la fonction reStart) avec l'envoi à nouveau de l'adresse en indiquant la lecture. On obtient d'abord le poids fort et ensuite le poids faible. A noter la 1<sup>ère</sup> lecture avec acquittement, la 2<sup>ème</sup> sans.

```

int16_t I2C_WriteCfgReadRawTempLM92(void)
{
    //Déclaration des variables
    uint8_t msb = 1;
    uint8_t lsb = 1;
    int16_t RawTemp;

    BSP_LEDOn(BSP_LED_6); // provisoire : pour observation
    i2c_start();
    i2c_write(lm92_wr); // adresse + écriture
    i2c_write(lm92_temp_ptr); // sélection ptr. temp.

    i2c_reStart();
    i2c_write(lm92_rd); // adresse + lecture
    msb = i2c_read(1); // ack
    lsb = i2c_read(0); // no ack
    i2c_stop();

    BSP_LEDOff(BSP_LED_6); // provisoire : pour observation
    RawTemp = msb;
    RawTemp = RawTemp << 8;
    RawTemp = RawTemp | lsb;
    return RawTemp;
} // end I2C_WriteCfgReadRawTempLM92

```

### 9.5.3.2. OBSERVATION DE LA SEQUENCE

L'action sur la LED\_6 marque le début et la fin de la fonction **I2C\_WriteCfgReadRawTempLM92**. La LED\_5 est activée dans la fonction **i2c\_read**.

Canal 1 : SCK  
I2C-SCK / SCL2/RA2  
PORTAbits.RA2 / pin 58

Canal 2 : SDA  
I2C-SDA / SDa2/RA3  
PORTAbits.RA3 / pin 59

Canal 3 : LED\_5  
marqueur de la fonction  
i2c\_read

Canal 4 : LED\_6  
marqueur de la fonction  
I2C\_ReadRawTempLM92



On observe les 5 séries de 9 coups d'horloge. Le marqueur de l'action **i2c\_read** permet de bien repérer l'action de lecture.

La mesure au curseur du signal marquant l'exécution de la fonction nous indique 127 us.

### 9.5.4. CONFIGURATION DU I2C POUR LE LM92

Avant de pouvoir utiliser les fonctions I2C, il est nécessaire d'initialiser le module I2C.

Ceci est réalisé par la fonction I2C\_InitLM92, qui utilise la fonction i2c\_init ainsi que la fonction I2C\_WriteConfigLM92.

```
void I2C_InitLM92(void) {
    bool Fast = true;
    i2c_init( Fast );
    I2C_WriteConfigLM92();
}
```

#### 9.5.4.1. LA FONCTION I2C\_WRITECONFIGLM92

Cette fonction établit le pointeur sur le registre de température et effectue la lecture de la température. En s'inspirant de cette fonction, il est possible de configurer les alarmes du LM92.

```
void I2C_WriteConfigLM92(void)
{
    //Déclaration des variables
    uint8_t tmp;

    i2c_start();
    i2c_write(lm92_wr);           // adresse + écriture
    i2c_write(lm92_temp_ptr);     // sélection ptr. temp.
    i2c_reStart();
    i2c_write(lm92_rd);          // adresse + lecture
    tmp = i2c_read(1);           // ack
    tmp = i2c_read(0);           // no ack
    i2c_stop();
}
```

#### 9.5.4.2. DEFINITIONS DE L'ADRESSE DU LM92

Le fabricant indique que l'adresse est sur 7 bits avec 5 bits fixes (propre au LM92) et 2 configurables par les lignes A0 et A1. Ces 2 lignes étant connectées à la masse on obtient la situation suivante pour la valeur de l'adresse :

7	6	5	4	3	2	1	0
1	0	0	1	0	A1=0	A0=0	R/_W

Adresse sur 7 bits

D'où les deux définitions :

```
// Définition pour LM92
#define lm92_rd    0x91 // lm92 address for read
#define lm92_wr    0x90 // lm92 address for write
```

### 9.5.4.3. DEFINITIONS DU POINTEUR DE TEMPERATURE

L'organisation de la valeur 8 bits pour la sélection du pointeur est la suivante:

P7	P6	P5	P4	P3	P2	P1	P0
0	0	0	0	0			Register Select

P2	P1	P0	Register
0	0	0	Temperature (Read only) (Power-up default)
0	0	1	Configuration (Read/Write)
0	1	0	T_HYST (Read/Write)
0	1	1	T_CRIT (Read/Write)
1	0	0	T_LOW (Read/Write)
1	0	1	T_HIGH (Read/Write)
1	1	1	Manufacturer's ID

Ce qui conduit à la définition suivante :

```
#define lm92_temp_ptr 0x00 // adr. pointeur température
```

### 9.5.4.4. ORGANISATION DU REGISTRE DE TEMPERATURE

L'exemple de code fournit 2 octets correspondant au registre de température. Pour afficher la température il est nécessaire d'effectuer un regroupement sur 16 bits.

Puis d'effectuer un décalage de 3 bits à droite ou de diviser par 8.

#### 1.10 TEMPERATURE REGISTER

(Read Only):

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Sign	MSB	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	CRIT	HIGH	LOW

D0–D2: Status Bits

D3–D15: Temperature Data. One LSB = 0.0625°C. Two's complement format.

Le bit de poids faible représente 0.0625 degré soit 1/16, si on souhaite exprimer la température en degré, il faut multiplier par 0.0625.

### 9.5.4.5. LA FONCTION CONVRAWTODEG

La fonction LM92\_ConvRawToDeg effectue la conversion de la température brute en une valeur en degré.

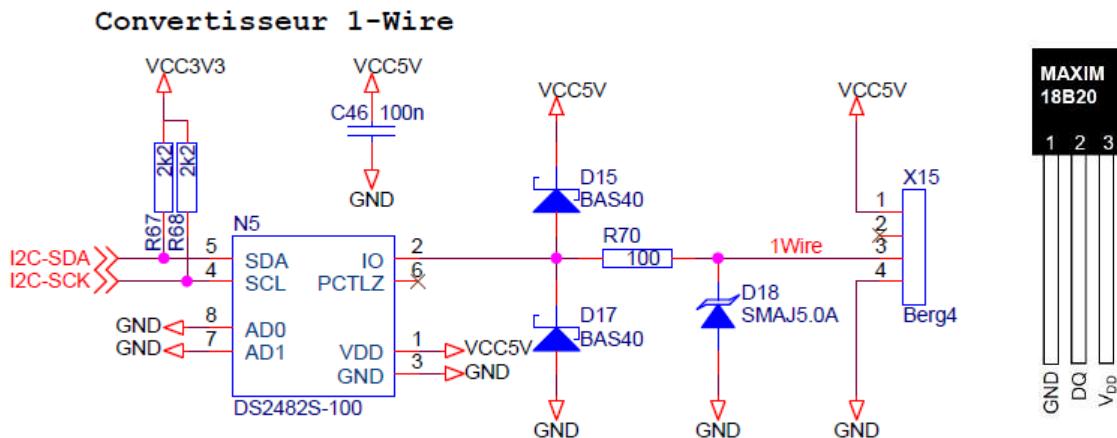
Utilisation d'un passage par référence suite à des problèmes rencontrés lors de l'utilisation dans une réponse à l'interruption.

```
void LM92_ConvRawToDeg( int16_t RowTemp, float *pTemp)
{
    float TempLoc;

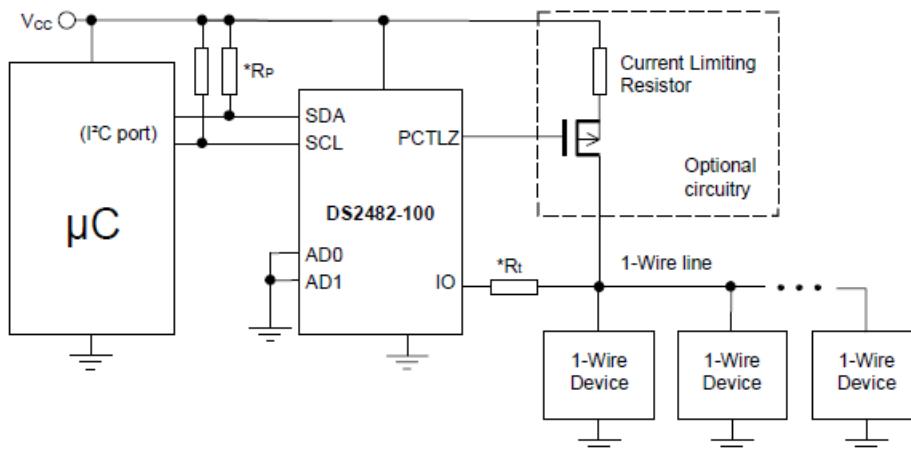
    RowTemp = RowTemp / 8;
    // bit poids faible = 0.0625 degré
    TempLoc = RowTemp * 0.0625;
    *pTemp = TempLoc;
}
```

## 9.6. MISE EN ŒUVRE DU DS2482-100

Le DS2482-100 étant un convertisseur I2C en OneWire, nous allons donc connecter un composant OneWire comme un capteur de température DS18B20 afin de pouvoir réaliser un test complet.



### 9.6.1. VUE D'ENSEMBLE DE PRINCIPE



### 9.6.2. FONCTIONNEMENT DU DS2482-100

Le DS2482-100 gère le signal OneWire. Il exécute 8 commandes différentes nécessaires à la communication. Elles sont réparties en 4 catégories, device control, I<sup>2</sup>C communication, 1-Wire set-up and 1-Wire communication.

Voici sans trop de détails ni traduction, ces commandes. Avec tout d'abord le détail du registre de statut du DS2482-100.

#### 9.6.2.1. REGISTRE DE STATUT DU DS2482-100

##### Status Register Bit Assignment

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
DIR	TSB	SBR	RST	LL	SD	PPD	1WB

### 9.6.2.1.1. Détail des bits du registre

#### 1-Wire Busy (1WB)

The 1WB bit reports to the host processor whether the 1-Wire line is busy. During 1-Wire communication 1WB is 1; once the command is completed, 1WB returns to its default 0. Details on when 1WB changes state and for how long it remains at 1 are found in the *Function Commands* section.

#### Presence-Pulse Detect (PPD)

The PPD bit is updated with every 1-Wire Reset command. If the DS2482 detects a presence pulse from a 1-Wire device at  $t_{MSP}$  during the Presence Detect cycle, the PPD bit will be set to 1. This bit returns to its default 0 if there is no presence pulse or if the 1-Wire line is shorted during a subsequent 1-Wire Reset command.

#### Short Detected (SD)

The SD bit is updated with every 1-Wire Reset command. If the DS2482 detects a logic 0 on the 1-Wire line at  $t_{S1}$  during the Presence Detect cycle, the SD bit is set to 1. This bit returns to its default 0 with a subsequent 1-Wire Reset command provided that the short has been removed. If SD is 1, PPD is 0. The DS2482 cannot distinguish between a short and a DS1994 or DS2404 signaling a 1-Wire interrupt. For this reason, if a DS2404/DS1994 is used in the application, the interrupt function must be disabled. The interrupt signaling is explained in the respective device data sheets.

#### Logic Level (LL)

The LL bit reports the logic state of the active 1-Wire line without initiating any 1-Wire communication. The 1-Wire line is sampled for this purpose every time the Status register is read. The sampling and updating of the LL bit takes place when the host processor has addressed the DS2482 in read mode (during the acknowledge cycle), provided that the Read Pointer is positioned at the Status register.

#### Device Reset (RST)

If the RST bit is 1, the DS2482 has performed an internal reset cycle, either caused by a power-on reset or from executing the Device Reset command. The RST bit is cleared automatically when the DS2482 executes a Write Configuration command to restore the selection of the desired 1-Wire features.

#### Single Bit Result (SBR)

The SBR bit reports the logic state of the active 1-Wire line sampled at  $t_{MSR}$  of a 1-Wire Single Bit command or the first bit of a 1-Wire Triplet command. The power-on default of SBR is 0. If the 1-Wire Single Bit command sends a 0-bit, SBR should be 0. With a 1-Wire Triplet command, SBR could be 0 as well as 1, depending on the response of the 1-Wire devices connected. The same result applies to a 1-Wire Single Bit command that sends a 1-bit.

#### Triplet Second Bit (TSB)

The TSB bit reports the logic state of the active 1-Wire line sampled at  $t_{MSR}$  of the second bit of a 1-Wire Triplet command. The power-on default of TSB is 0. This bit is updated only with a 1-Wire Triplet command and has no function with other commands.

#### Branch Direction taken (DIR)

Whenever a 1-Write Triplet command is executed, this bit reports to the host processor the search direction that was chosen by the 3<sup>rd</sup> bit of the triplet. The power-on default of DIR is 0. This bit is updated only with a 1-Wire Triplet command and has no function with other commands. For additional information see the description of the 1-Wire Triplet command and the Dallas Application Note 187, "1-Wire Search Algorithm".

### 9.6.2.2. COMMAND, DEVICE RESET

#### Device Reset

<b>Command Code</b>	F0h
<b>Command Parameter</b>	None
<b>Description</b>	Performs a global reset of device state machine logic. Terminates any ongoing 1-Wire communication.
<b>Typical Use</b>	Device initialization after power-up; re-initialization (reset) as desired.
<b>Restriction</b>	None (can be executed at any time)
<b>Error Response</b>	None
<b>Command Duration</b>	Maximum 525ns, counted from falling SCL edge of the command code acknowledge bit.
<b>1-Wire Activity</b>	Ends maximum 262.5ns after the falling SCL edge of the command code acknowledge bit.
<b>Read Pointer Position</b>	Status register (for busy polling)
<b>Status Bits Affected</b>	RST set to 1, 1WB, PPD, SD, SBR, TSB, DIR set to 0
<b>Configuration Bits Affected</b>	1WS, APU, PPM, SPU set to 0

### 9.6.2.3. COMMAND, SET READ POINTER

#### Set Read Pointer

<b>Command Code</b>	E1h
<b>Command Parameter</b>	Pointer Code
<b>Description</b>	Sets the Read Pointer to the specified register. Overwrites the read pointer position of any 1-Wire communication command in progress.
<b>Typical Use</b>	To prepare reading the result from a 1-Wire Byte command; random read access of registers.
<b>Restriction</b>	None (can be executed at any time)
<b>Error Response</b>	If the pointer code is not valid, the pointer code is not acknowledged and the command is ignored.
<b>Command Duration</b>	None; the read pointer is updated on the rising SCL edge of the pointer code acknowledge bit.
<b>1-Wire Activity</b>	Not affected
<b>Read Pointer Position</b>	As specified by the pointer code
<b>Status Bits Affected</b>	None
<b>Configuration Bits Affected</b>	None

#### Valid Pointer Codes

Register Selection	Code
Status Register	F0h
Read Data Register	E1h
Configuration Register	C3h

### 9.6.2.4. COMMAND, WRITE CONFIGURATION

#### Write Configuration

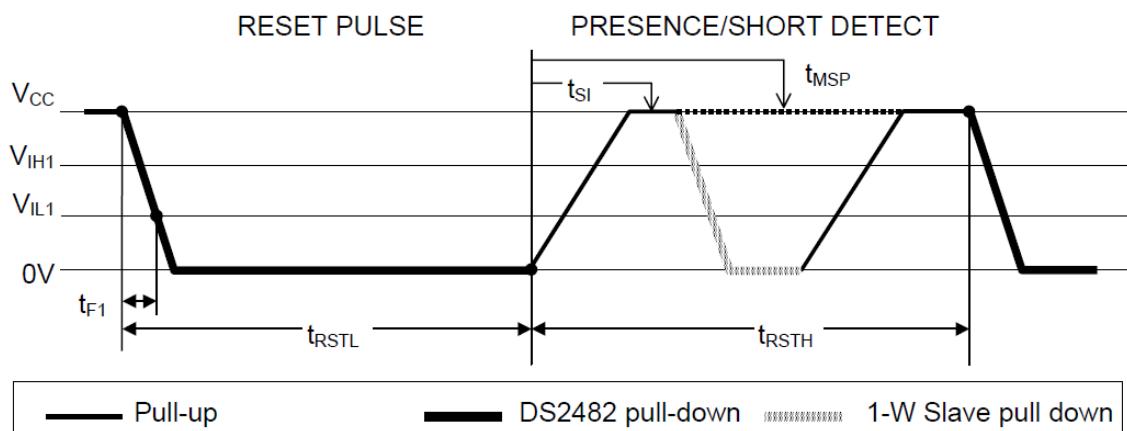
<b>Command Code</b>	D2h
<b>Command Parameter</b>	Configuration Byte
<b>Description</b>	Writes a new configuration byte. The new settings take effect immediately. <b>NOTE:</b> When writing to the Configuration register, the new data is accepted only if the upper nibble (bits 7 to 4) is the one's complement of the lower nibble (bits 3 to 0). When read, the upper nibble is always 0h.
<b>Typical Use</b>	Defining the features for subsequent 1-Wire communication.
<b>Restriction</b>	1-Wire activity must have ended before the DS2482 can process this command.
<b>Error Response</b>	Command code and parameter are not acknowledged if 1WB = 1 at the time the command code is received and the command is ignored.
<b>Command Duration</b>	None; the Configuration register is updated on the rising SCL edge of the configuration byte acknowledge bit.
<b>1-Wire Activity</b>	None
<b>Read Pointer Position</b>	Configuration register (to verify write)
<b>Status Bits Affected</b>	RST set to 0
<b>Configuration Bits Affected</b>	1WS, SPU, PPM, APU updated

### 9.6.2.5. COMMAND, 1-WIRE RESET

#### 1-Wire Reset

<b>Command Code</b>	B4h
<b>Command Parameter</b>	None
<b>Description</b>	Generates a 1-Wire Reset/Presence Detect cycle (Figure 5) at the 1-Wire line. The state of the 1-Wire line is sampled at $t_{SI}$ and $t_{MSP}$ and the result is reported to the host processor through the Status register, bits PPD and SD.
<b>Typical Use</b>	To initiate or end any 1-Wire communication sequence.
<b>Restriction</b>	1-Wire activity must have ended before the DS2482 can process this command.
<b>Error Response</b>	Command code is not acknowledged if $1WB = 1$ at the time the command code is received and the command is ignored.
<b>Command Duration</b>	$t_{RSTL} + t_{RSTH} + \text{maximum } 262.5\text{ns}$ , counted from the falling SCL edge of the command code acknowledge bit.
<b>1-Wire Activity</b>	Begins maximum 262.5ns after the falling SCL edge of the command code acknowledge bit.
<b>Read Pointer Position</b>	Status register (for busy polling)
<b>Status Bits Affected</b>	1WB (set to 1 for $t_{RSTL} + t_{RSTH}$ ), PPD is updated at $t_{RSTL} + t_{MSP}$ , SD is updated at $t_{RSTL} + t_{SI}$
<b>Configuration Bits Affected</b>	1WS, PPM, APU apply

### 9.6.2.6. SIGNAL ISSU DE LA COMMAND 1-WIRE RESET



For presence pulse masking and pull-up details see Figure 3.

### 9.6.2.7. COMMAND, 1-WIRE SINGLE BIT

#### 1-Wire Single Bit

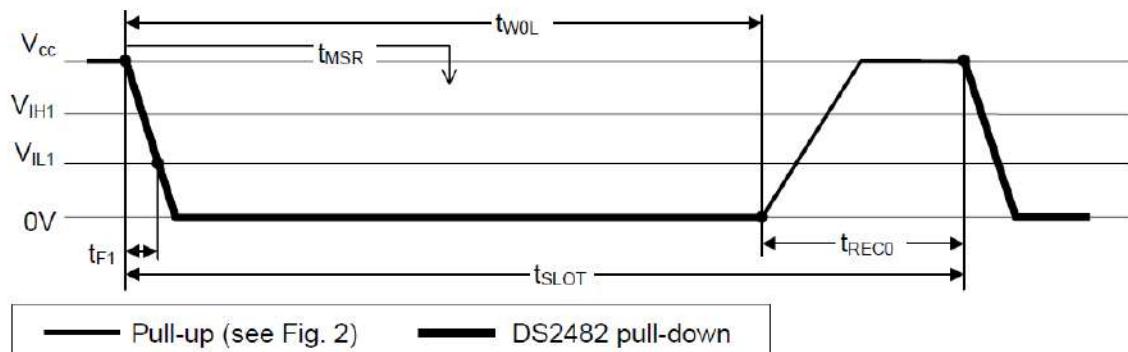
<b>Command Code</b>	87h
<b>Command Parameter</b>	Bit Byte
<b>Description</b>	Generates a single 1-Wire time slot with a bit value 'V' as specified by the bit byte at the 1-Wire line. A 'V' value of 0b generates a write-zero time slot (Figure 6), a value of 1b generates a write-one slot, which also functions as a read-data time slot (Figure 7). In either case the logic level at the 1-Wire line is tested at $t_{MSR}$ and SBR is updated.
<b>Typical Use</b>	To perform single bit writes or reads at the 1-Wire line when single bit communication is necessary (the exception).
<b>Restriction</b>	1-Wire activity must have ended before the DS2482 can process this command.
<b>Error Response</b>	Command code and bit byte are not acknowledged if 1WB = 1 at the time the command code is received and the command is ignored.
<b>Command Duration</b>	$t_{SLOT}$ + maximum 262.5ns, counted from the falling SCL edge of the first bit (MS bit) of the bit byte.
<b>1-Wire Activity</b>	Begins maximum 262.5ns after the falling SCL edge of the MS bit of the bit byte.
<b>Read Pointer Position</b>	Status register (for busy polling and data reading)
<b>Status Bits Affected</b>	1WB (set to 1 for $t_{SLOT}$ ) SBR is updated at $t_{MSR}$ DIR (may change its state)
<b>Configuration Bits Affected</b>	1WS, APU, SPU apply

#### Bit Allocation in the Bit Byte

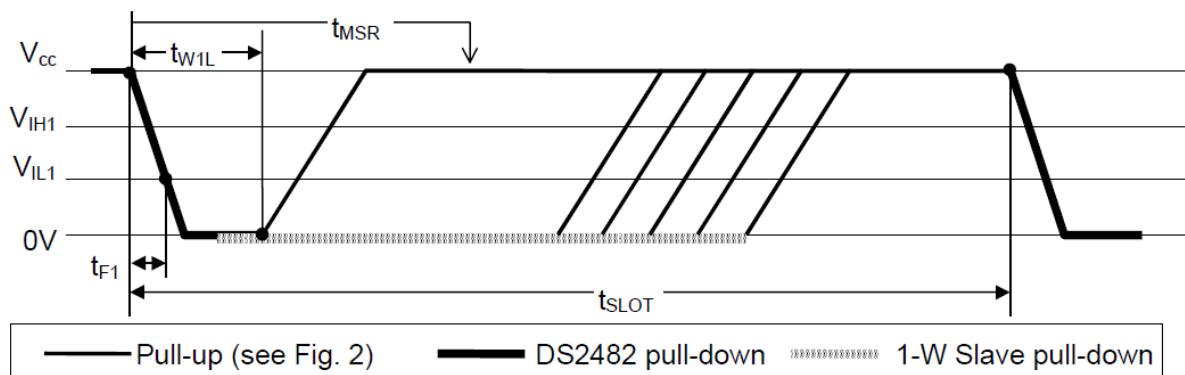
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
V	x	x	x	x	x	x	x

x = don't care

#### 9.6.2.7.1. Timing pour écriture bit à 0



#### 9.6.2.7.2. Timing écriture bit à 1 et lecture data



### 9.6.2.8. COMMAND, 1-WIRE WRITE BYTE

#### 1-Wire Write Byte

<b>Command Code</b>	A5h
<b>Command Parameter</b>	Data Byte
<b>Description</b>	Writes single data byte to the 1-Wire line.
<b>Typical Use</b>	To write commands or data to the 1-Wire line; equivalent to executing eight 1-Wire Single Bit commands, but faster due to less I <sup>2</sup> C traffic.
<b>Restriction</b>	1-Wire activity must have ended before the DS2482 can process this command.
<b>Error Response</b>	Command code and data byte are not acknowledged if 1WB = 1 at the time the command code is received and the command will be ignored.
<b>Command Duration</b>	$8 \times t_{SLOT} + \text{maximum } 262.5\text{ns}$ , counted from falling edge of the last bit (LS bit) of the data byte.
<b>1-Wire Activity</b>	Begins maximum 262.5ns after falling SCL edge of the LS bit of the data byte (i.e., before the data byte acknowledge). <b>NOTE:</b> The bit order on the I <sup>2</sup> C bus and the 1-Wire line is different. (1-Wire: LS-bit first; I <sup>2</sup> C: MS-bit first) Therefore, 1-Wire activity cannot begin before the DS2482 has received the full data byte.
<b>Read Pointer Position</b>	Status register (for busy polling)
<b>Status Bits Affected</b>	1WB (set to 1 for $8 \times t_{SLOT}$ )
<b>Configuration Bits Affected</b>	1WS, SPU, APU apply

### 9.6.2.9. COMMAND, 1-WIRE READ BYTE

#### 1-Wire Read Byte

<b>Command Code</b>	96h
<b>Command Parameter</b>	None
<b>Description</b>	Generates eight read-data time slots on the 1-Wire line and stores result in the Read Data register.
<b>Typical Use</b>	To read data from the 1-Wire line; equivalent to executing eight 1-Wire Single Bit commands with V = 1 (write-1 time slot), but faster due to less I <sup>2</sup> C traffic.
<b>Restriction</b>	1-Wire activity must have ended before the DS2482 can process this command.
<b>Error Response</b>	Command code is not acknowledged if 1WB = 1 at the time the command code is received and the command is ignored.
<b>Command Duration</b>	$8 \times t_{SLOT} + \text{maximum } 262.5\text{ns}$ , counted from the falling SCL edge of the command code acknowledge bit.
<b>1-Wire Activity</b>	Begins maximum 262.5ns after the falling SCL edge of the command code acknowledge bit.
<b>Read Pointer Position</b>	Status register (for busy polling) <b>NOTE:</b> To read the data byte received from the 1-Wire line, issue the Set Read Pointer command and select the Read Data register. Then access the DS2482 in read mode.
<b>Status Bits Affected</b>	1WB (set to 1 for $8 \times t_{SLOT}$ )
<b>Configuration Bits Affected</b>	1WS, APU apply

### 9.6.2.10. COMMAND, 1-WIRE TRIPLET

#### 1-Wire Triplet

<b>Command Code</b>	78h
<b>Command Parameter</b>	Direction Byte
<b>Description</b>	<p>Generates three time slots, two read time slots and one write time slot at the 1-Wire line. The type of write time slot depends on the result of the read time slots and the direction byte.</p> <p>The direction byte determines the type of write time slot if both read time slots are 0 (a typical case). In this case the DS2482 generates a write 1-time slot if V = 1 and a write-0 time slot if V = 0.</p> <p>If the read time slots are 0 and 1, there follows a write-0 time slot.</p> <p>If the read time slots are 1 and 0, there follows a write-1 time slot.</p> <p>If the read time slots are both 1 (error case), the subsequent write time slot is a write 1.</p>
<b>Typical Use</b>	To perform a 1-Wire Search ROM sequence; a full sequence requires this command to be executed 64 times to identify and address one device.
<b>Restriction</b>	1-Wire activity must have ended before the DS2482 can process this command.
<b>Error Response</b>	Command code and direction byte is not acknowledged if 1WB = 1 at the time the command code is received and the command will be ignored.
<b>Command Duration</b>	$3 \times t_{SLOT} + \text{maximum } 262.5\text{ns}$ , counted from the falling SCL edge of the first bit (MS bit) of the direction byte.
<b>1-Wire Activity</b>	Begins maximum 262.5ns after the falling SCL edge of the MS bit of the direction byte.
<b>Read Pointer Position</b>	Status register (for busy polling)
<b>Status Bits Affected</b>	<p>1WB (set to 1 for <math>3 \times t_{SLOT}</math>)</p> <p>SBR is updated at the first <math>t_{MSR}</math></p> <p>TSB and DIR are updated at the second <math>t_{MSR}</math> (i. e., at <math>t_{SLOT} + t_{MSR}</math>)</p>
<b>Configuration Bits Affected</b>	1WS, APU apply

### 9.6.3. UTILISATION DU DS2482 EN RELATION AVEC DS18B20

Pour arriver à communiquer avec le DS18B20, il faut déterminer le principe de communication et utiliser les commandes correspondant aux besoins.

#### 9.6.3.1. PRINCIPE DE LA COMMUNICATION AVEC LE DS18B20

Voici un extrait du document "Le bus 1-wire" écrit par D. Menesplier de l'ENAC :

Les DS18B20 possèdent un code unique sur 64 bits comme tous les circuits 1-Wire. Le code famille est h"28", suivi de 6 octets propres au circuit et d'un octet de CRC.

La détection de présence de ce circuit se fait en envoyant le pulse de Reset, qui est un état bas pendant au moins 480 µs. Quand un circuit DS 18B20 est présent sur le bus, il le signale en maintenant le bus à l'état bas pendant 60 à 240 µs.

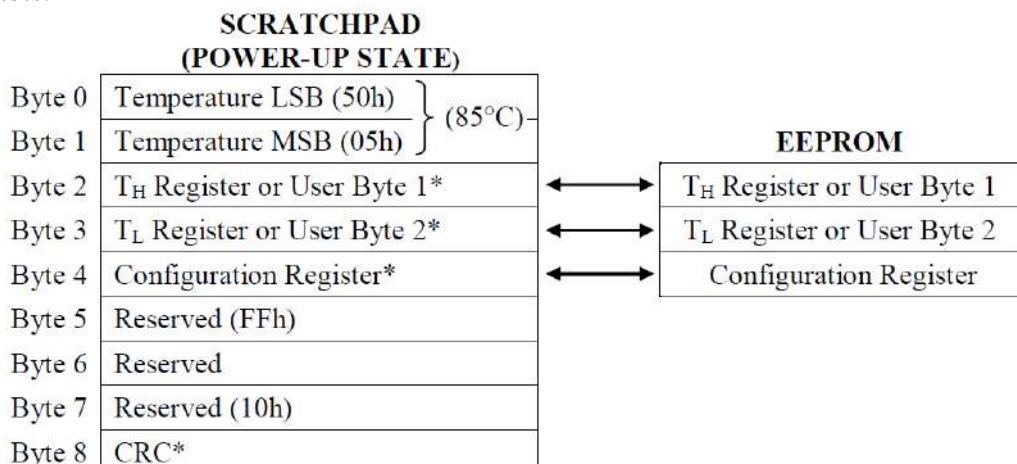
Toute transaction avec un tel circuit doit démarrer par un pulse de Reset suivie de l'envoi d'une commande ROM. On pourra après envoyer une commande de fonction propre à ce type de circuit.

Si le circuit est seul sur le bus 1-Wire, la commande ROM peut être l'appel général SKIP ROM = 0xCC. Si ce n'est pas le cas, il faudra connaître les 64 bits propres au circuit que l'on veut atteindre et utiliser la commande MATCH ROM = 0x55 suivie des 8 octets du code.

Une recherche préalable des 8 octets de code sera faite par la commande READ ROM = 0x33 si le circuit est seul ou bien par SEARCH ROM = 0xF0 s'il y a plusieurs circuits sur le bus.

#### 9.6.3.1.1. Organisation de la mémoire interne

Elle est constituée d'une zone RAM de 9 octets et d'une zone EEPROM non volatile de 3 octets.



\*Power-up state depends on value(s) stored in EEPROM.

### 9.6.3.2. COMMANDES DU DS18B20

Après avoir envoyé une commande ROM pour adresser un DS18B20 esclave, le maître doit envoyer un code de commande. Voici quelques commandes qui devraient suffire pour lire la température.

### 9.6.3.3. Ds18B20, CONVERT

Code 0x44. Cette commande lance la conversion de température. Le résultat est rangé dans les 2 octets LSB et MSB. Le temps de conversion dépend de la résolution choisie. Le maître doit interroger le DS18B20 qui répond par un bit à "0" tant que la conversion n'est pas terminée. Quand l'opération est terminée, l'esclave répond par un bit à "1".

### 9.6.3.4. Ds18B20, READ SCRATCHPAD

Code 0xBE. Les 9 octets de la RAM sont envoyés vers le maître. L'esclave commence par le bit 0 du premier octet et transmet ainsi les 9 octets de sa RAM.

Le maître peut interrompre à tout moment la lecture en faisant un Reset.

### 9.6.3.5. Ds18B20, RÉSUMÉ DES COMMANDES

Voici le résumé des commandes tiré directement du datasheet du DS18B20 :

COMMAND	DESCRIPTION	PROTOCOL	1-WIRE BUS ACTIVITY AFTER COMMAND IS ISSUED	NOTES
<b>TEMPERATURE CONVERSION COMMANDS</b>				
Convert T	Initiates temperature conversion.	44h	DS18B20 transmits conversion status to master (not applicable for parasite-powered DS18B20s).	1
<b>MEMORY COMMANDS</b>				
Read Scratchpad	Reads the entire scratchpad including the CRC byte.	BEh	DS18B20 transmits up to 9 data bytes to master.	2
Write Scratchpad	Writes data into scratchpad bytes 2, 3, and 4 ( $T_H$ , $T_L$ , and configuration registers).	4Eh	Master transmits 3 data bytes to DS18B20.	3
Copy Scratchpad	Copies $T_H$ , $T_L$ , and configuration register data from the scratchpad to EEPROM.	48h	None	1
Recall $E^2$	Recalls $T_H$ , $T_L$ , and configuration register data from EEPROM to the scratchpad.	B8h	DS18B20 transmits recall status to master.	
Read Power Supply	Signals DS18B20 power supply mode to the master.	B4h	DS18B20 transmits supply status to master.	

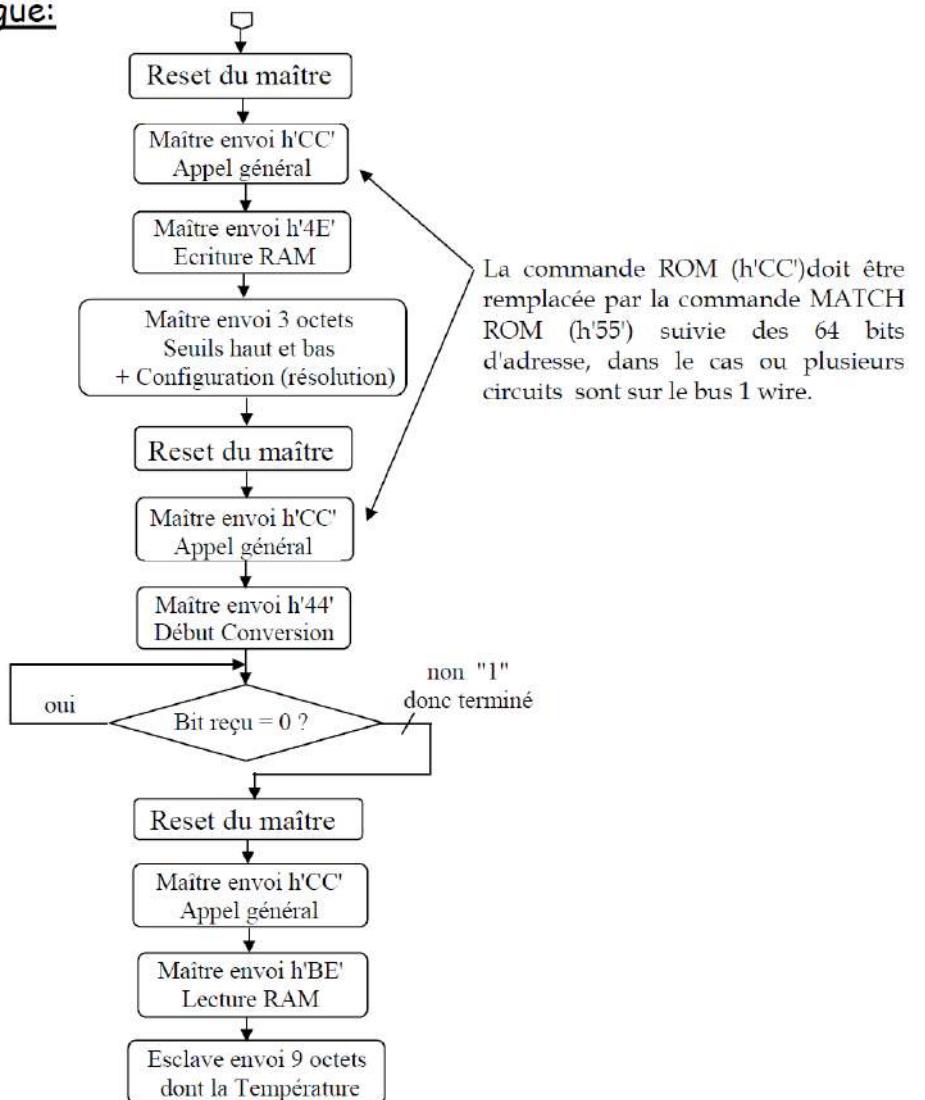
**Note 1:** For parasite-powered DS18B20s, the master must enable a strong pullup on the 1-Wire bus during temperature conversions and copies from the scratchpad to EEPROM. No other bus activity may take place during this time.

**Note 2:** The master can interrupt the transmission of data at any time by issuing a reset.

**Note 3:** All three bytes must be written before a reset is issued.

### 9.6.3.6. EXEMPLE DE DIALOGUE

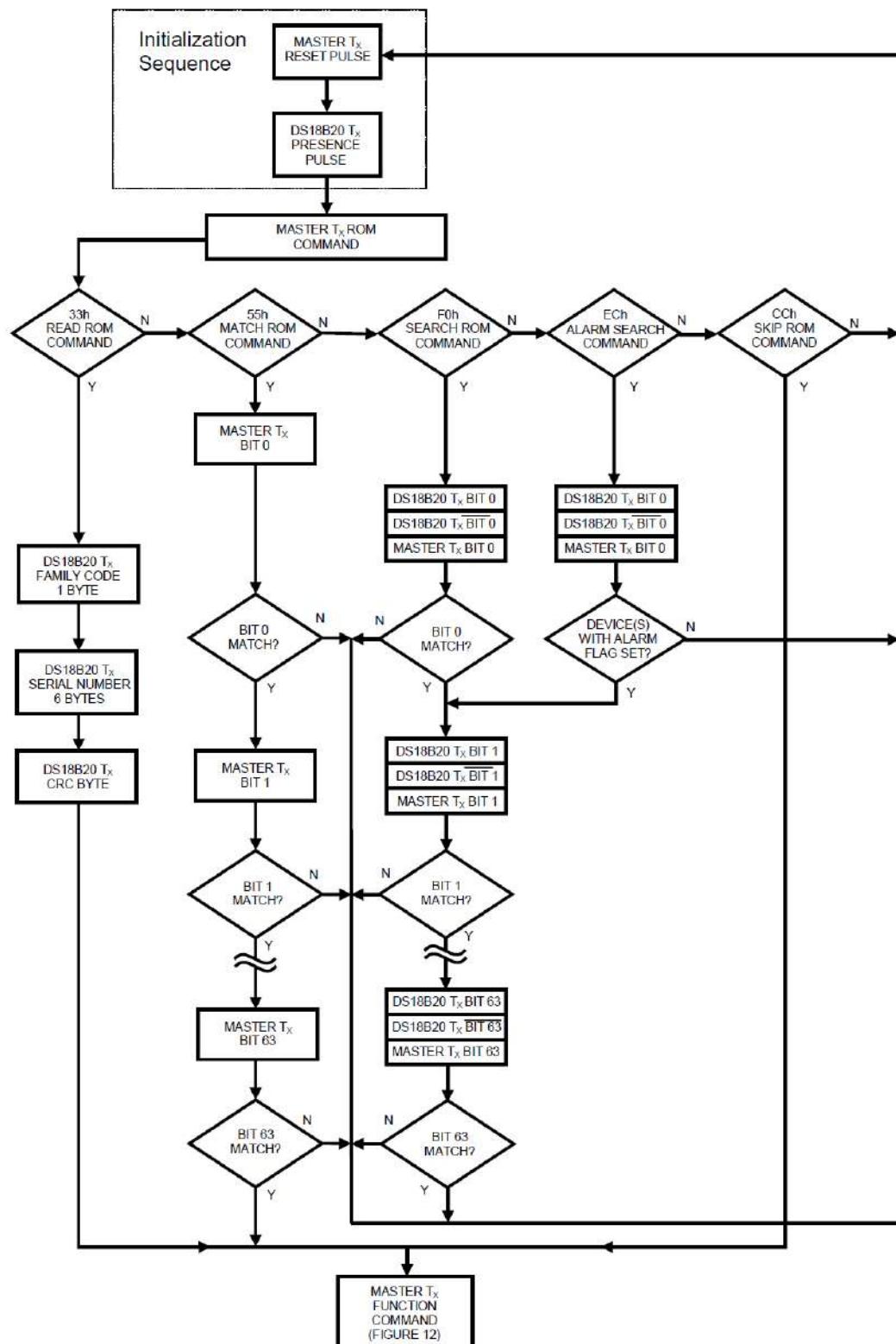
#### Exemple de dialogue:



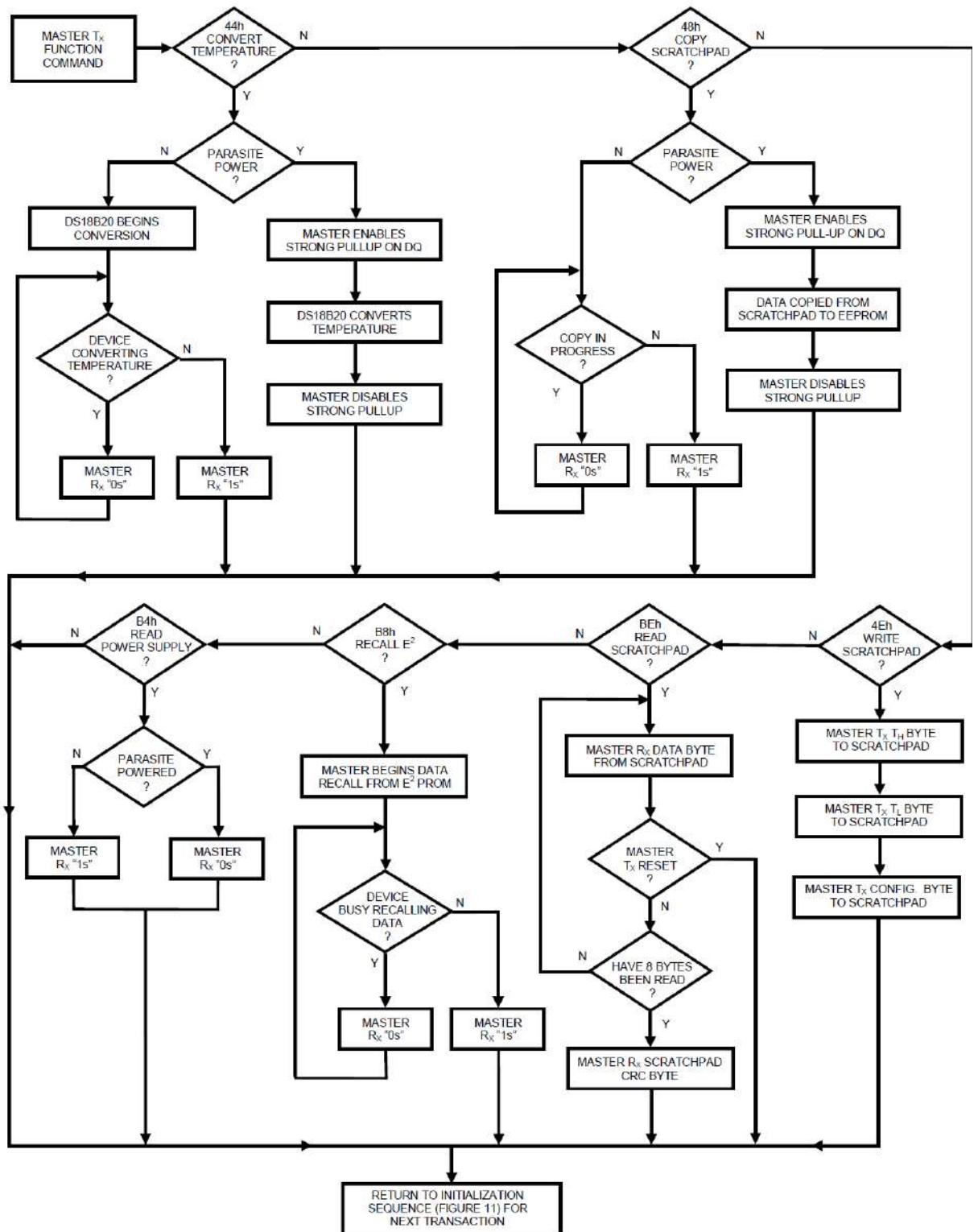
Une fois la configuration effectuée, il suffit de répéter l'envoi de la demande de conversion et la lecture de la RAM.

### 9.6.3.7. SÉQUENCE COMMANDE ROM

Flowchart issu du datasheet du DS18B20 :



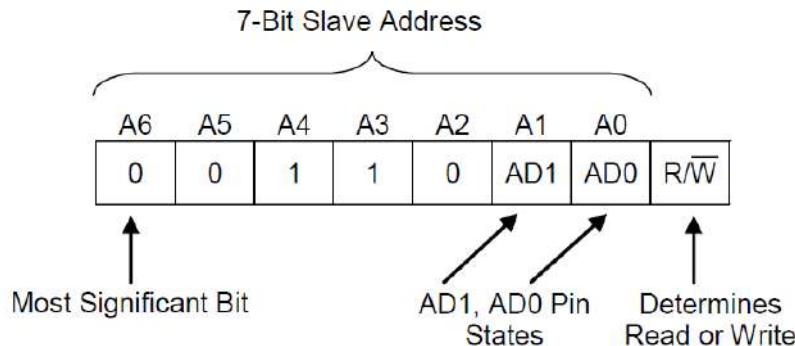
### 9.6.3.8. SÉQUENCE TRAITEMENT DES COMMANDES



Pour exécuter une 2<sup>ème</sup> commande, il est nécessaire de recommencer entièrement la séquence.

#### **9.6.4. COMMUNICATION AVEC LE COMPOSANT DS18B20**

#### **9.6.4.1. DÉFINITIONS POUR LE DS2482**



```

#define ds2482_100_write 0x30 // addr ds2482-100 R/_W = 0
#define ds2482_100_read   0x31 // addr ds2482-100 R/_W = 1

// Reset
// code de reset du ds2482-100
#define ds2482_100_reset_code 0xf0
// envoi d'un reset/presence sur la ligne one wire
#define ds2482_100_one_wire_device_reset 0xb4

// Configuration
// code de commande pour écrire le byte de configuration
#define ds2482_100_write_config_code 0xd2

#define ds2482_100_config_byte_spu 0b10100101
// 1ws = 1 wire speed = standard (low) = 0
// SPU = strong pullup = 1
// PPM = presence pulse masking = 0
// APU = active pullup = 1
#define ds2482_100_config_byte_nospu 0b11100001
// 1ws = 1 wire speed = standard (low) = 0
// SPU = strong pullup = 0
// PPM = presence pulse masking = 0
// APU = active pullup = 1

// set read pointer
// code de commande pour positionner le pointeur de lecture
#define ds2482_100_set_read_pointer_code 0xe1
// positionne le pointeur de lecture sur le status register
#define ds2482_100_set_read_pointer_to_status_register 0xf0

// Positionne pointeur lecture sur le read data register
#define ds2482_100_set_read_pointer_to_read_data_register 0xe1
// positionne pointeur lecture sur le channel selection register
#define ds2482_100_set_read_pointer_to_channel_selection_register 0xd2
// positionne le pointeur de lecture sur le configuration register
#define ds2482_100_set_read_pointer_to_configuration_register 0xf0

```

```
// write byte to 1 wire
// code de commande pour écrire un byte sur le ds2482-100
#define ds2482_100_1wire_write_byte_code 0xa5

// read byte to 1 wire
// code de commande pour lire un byte sur le ds2482-100
// résultat de la lecture dans le read data register
#define ds2482_100_1wire_read_byte_code 0x96

// write (and read) single bit
// code de commande pour écrire un bit slot sur la ligne
// on passe en paramètre la valeur du bit dans le msb
// et on retrouve la valeur du bit (soit écrite soit lue
// par le ds2482) dans le bit SBR du byte de status
#define ds2482_100_1wire_single_bit_code 0x87
// code de commande pour réaliser la séquence complète
#define ds2482_100_1wire_triplet_code 0x78
// valeur pour un single bit à 0
#define ds2482_100_1wire_single_bit_0 0x00
// valeur pour un single bit à 1
#define ds2482_100_1wire_single_bit_1 0x80
```

#### 9.6.4.2. DÉFINITIONS 1-WIRE ET DS18B20

```
// Définitions one wire pour les commandes
#define one_wire_read_rom_command_code 0x33
#define one_wire_match_rom_command_code 0x55
#define one_wire_search_rom_command_code 0xf0
#define one_wire_skip_rom_command_code 0xcc
#define one_wire_no_command_code 0xff

// Définitions pour les sondes de température ds18b20
#define ds18b20_family_code 0x28
#define ds18b20_convert_T_command_code 0x44
#define ds18b20_copy_scratchpad_command_code 0x48
#define ds18b20_write_scratchpad_command_code 0x4e
#define ds18b20_read_scratchpad_command_code 0xbe
#define ds18b20_th_value 0x0
#define ds18b20_t1_value 0x0
// 12 bits résolution
#define ds18b20_config_byte 0b01111111
#define ds18b20_ok_status 0 // tout est ok
#define ds18b20_shorted 1 // ligne one wire court-circuitée
#define ds18b20_missing 2 // pas de capteur
// erreur de crc lors de la lecture de la température
#define ds18b20_temp_reading_crc_error 3
// condition de reset 85.0 °C
#define ds18b20_reset_condition 4
#define ds18b20_ident_code_reading_crc_error 5
// erreur de crc lors de la lecture du code d'identification

// définition de la température par défaut au power-up
// cette valeur n'est transmise qu'avant le premier accès à un canal
de sonde
#define power_up_temp_value (200) // 20°C
```

#### 9.6.4.3. STRUCTURES ET VARIABLES

```
// pour ds2482-100
byte ds2482_100_status;

// Premièrement une structure pour le scratchpad du DS18b20
typedef struct
{
    byte temp_lsb,temp_msb,r2,r3,r4,r5,r6,r7,crc;
} t_ds18b20_scratchpad;

// Quatrièmement une union pour pouvoir accéder à 16bits
// soit en 2 bytes, soit en 16bits, soit en 16 bits signés
typedef union {
    struct
    {
        byte lsb,msb;
    } octet;
    uint16_t word;
    int16_t signed_word;
} t_16bits;

t_16bits new_temp;

// Structure pour la gestion d'un capteur
typedef struct
{
    byte Ds18b20Ident[8];
    t_ds18b20_scratchpad ds18b20_scratchpad;
    byte Ds18b20_status;
    t_16bits Temp16;
    t_16bits Last_temp16;
} t_sensor;
//
// Nous allons avoir besoin d'une de ces structure
t_sensor sensor;
```

**9.6.4.4. LES FONCTIONS DE COMMUNICATIONS 1-WIRE****9.6.4.4.1. Write one byte**

```
void ds2482_100_write_one_byte(uint8_t write_and_dest_chip,
                                uint8_t byte0 )
{
    i2c_start();
    i2c_write(write_and_dest_chip);
    i2c_write(byte0);
    i2c_stop();
}
```

**9.6.4.4.2. Write two bytes**

```
void ds2482_100_write_two_bytes(
                                uint8_t write_and_dest_chip,
                                uint8_t byte1, uint8_t byte2) {
    i2c_start();
    i2c_write(write_and_dest_chip);
    i2c_write(byte1);
    i2c_write(byte2);
    i2c_stop();
}
```

**9.6.4.4.3. Write three bytes**

```
void ds2482_100_write_three_bytes
                                (uint8_t write_and_dest_chip,
                                 uint8_t byte1, uint8_t byte2, uint8_t byte3)
{
    i2c_start();
    i2c_write(write_and_dest_chip);
    i2c_write(byte1);
    i2c_write(byte2);
    i2c_write(byte3);
    i2c_stop();
}
```

**9.6.4.4.4. Read one byte**

```
byte ds2482_100_read_one_wire_byte
                                (uint8_t read_and_source_chip){
    byte ret_byte;
    // read byte
    i2c_start();
    i2c_write(read_and_source_chip);
    ret_byte = i2c_read(0); // no ack
    i2c_stop();
    return ret_byte;
}
```

#### 9.6.4.4.5. Read status

```
void one_wire_read_status(uint8_t read_and_source_chip) {
    // lecture du status de ds2482-800 ?
    i2c_start();
    i2c_write(read_and_source_chip);
    ds2482_100_status = i2c_read(0); // no ack
    i2c_stop();
}
```

#### 9.6.4.4.6. Channel Busy

```
bool one_wire_channel_busy(uint8_t read_and_source_chip)
{
    // Transmission en cours sur la ligne one wire ?
    bool fin;

    one_wire_read_status(read_and_source_chip);
    fin = ((ds2482_100_status & 0x01) > 0);
    // retourne un 1 si une transmission est en cours
    return fin;
}
```

#### 9.6.4.4.7. End Xmit

Attend que la ligne 1-Wire se libère. Cette fonction est à utiliser pour attendre la fin d'exécution d'une commande 1-Wire.

```
void one_wire_end_xmit(uint8_t read_and_source_chip)
{
    do {
        } while (one_wire_channel_busy(read_and_source_chip));
}
```

#### 9.6.4.5. LA FONCTION DE LECTURE DE LA TEMPÉRATURE

Cette fonction fournit la température et une indication sur la situation du composant OneWire.

⊗ Il est nécessaire d'utiliser un passage par référence car lorsque l'on retourne une valeur float dans une variable globale, on obtient une valeur fantaisiste.

```
void ReadDS18b20(int8_t *Status, float *pTemp)
{
    float Temp = 21.5;
    uint8_t nb_bytes;
    // Reset du DS2482
    ds2482_100_write_one_byte(ds2482_100_write,
                               ds2482_100_reset_code);
    one_wire_end_xmit(ds2482_100_read_address);

    // One-Wire reset/presence pulse
    ds2482_100_write_one_byte(ds2482_100_write,
                               ds2482_100_one_wire_device_reset);
    one_wire_end_xmit(ds2482_100_read_address);

    // Lecture et analyse du Status One-Wire
    one_wire_read_status(ds2482_100_read);
    switch (ds2482_100_status & 0x06) { // keep SD & PPD
        case 0: // no sensor
            sensor.ds18b20_status = ds18b20_missing;
            // no sensor on line (ds18b20)
            break;
        case 2: // normal operation,
            // not shorted and presence pulse ok
            sensor.ds18b20_status = ds18b20_ok_status;
            break;
        case 4:
        case 6: // line shorted
            sensor.ds18b20_status = ds18b20_shorted;
            // short circuit (ds18b20) on line
            break;
    }
    // end switch DS2482_100_status

    *status = ds2482_100_status & 0x06;

    if (*status != 2) {

        Goto ExitReadDs18b20;
    }

    // Envoi commande 0xCC SKIP ROM
    ds2482_100_write_two_bytes(ds2482_100_write_address,
                               ds2482_100_1wire_write_byte_code,
                               one_wire_skip_rom_command_code);
    one_wire_end_xmit(ds2482_100_read_address);
}
```

```
// Force strong Pullup
// (Nécessaire si Vcc non utilisé)
ds2482_100_write_two_bytes(ds2482_100_write_address,
                           ds2482_100_write_config_code,
                           ds2482_100_config_byte_spu);

// Envoi commande 0X44 début conversion T
ds2482_100_write_two_bytes(ds2482_100_write_address,
                           ds2482_100_1wire_write_byte_code,
                           ds18b20_convert_T_command_code);
one_wire_end_xmit(ds2482_100_read_address);

// Attente bit reçu = 0
// L'attente n'est possible que si circuit alimenté
// BSP_LEDOn(BSP_LED_7); // provisoire pour observation
delay_ms(750); // par mesure 720 ms
// BSP_LEDOff(BSP_LED_7); // provisoire pour observation

// Supprime strong Pullup
ds2482_100_write_two_bytes(ds2482_100_write_address,
                           ds2482_100_write_config_code,
                           ds2482_100_config_byte_nospu);

// 2ème one wire reset/presence pulse
// -----
// Remarque il n'est pas nécessaire de refaire
// le reset du DS2482-100
ds2482_100_write_one_byte(ds2482_100_write,
                           ds2482_100_one_wire_device_reset);
one_wire_end_xmit(ds2482_100_read_address);

// Envoi commande 0xCC SKIP ROM
ds2482_100_write_two_bytes(ds2482_100_write_address,
                           ds2482_100_1wire_write_byte_code,
                           one_wire_skip_rom_command_code);
one_wire_end_xmit(ds2482_100_read_address);

// Envoi commande 0xBE read scratchpad
// Lecture de 8 bytes de données et un crc, soit 9 bytes
ds2482_100_write_two_bytes(ds2482_100_write,
                           ds2482_100_1wire_write_byte_code,
                           ds18b20_read_scratchpad_command_code);
one_wire_end_xmit(ds2482_100_read_address);
ds18b20_scratchpad_ptr =
    &sensor.ds18b20_scratchpad.temp_lsb;
```

```
// principe séquencement des actions de lecture
// 1) demander la lecture d'un byte au ds18b20
// 2) lire le byte

for (nb_bytes = 0 ; nb_bytes < 9 ; nb_bytes++) {

    // Envoi commande de lecture d'un byte
    ds2482_100_write_one_byte(ds2482_100_write,
                               ds2482_100_1wire_read_byte_code);
    one_wire_end_xmit(ds2482_100_read_address);

    // Effectue la lecture d'un byte
    // Pointeur DS2482 sur Read data register
    // Lecture dans le DS2482
    ds2482_100_write_two_bytes(ds2482_100_write,
                                ds2482_100_set_read_pointer_code,
                                ds2482_100_set_read_pointer_to_read_data_register);

    *ds18b20_scratchpad_ptr =
    ds2482_100_read_one_wire_byte(ds2482_100_read_address);
    // Remettre pointeur sur le Status
    // pour attendre fin action
    ds2482_100_write_two_bytes(ds2482_100_write,
                                ds2482_100_set_read_pointer_code,
                                ds2482_100_set_read_pointer_to_status_register);
    one_wire_end_xmit(ds2482_100_read_address);
    // Pointe sur le byte suivant
    ds18b20_scratchpad_ptr++;

} // end for
// Expression de la température en degré
new_temp.octet.msb = sensor.ds18b20_scratchpad.temp_msb;
new_temp.octet.lsb = sensor.ds18b20_scratchpad.temp_lsb;
temp = new_temp.signed_word * 0.0625;

ExitReadDs18b20:
    *pTemp = temp;
}
```

On constate que la séquence déjà relativement compliquée au niveau du DS18B20, est rendue plus compliquée par les actions nécessaires sur le DS2482-100. La lecture se complique par le besoin de changer le pointeur de lecture entre donnée et statut.

La séquence présentée est le scénario le plus simple. Dès que l'on veut vérifier la présence d'un composant avec identification, cela devient plus compliqué.

Le délai de 750 ms pose problème, il est nécessaire d'espacer les appels de la fonction sans descendre en dessous d'une seconde.

Dans une situation où il y a plusieurs composants OneWire sur la ligne, le traitement devient assez conséquent.

#### 9.6.4.6. OBSERVATION DE LA TRANSACTION

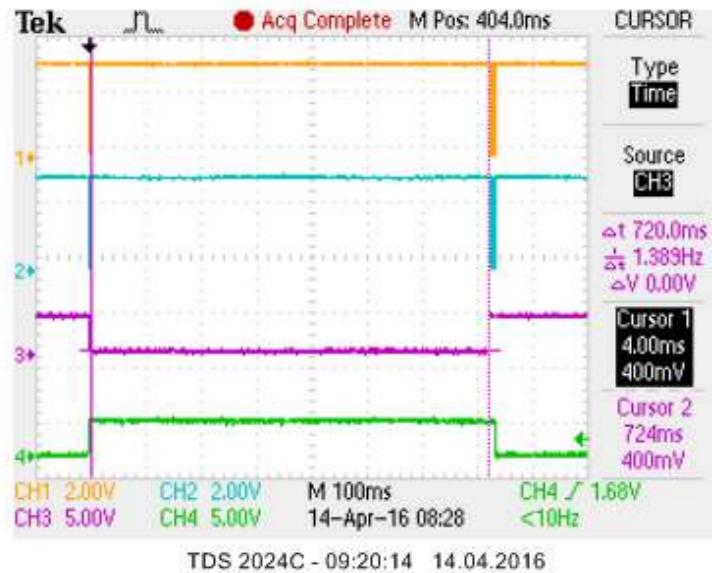
L'action sur la LED\_6 marque le début et la fin de la fonction **ReadDs18b20**.

Canal 1 : SCK  
I2C-SCK / SCL2/RA2  
PORTAbits.RA2 / pin 58

Canal 2 : SDA  
I2C-SDA / SDa2/RA3  
PORTAbits.RA3 / pin 59

Canal 3 : LED\_7  
marqueur durée du délai  
dans la fonction  
ReadDS18B20

Canal 4 : LED\_6  
marqueur durée de la  
fonction ReadDS18B20.



On remarque de l'activité sur le bus I2C au début et à la fin du délai prévu de 750 ms qui fait en réalité 720 ms.

#### 9.6.4.7. DÉTAILS DU DÉBUT LA TRANSACTION

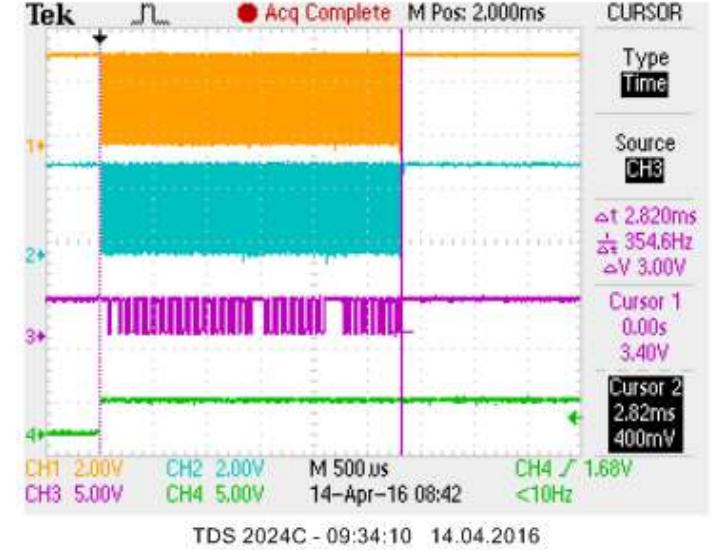
L'action sur la LED\_6 marque le début et la fin de la fonction **ReadDS18B20**. La LED\_5 est activée dans la fonction **i2c\_read**.

Canal 1 : SCK  
I2C-SCK / SCL2/RA2  
PORTAbits.RA2 / pin 58

Canal 2 : SDA  
I2C-SDA / SDa2/RA3  
PORTAbits.RA3 / pin 59

Canal 3 : LED\_5  
marqueur dans la fonction  
i2c\_read

Canal 4 : LED\_6  
marqueur durée de la  
fonction ReadDS18B20.



La durée du début de la transaction est de 2.8 ms.

#### 9.6.4.8. DÉTAILS DE LA FIN DE LA TRANSACTION

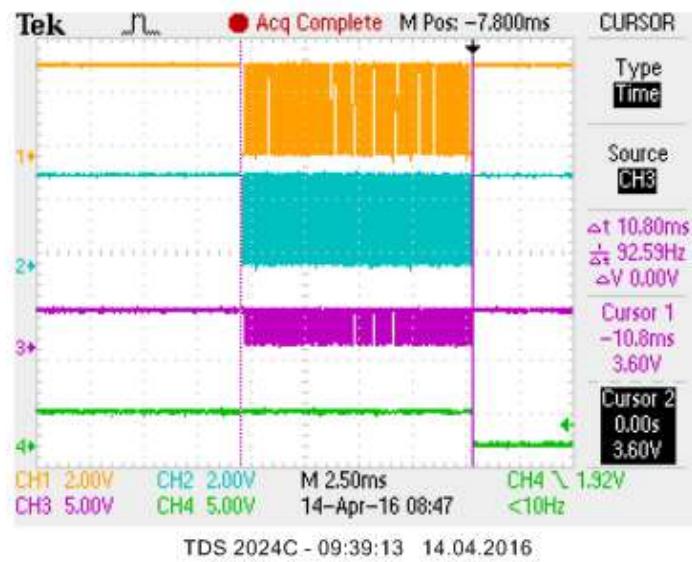
L'action sur la LED\_6 marque le début et la fin de la fonction ReadDS18B20. La LED\_5 est activée dans la fonction **i2c\_read**.

Canal 1 : SCK  
I2C-SCK / SCL2/RA2  
PORTAbits.RA2 / pin 58

Canal 2 : SDA  
I2C-SDA / SDa2/RA3  
PORTAbits.RA3 / pin 59

Canal 3 : LED\_5  
marqueur dans la fonction  
i2c\_read

Canal 4 : LED\_6  
marqueur autour de la  
fonction ReadDS18B20.



La durée de la fin de la transaction est de 10.8 ms.

#### 9.6.4.9. CONTENU DU FICHIER APP.C DE L'APPLICATION UTILISÉE

Remarque : L'application est activée toutes les 1500 ms.

```
#include "app.h"
#include "Mc32DriverLcd.h"
#include "Mc32gestI2cLM92.h"
#include "Mc32_DS18b20.h"
#include "Mc32Delays.h"

// Variables
APP_DATA appData;
int16_t APP_RawTemp = 225;
float APP_TempLm92 = 21.7;
float APP_DS18B20Temp = 17.1;
uint8_t OneWireStatus = 2;
```

```
void APP_Tasks ( void )
{
    /* Check the application's current state. */
    switch ( appData.state )
    {
        /* Application's initial state. */
        case APP_STATE_INIT:
        {
            lcd_init();
            lcd_bl_on();

            // Init
            I2C_InitLM92();
            init_oneWire();
            printf_lcd("TEST I2C Chap 9      ");
            lcd_gotoxy(1,2);
            printf_lcd("C. Huber 12.04.2016");
            DRV_TMR0_Start(); // depuis Harmony 1.06
            appData.state = APP_STATE_WAIT;
            break;
        }

        case APP_STATE_WAIT :
            // nothing to do
            break;

        case APP_STATE_SERVICE_TASKS:
            BSP_LEDToggle(BSP_LED_2);

            // APP_RawTemp = I2C_ReadRawTempLM92();
            APP_RawTemp = I2C_WriteCfgReadRawTempLM92();
            LM92_ConvRawToDeg( APP_RawTemp, &APP_TempLm92);

            // Affichage temperature du LM92
            lcd_gotoxy(1,3);
            printf_lcd("Temp = %6.1f", APP_TempLm92);
            delay_us(200) ; //pour séparer les actions
            BSP_LEDOff(BSP_LED_6);
            ReadDS18B20(&OneWireStatus, &APP_DS18B20Temp);
            BSP_LEDOn(BSP_LED_6);
    }
}
```

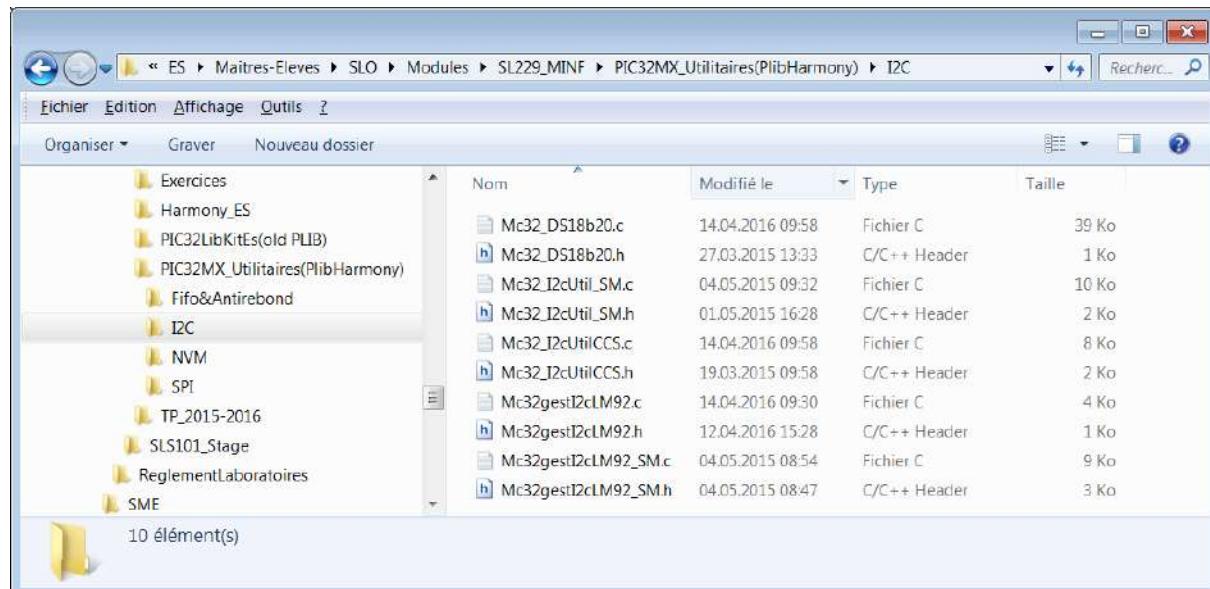
```
lcd_gotoxy( 1, 4);
switch (OneWireStatus) {
    case 0 :
        printf_lcd("Missing DS18B20");
        break;
    case 2 :
        // Ok affiche la température
        printf_lcd( "DS18B20 temp: %5.2f",
                    APP_DS18B20Temp);
        break;
    case 4 :
    case 6 :
        printf_lcd("Line shorted      ");
        break;
} // end switch
appData.state = APP_STATE_WAIT;
break;
/* TODO: implement your application state
machine.*/

/* The default state should never be executed. */
default:
{
    /* TODO: Handle error in application's state
       machine. */
    break;
}
}
```

## 9.7. FICHIERS A DISPOSITION

Les 6 fichiers sont sous :

...\\Maitres-Eleves\\SLO\\Modules\\SL229\_MINF\\PIC32MX\_Utilitaires(PlibHarmony)\\I2C



Remarque : fichiers fournis avec les actions sur les LEDs d'observation mises en commentaire. Les fichiers xxx\_SM seront traités dans le cadre des travaux pratiques.

## 9.8. CONCLUSION

On constate avec satisfaction que les fonctions I2C développées permettent de gérer simplement des composants I2C. Les librairies présentées d'accès aux capteurs de température LM92 ainsi qu'au DS18B20 (via une conversion I2C-OneWire) sont fonctionnelles.

Ce document devrait permettre, en s'inspirant des principes utilisés dans les deux exemples, de s'adapter à la gestion par le bus I2C de différents composants.

Il s'agira à chaque fois d'étudier avec soin la séquence à générer, en tenant compte de l'adresse de base du composant ainsi que de ses particularités. Une étude détaillée des datasheets est indispensable.

L'intégration d'un exemple de gestion d'un capteur OneWire fournit une base pour la gestion de ce type de composant par le biais du DS2482-100.

## **9.9. HISTORIQUE DES VERSIONS**

### **9.9.1. VERSION 1.5 AVRIL 2015**

Version de base pour PIC32MX avec compilateur XC32 et Harmony. La version 1.5 indique l'utilisation de la plib\_i2c de Harmony.

### **9.9.2. VERSION 1.7 AVRIL 2016**

La version 1.7 indique l'utilisation de la plib\_i2c de Harmony V1.06 combiné avec le MPLABX 3.10. Contrôle et adaptation. Toutes les mesures ont été refaites.

### **9.9.3. VERSION 1.7.1 AVRIL 2016**

En page 4 correction de SDL en SCL.

### **9.9.4. VERSION 1.8 AVRIL 2017**

Relecture générale par SCA. Remise en forme. Eclaircissement des problèmes du code d'initialisation généré par Harmony.

### **9.9.5. VERSION 1.9 FEVRIER 2018**

Ajouts documents de référence. Corrections mineures.

### **9.9.6. VERSION 1.91 FEVRIER 2022**

Enlevé références à CCS et PIC18.

**MINF**  
**Programmation des PIC32MX**

# **Chapitre 10**

## **Programmation concurrente**



### **Théorie PIC32MX**

**Christian HUBER (CHR)**  
**Serge CASTOLDI (SCA)**  
**Version 1.91 janvier 2021**



## CONTENU DU CHAPITRE 10

<b>10.</b>	<i>Programmation concurrente (PIC32MX)</i>	<b>10-1</b>
<b>10.1.</b>	<b>Besoin de la programmation concurrente</b>	<b>10-1</b>
<b>10.2.</b>	<b>Les fonctions à machine d'état</b>	<b>10-2</b>
10.2.1.	Gestion d'un mouvement, fonction à machine d'état	10-2
10.2.1.1.	Principe gestion du mouvement, machine d'état	10-3
10.2.2.	Exemple de réalisation de gestion de 3 mouvements	10-4
10.2.2.1.	Fonctions de gestion du moteur	10-4
10.2.2.2.	Fonction d'initialisation des mouvements	10-5
10.2.2.3.	Fonction d'exécution d'un déplacement relatif	10-5
10.2.2.4.	Fonctions simulant le start et stop du moteur	10-5
10.2.2.5.	Fonctions pour la gestion de la position	10-6
10.2.2.6.	Fonction d'exécution des mouvements	10-6
10.2.2.7.	Mise à jour de la position (interruptions externes)	10-7
10.2.2.8.	Appel cyclique des fonctions ExecMove	10-7
10.2.2.9.	Traitements au niveau de l'application	10-8
10.2.2.10.	La fonction APP_ExecMovements	10-10
10.2.2.11.	Les fonctions APP_UpdatePosX	10-10
10.2.2.12.	Test du fonctionnement	10-11
<b>10.3.</b>	<b>Programmation concurrente par Multi-Threading</b>	<b>10-12</b>
10.3.1.	Gestion des 3 mouvements avec multi-processus	10-13
10.3.2.	FreeRTOS avec Harmony	10-14
10.3.2.1.	La fonction vTaskDelay	10-14
10.3.2.2.	La fonction vTaskDelayUntil	10-15
10.3.3.	Ajout RTOS dans la configuration	10-16
10.3.3.1.	Ajout de 3 tâches supplémentaires	10-17
10.3.3.2.	Situation de SYS_Tasks	10-18
10.3.3.3.	Réalisation des tâches	10-19
10.3.4.	Réalisation des tâches moteurs	10-21
10.3.4.1.	Réalisation de APPM1_Tasks	10-21
10.3.4.2.	Réalisation de APPM2_Tasks	10-22
10.3.4.3.	Réalisation de APPM3_Tasks	10-22
10.3.5.	Modification de GesMot	10-23
10.3.6.	Modification de System Interrupt	10-24
10.3.7.	Evolution de App.c	10-25
10.3.8.	Contrôle du fonctionnement	10-28
<b>10.4.</b>	<b>Communication inter-tâches</b>	<b>10-29</b>
10.4.1.	Queue	10-29
10.4.2.	Semaphore	10-30
10.4.2.1.	Binary semaphore	10-30
10.4.2.2.	Counting semaphore	10-31
10.4.3.	Mutex	10-31
10.4.3.1.	Mutex	10-31
10.4.3.2.	Recursive mutex	10-31
<b>10.5.</b>	<b>Conclusion</b>	<b>10-32</b>

<b>10.6. Sources</b>	<b>10-33</b>
<b>10.7. Historique des versions</b>	<b>10-33</b>
10.7.1. Version 1.5 juin 2015	10-33
10.7.2. Version 1.7 juin 2016	10-33
10.7.3. Version 1.71 juin 2016	10-33
10.7.4. Version 1.8 mai 2017	10-33
10.7.5. Version 1.9 février 2018	10-33
10.7.6. Version 1.91 janvier 2021	10-33

## 10. PROGRAMMATION CONCURRENTE (PIC32MX)

Ce chapitre traite de la programmation concurrente au niveau du PIC32MX et des possibilités offertes par Harmony. Deux approches seront étudiées :

- Les fonctions à machine d'état.
- L'usage d'un noyau temps réel (FreeRTOS).

### 10.1. BESOIN DE LA PROGRAMMATION CONCURRENTE

Pour mieux comprendre le besoin de la programmation concurrente et comment la réaliser, voici un exemple pratique qui sera utilisé tout au long de ce chapitre :

Supposons que l'on souhaite gérer trois mouvements d'un robot. Une programmation conventionnelle, donc séquentielle, nous amènera à gérer l'un après l'autre chaque mouvement. Par contre si on désire une exécution simultanée des 3 mouvements, il sera nécessaire de recourir à la programmation concurrente.

Remarque : Il est tout de même possible de traiter avec un seul morceau de programme les 3 mouvements en même temps, mais ce morceau de programme devient délicat à écrire. De plus, il résistera mal aux modifications comme l'ajout d'un 4<sup>ème</sup> axe.

## 10.2. LES FONCTIONS A MACHINE D'ETAT

Les fonctions à machine d'état sont des fonctions appelées cycliquement. Chaque machine possède une variable donnant son état. **En principe, ces fonctions ne doivent pas être bloquantes, ni effectuer des attentes par des boucles ou des fonctions de délais.** Les attentes sont réalisées par comptage des appels ou lecture d'un timer. Les fonctions à machine d'état utilisent généralement une structure switch .. case en utilisant et en faisant évoluer la valeur de la variable de situation (variable d'état).

### 10.2.1. GESTION D'UN MOUVEMENT, FONCTION A MACHINE D'ETAT

Voici tout d'abord la description du principe d'un mouvement :

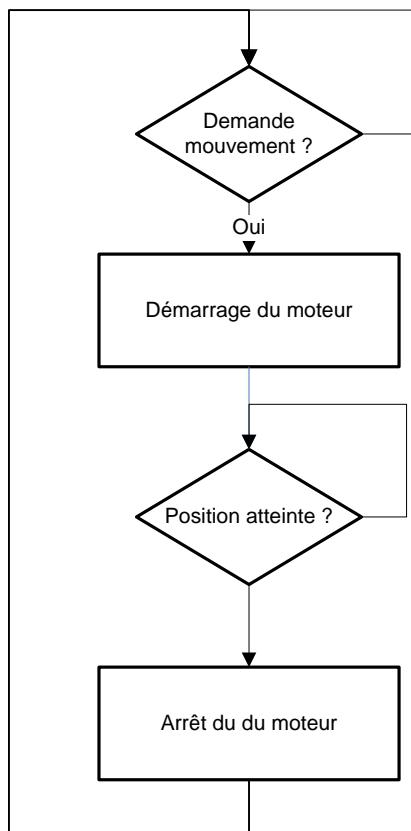


Figure 10-1

### 10.2.1.1. PRINCIPE GESTION DU MOUVEMENT, MACHINE D'ETAT

Le principe de gestion du mouvement devient le suivant avec une machine d'état :

StatusMove correspond au statut du mouvement (0 à l'arrêt, 1 en mouvement).

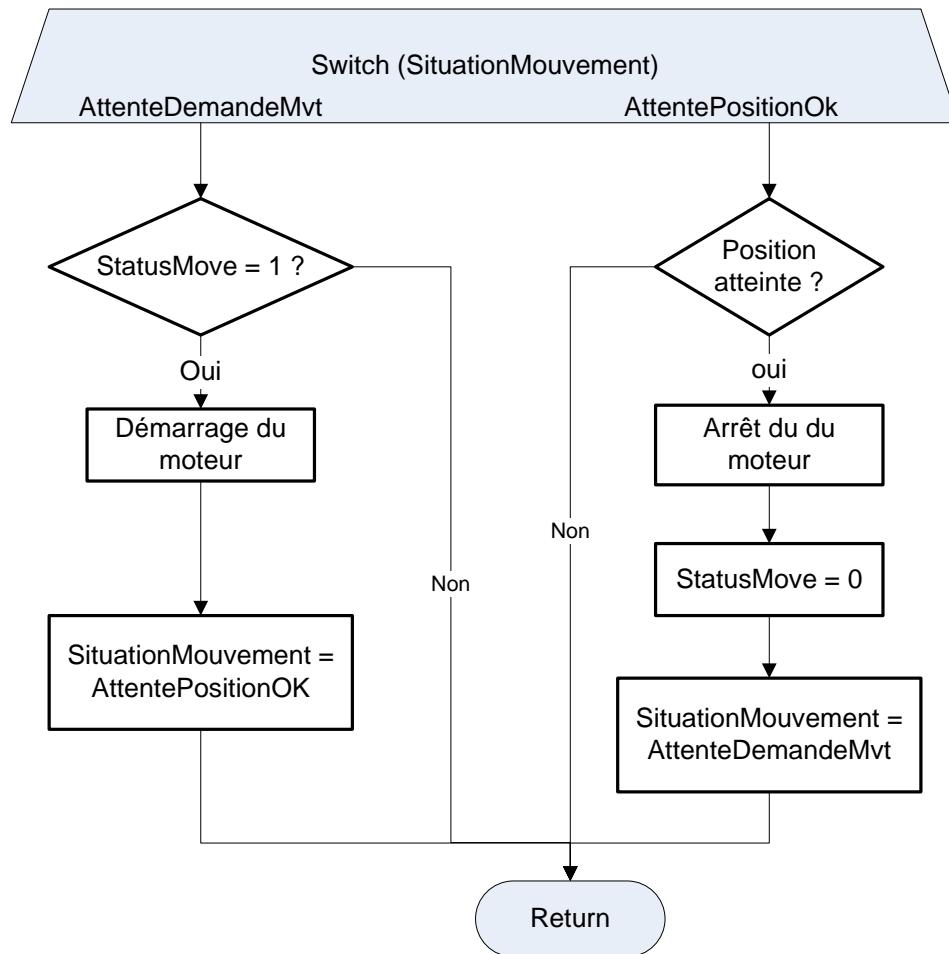


Figure 10-2

Pour démarrer le mouvement, on met à 1 le bit StartMove (action extérieure à la fonction). Il faut attendre StartMove = 1 indication de la fin du mouvement.

Les attentes sont réalisées par des tests qui ressortent de la fonction sans attentes.

## 10.2.2. EXEMPLE DE REALISATION DE GESTION DE 3 MOUVEMENTS

Le programme suivant illustre la gestion de 3 mouvements en réalisant une fonction unique, appelée 3 fois en utilisant un descripteur pour chaque mouvement. L'exemple simplifie la réalité en simulant la rotation du moteur par un signal généré par un OC qui simule les impulsions que fournirait un capteur incrémental. La position du moteur est gérée par comptage dans des interruptions externes. Pour simplifier, on considère de plus que le moteur tourne toujours dans le même sens.

La fonction ExecMove reprend le principe de la Figure 10-2.

### 10.2.2.1. FONCTIONS DE GESTION DU MOTEUR

Les fonctions et définitions permettant de simuler la gestion du moteur sont regroupées dans les fichiers GesMot.h et GesMot.c

#### 10.2.2.1.1. Contenu du fichier GesMot.h

```
#ifndef GESMOT_H
#define GESMOT_H

#include <stdint.h>
#include <stdbool.h>

typedef enum { AttenteDemandeMvt, AttentePositionOK}
    E_MvtSituation;

// Descripteur d'un mouvement
typedef struct {
    int8_t          NoMvt;
    bool            StartToDo;
    E_MvtSituation MvtSituation;
    int32_t         CurPosition;
    int32_t         GoalPosition;
} S_DescrMvt;

// Cette fonction initialise le descripteur de mouvement
void MvtInit(S_DescrMvt *pDescr, int8_t NoMvt);

// Fonction pour exécution d'un mouvement
void RelMove(S_DescrMvt *pDescr, int32_t Pulses);

// Lancement et arrêt du moteur (simulation)
void StartMot(int8_t NoMvt) ;
void StopMot(int8_t NoMvt) ;

// Fonction de gestion des mouvements (appel cyclique)
void ExecMove(S_DescrMvt *pDescr);

// Fonctions pour gestion de la position
bool   IsInPosition( S_DescrMvt *pDescr);
int32_t GetPosition( S_DescrMvt *pDescr);
void   IncPosition (S_DescrMvt *pDescr);

#endif
```

### 10.2.2.2. FONCTION D'INITIALISATION DES MOUVEMENTS

Cette fonction initialise le descripteur de mouvement.

```
void MvtInit(S_DescrMvt *pDescr, int8_t NoMvt)
{
    pDescr->NoMvt = NoMvt;
    pDescr->StartToDo = false;
    pDescr->MvtSituation = AttenteDemandeMvt;
    pDescr->CurPosition = 0;
    pDescr->GoalPosition = 0;
}
```

### 10.2.2.3. FONCTION D'EXECUTION D'UN DEPLACEMENT RELATIF

Cette fonction établit la position de destination dans le descripteur et met à un le bit de demande de mouvement.

```
void RelMove(S_DescrMvt *pDescr, int32_t DeltaPos)
{
    // Prépare le mouvement
    pDescr->GoalPosition = pDescr->CurPosition + DeltaPos;
    pDescr->StartToDo = true;
}
```

### 10.2.2.4. FONCTIONS SIMULANT LE START ET STOP DU MOTEUR

Ces fonctions simulent le lancement et l'arrêt d'un moteur en agissant sur l'OC correspondant.

```
// Lance le moteur (simulation)
// Corresponds à activer l'OC correspondant
void StartMot(int8_t NoMvt)
{
    switch (NoMvt) {
        case 1 : DRV_OC0_Start(); break;
        case 2 : DRV_OC1_Start(); break;
        case 3 : DRV_OC2_Start(); break;
    }
}

// stop le moteur (simulation)
void StopMot(int8_t NoMvt)
{
    switch (NoMvt) {
        case 1 : DRV_OC0_Stop(); break;
        case 2 : DRV_OC1_Stop(); break;
        case 3 : DRV_OC2_Stop(); break;
    }
}
```

### 10.2.2.5. FONCTIONS POUR LA GESTION DE LA POSITION

Ces fonctions sont prévues pour être utilisées depuis l'application. Elles exploitent le contenu du descripteur.

```
bool IsInPosition( S_DescrMvt *pDescr)
{
    bool stat = false;
    if ( pDescr->MvtSituation == AttenteDemandeMvt)
        stat = true;
    return stat;
}

int32_t GetPosition( S_DescrMvt *pDescr)
{
    return pDescr->CurPosition;
}

void IncPosition (S_DescrMvt *pDescr)
{
    pDescr->CurPosition++;
}
```

### 10.2.2.6. FONCTION D'EXECUTION DES MOUVEMENTS

Cette fonction, prévue pour un appel cyclique, assure l'exécution d'un mouvement.

```
void ExecMove (S_DescrMvt *pDescr)
{
    switch (pDescr->MvtSituation) {

        case AttenteDemandeMvt :
            if ( pDescr->StartToDo) {
                pDescr->StartToDo = false;
                // Demande rotation
                StartMot( pDescr->NoMvt) ;
                pDescr->MvtSituation = AttentePositionOK;
            }
            break;

        case AttentePositionOK:
            // Test si position atteinte
            // (la position augmente toujours)
            if (pDescr->CurPosition >= pDescr->GoalPosition) {
                // Position atteinte
                // Stop la rotation
                StopMot( pDescr->NoMvt) ;
                pDescr->MvtSituation = AttenteDemandeMvt;
            }
            break;
    } // end switch
} // end ExecMove
```

### 10.2.2.7. MISE A JOUR DE LA POSITION (INTERRUPTIONS EXTERNES)

Utilisation de fonctions de l'application.

```
void __ISR(_EXTERNAL_1_VECTOR, IPL5AUTO)
          _IntHandlerExternalInterruptInstance0(void)
{
    PLIB_INT_SourceFlagClear(INT_ID_0,
                             INT_SOURCE_EXTERNAL_1);
    APP_UpdatePos1();
}

void __ISR(_EXTERNAL_2_VECTOR, IPL5AUTO)
          _IntHandlerExternalInterruptInstance1(void)
{
    PLIB_INT_SourceFlagClear(INT_ID_0,
                             INT_SOURCE_EXTERNAL_2);
    APP_UpdatePos2();
}

void __ISR(_EXTERNAL_3_VECTOR, IPL5AUTO)
          _IntHandlerExternalInterruptInstance2(void)
{
    PLIB_INT_SourceFlagClear(INT_ID_0,
                             INT_SOURCE_EXTERNAL_3);
    APP_UpdatePos3();
}
```

### 10.2.2.8. APPEL CYCLIQUE DES FONCTIONS EXECMOVE

L'appel cyclique des fonctions ExecMove est placé dans la réponse à l'interruption du timer 1 (cycle 1 ms). La fonction de l'application APP\_ExecMovements effectue l'appel des 3 fonctions ExecMove.

```
// Réponse interruption Timer1 (cycle = 1 ms)
void __ISR(_TIMER_1_VECTOR, ipl4AUTO)
          _IntHandlerDrvTmrInstance0(void)
{
    static int count = 0;

    PLIB_INT_SourceFlagClear(INT_ID_0, INT_SOURCE_TIMER_1);
    count++;
    APP_ExecMovements();
    if (count >= 2000) {
        APP_UpdateState(APP_STATE_SERVICE_TASKS);
        count = 1980;
    }
}
```

### 10.2.2.9. TRAITEMENTS AU NIVEAU DE L'APPLICATION

Voici les différentes actions au niveau de l'application.

#### 10.2.2.9.1. Variable globales

Les trois descripteurs de mouvement et une variable pour la gestion des séquences de mouvements.

```
// Descripteurs de mouvements
S_DescriMvt DescrMot1;
S_DescriMvt DescrMot2;
S_DescriMvt DescrMot3;
// Variable pour gestion séquences de mouvements
int APP_SituationMouvement;
```

#### 10.2.2.9.2. Initialisations

Dans la section case APP\_STATE\_INIT, on effectue l'initialisation des descripteurs et le lancement des timers. Et aussi ce qui est nécessaire à l'affichage sur le lcd.

```
case APP_STATE_INIT:
    lcd_init();
    lcd_bl_on();

    // Init des descripteurs de mouvements
    MvtInit(&DescriMot1, 1);
    MvtInit(&DescriMot2, 2);
    MvtInit(&DescriMot3, 3);
    APP_SituationMouvement = 0;
    DRV_TMR0_Start(); // Start Timer1 (cycle application)
    DRV_TMR1_Start(); // Start Timer2 (base temps des OC)

    printf_lcd("Chap10 prog concu.");
    lcd_gotoxy(1,2);
    printf_lcd("C. Huber 04.06.2015");

    appData.state = APP_STATE_WAIT;
break;
```

#### 10.2.2.9.3. Exécution des séquences de mouvements

Dans la section case APP\_STATE\_SERVICE\_TASKS, on introduit une machine d'état pour réaliser des séquences de mouvements.

```
case APP_STATE_SERVICE_TASKS:
    // Machine d'état traitement des 3 mouvements
    switch (APP_SituationMouvement) {

        case 0 :
            // Lance 1er déplacement d'ensemble
            // Exécution de trois mouvements
            RelMove(&DescriMot1, 3000);
            RelMove(&DescriMot2, 5000);
            RelMove(&DescriMot3, 7000);
            APP_SituationMouvement = 1;
    break;
```

```
        case 1 :
            // Attente fin des 3 mouvements
            if ( IsInPosition(&DescrMot1) &&
                IsInPosition(&DescrMot2) &&
                IsInPosition(&DescrMot3)) {
                APP_SituationMouvement = 2;
                // Affiche la position des 3 moteurs
                lcd_gotoxy(1,3);
                printf_lcd("M1:%06d M2:%06d ",
                           GetPosition(&DescrMot1),
                           GetPosition(&DescrMot2));
                lcd_gotoxy(1,4);
                printf_lcd("M3:%06d ",
                           GetPosition(&DescrMot3));
            }
            break;

        case 2 :
            // Lance 2ème déplacement d'ensemble
            // Exécution de trois mouvements
            RelMove(&DescrMot1, 500);
            RelMove(&DescrMot2, 4000);
            RelMove(&DescrMot3, 2000);
            APP_SituationMouvement = 3;
            break;

        case 3 :
            // Attente fin des 3 mouvement
            if ( IsInPosition(&DescrMot1) &&
                IsInPosition(&DescrMot2) &&
                IsInPosition(&DescrMot3)) {
                APP_SituationMouvement = 0;
                // Affiche la position des 3 moteurs
                lcd_gotoxy(1,3);
                printf_lcd("M1:%06d M2:%06d ",
                           GetPosition(&DescrMot1),
                           GetPosition(&DescrMot2));
                lcd_gotoxy(1,4);
                printf_lcd("M3:%06d ",
                           GetPosition(&DescrMot3));
                if (GetPosition(&DescrMot3) > 150000 ) {
                    MvtInit(&DescrMot1, 1);
                    MvtInit(&DescrMot2, 2);
                    MvtInit(&DescrMot3, 3);
                }
            }
            break;
    }

    appData.state = APP_STATE_WAIT;
    break;
}
```

### 10.2.2.10. LA FONCTION APP\_EXECMOVEMENTS

Cette fonction, prévue pour un appel cyclique, effectue l'appel des fonctions ExecMove.

```
void APP_ExecMovements(void)
{
    ExecMove (&DescrMot1);
    ExecMove (&DescrMot2);
    ExecMove (&DescrMot3);
}
```

¶ C'est ici que se réalise en quelque sorte l'exécution en parallèle.

### 10.2.2.11. LES FONCTIONS APP\_UPDATEPOSX

Ces fonctions servent d'interface aux fonctions de gestion des moteurs.

```
void APP_UpdatePos1(void)
{
    IncPosition (&DescrMot1);
}

void APP_UpdatePos2(void)
{
    IncPosition (&DescrMot2);
}

void APP_UpdatePos3(void)
{
    IncPosition (&DescrMot3);
}
```

### 10.2.2.12. TEST DU FONCTIONNEMENT

Il faut câbler les sorties OC sur les Int Ext :

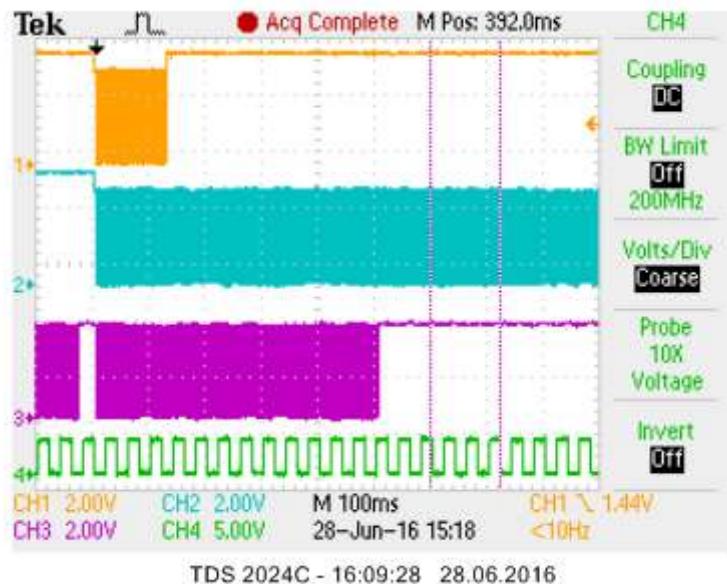
- OC2 → Int1 76 → 18
- OC3 → Int2 77 → 19
- OC4 → Int3 78 → 66

Ch1 : OC2 (moteur 1)

Ch2 : OC3 (moteur 2)

Ch3 : OC4 (moteur 3)

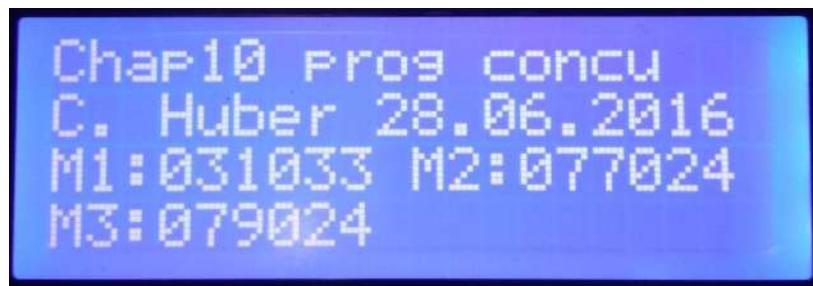
Ch4 : LED\_2  
(toggle dans application)



On peut observer les séquences de mouvement avec les périodes d'arrêt (signal au niveau haut).

#### 10.2.2.12.1. Respect des positions

Le cycle de traitement des "moteurs" étant de 1 ms et la période des OC de 250 us, il est normal que la position affichée dépasse de quelques impulsions.



### 10.3. PROGRAMMATION CONCURRENTE PAR MULTI-THREADING

Ce type de programmation requiert l'existence d'un noyau ou système d'exploitation capable de gérer des processus fonctionnant en pseudo parallélisme selon le principe suivant :

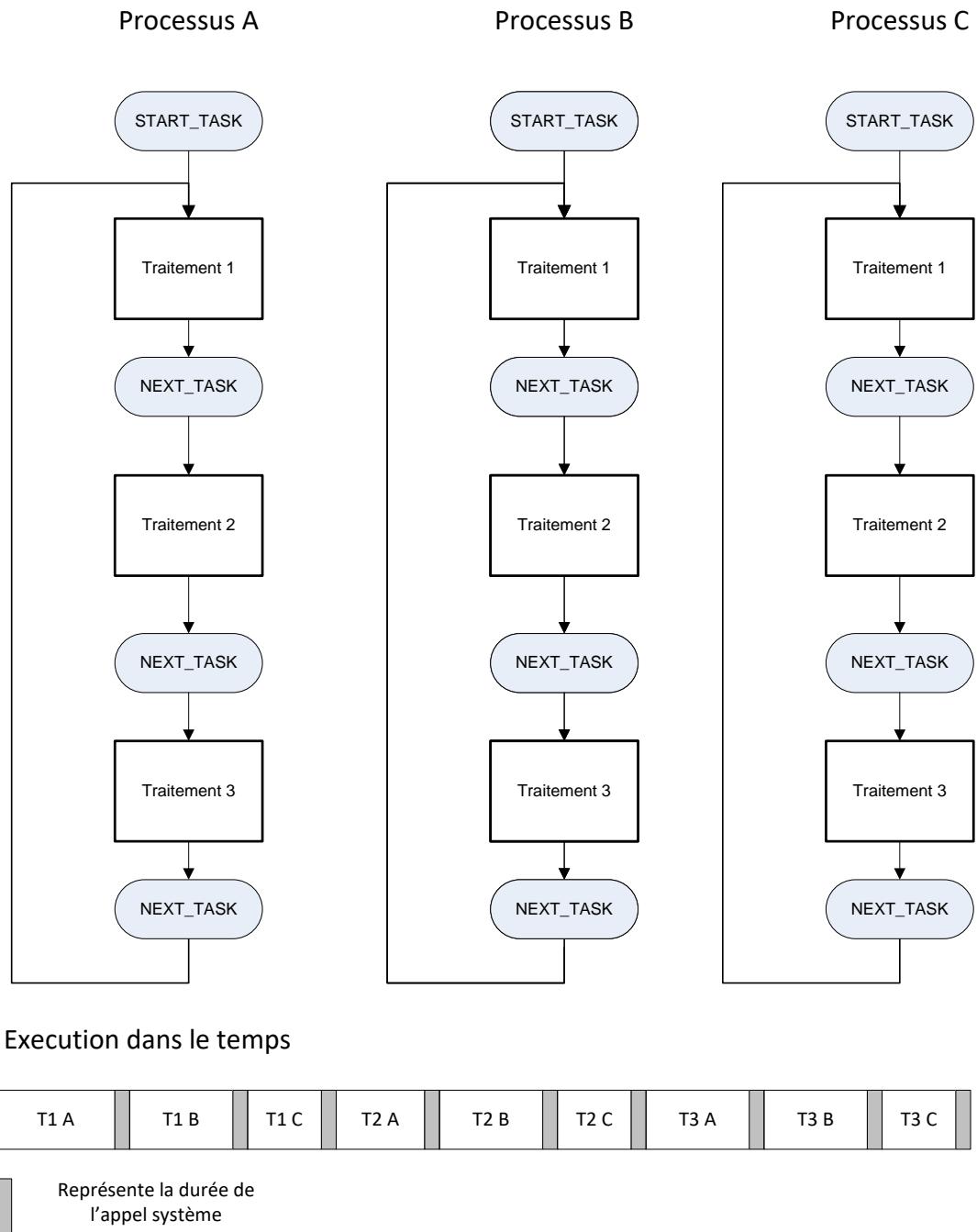


Figure 10-3

Chaque processus s'exécute pour lui-même, mais pour assurer la répartition du temps CPU par tranches, chaque processus doit, lorsqu'une action est accomplie, exécuter un appel au système pour que le processus suivant puisse s'exécuter. Ceci s'appelle un noyau **non-préemptif ou coopératif**.

Il existe aussi des systèmes n'utilisant pas d'appel explicite. Dans ce cas, c'est le système qui attribue le temps CPU d'un processus à l'autre sur la base d'une interruption à intervalle régulier. On appelle cela un noyau **préemptif**.

### 10.3.1. GESTION DES 3 MOUVEMENTS AVEC MULTI-PROCESSUS

Si on reprend notre exemple de gestion des 3 mouvements dans un cadre multiprocessus, le traitement d'un mouvement par un processus devient :

Processus Gestion d'un mouvement

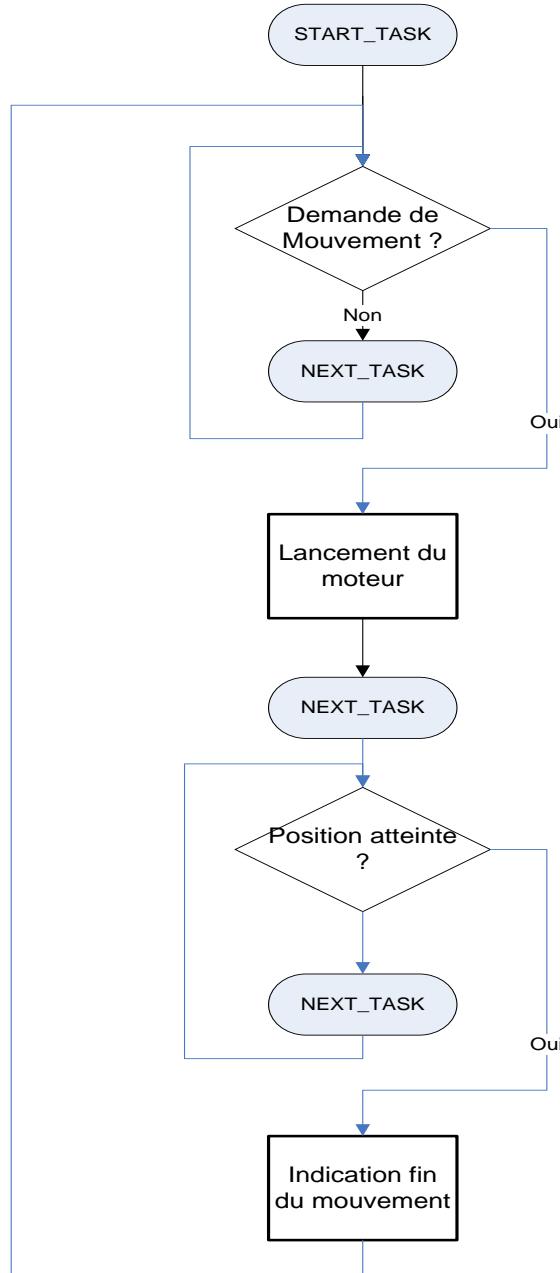


Figure 10-4

On constate que mis à part les appels système (`Next_Task`), on retrouve une écriture plus naturelle de la séquence.

### 10.3.2. FREERTOS AVEC HARMONY

Harmony, dans sa section "Third Party", supporte différents RTOS (Real Time Operating System) : notre choix se porte sur FreeRTOS.

FreeRTOS est un système d'exploitation temps réel faible empreinte, portable, préemptif et libre pour microcontrôleur. Il a été porté sur plusieurs dizaines d'architectures différentes.

Il est parmi les plus utilisés dans le marché des systèmes d'exploitation temps réel. Les domaines d'applications sont assez larges, car les principaux avantages de FreeRTOS sont l'exécution temps réel, un code source ouvert et une taille très faible. On le retrouvera dans des systèmes embarqués qui ont des contraintes d'espace pour le code, mais aussi pour des systèmes de traitement vidéo et des applications réseau qui ont des contraintes temps réel.

En étudiant et modifiant une copie du projet

<Répertoire Harmony>\v<n>\apps\rtos\freertos\basic, nous arrivons à la conclusion que l'on peut utiliser vTaskDelay() pour réaliser un équivalent du NEXT\_TASK de la Figure 10-4.

Cette partie a été réalisée avec les logiciels suivants :

- Harmony v1\_06
- MPLABX IDE v3.10
- XC32 v1.40

#### 10.3.2.1. LA FONCTION VTASKDELAY

Voici le prototype de la fonction vTaskDelay :

```
void vTaskDelay( const TickType_t xTicksToDelay );
```

Et les commentaires qui vont avec :

Delay a task for a given number of ticks. The actual time that the task remains blocked depends on the tick rate. The constant portTICK\_PERIOD\_MS can be used to calculate real time from the tick rate - with the resolution of one tick period.

vTaskDelay() specifies a time at which the task wishes to unblock **relative to** the time at which vTaskDelay() is called. For example, specifying a block period of 100 ticks will cause the task to unblock 100 ticks after vTaskDelay() is called.

### 10.3.2.2. LA FUNCTION vTASKDELAYUNTIL

L'utilisation de la fonction vTaskDelay() n'est pas appropriée pour contrôler la fréquence d'exécution d'une tâche périodique. En effet, le temps d'exécution de la tâche elle-même (qui peut être variable), ainsi que les autres tâches et les interruptions vont affecter la fréquence à laquelle vTaskDelay() sera appelée, et par conséquent le prochain instant d'exécution de la tâche.

La fonction **vTaskDelayUntil()** permet une exécution de tâche à fréquence fixe. Ceci est rendu possible en lui transmettant en paramètre un temps absolu (au lieu de relatif pour vTaskDelay()) auquel le délai se terminera (et donc la tâche appelante sera débloquée).

Exemple d'utilisation :

```
// Perform an action every 10 ticks.
void vTaskFunction(void)
{
    TickType_t xLastWakeTime;
    const TickType_t xPeriod = 10;

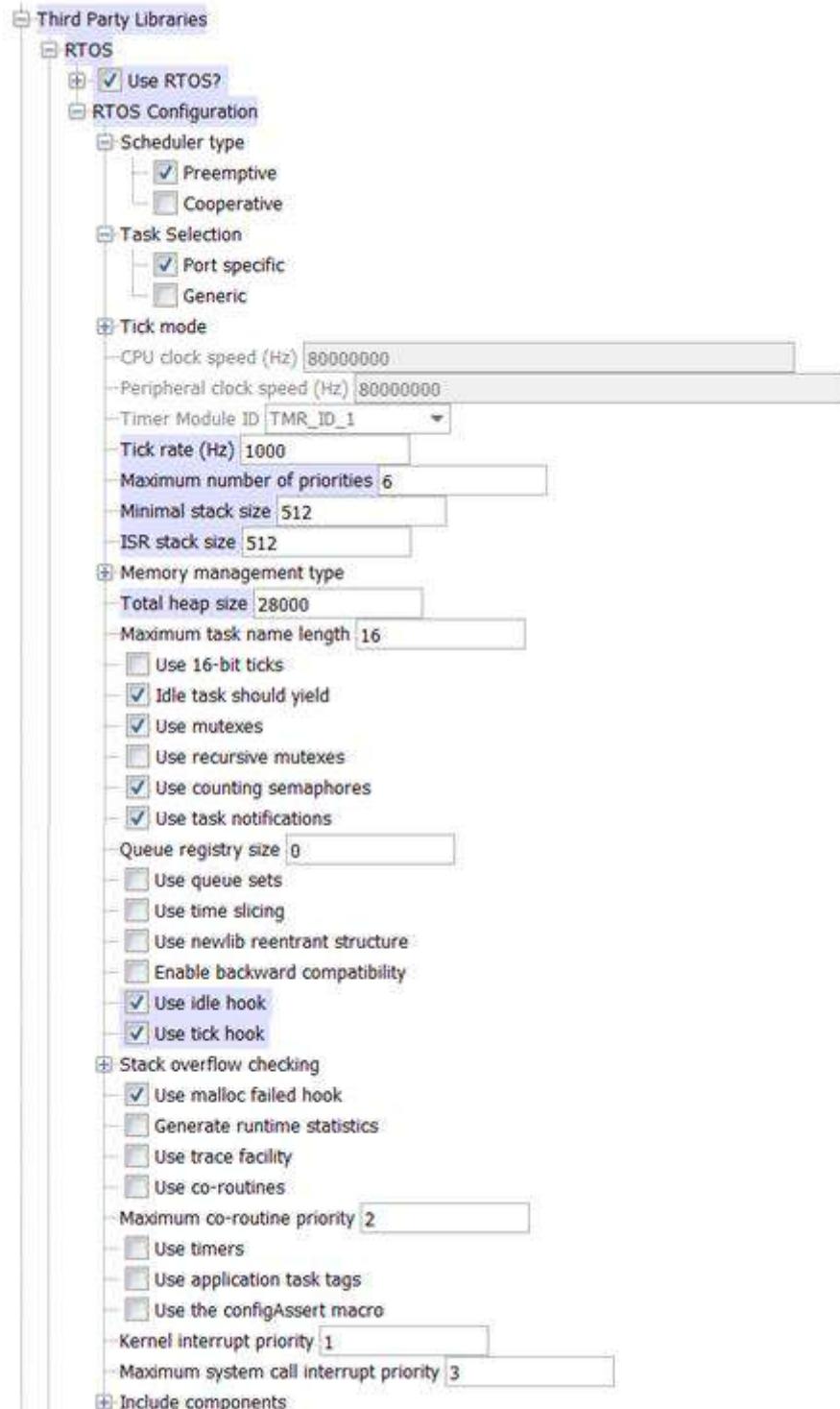
    //Init. the xLastWakeTime variable with the current time
    xLastWakeTime = xTaskGetTickCount();

    for( ; ; )
    {
        // Wait for the next cycle.
        vTaskDelayUntil( &xLastWakeTime, xPeriod );

        // Perform action here.
    }
}
```

### 10.3.3. AJOUT RTOS DANS LA CONFIGURATION

Au niveau des Third Party Librairies on coche Use RTOS et on configure en s'inspirant de l'exemple.

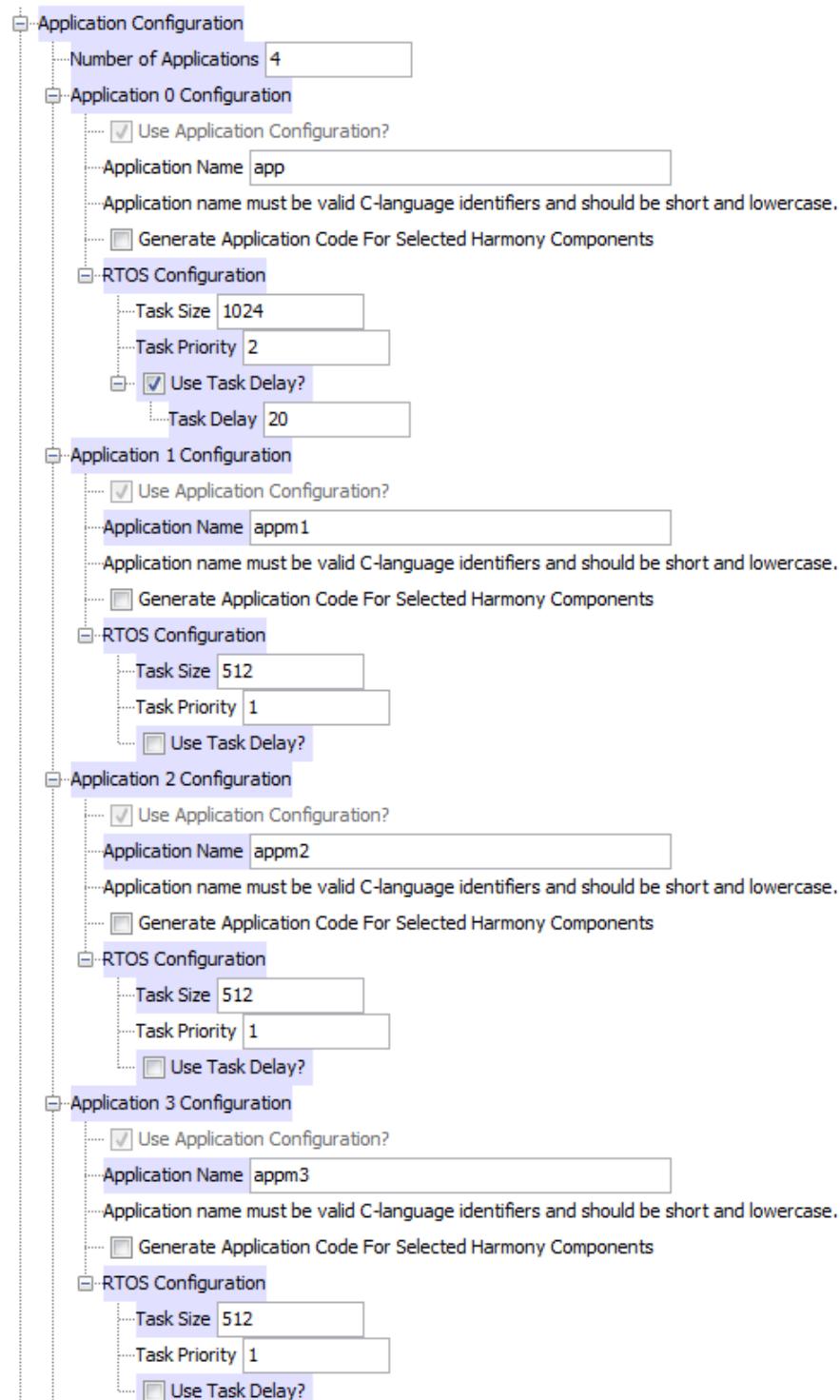


En se documentant, on trouve que le timer 1 est utilisé par FreeRTOS. On ne peut donc pas l'utiliser !

Cette configuration fonctionne.

### 10.3.3.1. AJOUT DE 3 TACHES SUPPLEMENTAIRES

- Attribution de la priorité 1 aux 3 tâches moteur et 2 à la tâche application.
- Task Delay de valeur 20 ticks pour l'application principale. La configuration de FreeRTOS ci-dessus fixe le tick rate à 1 kHz.



### 10.3.3.2. SITUATION DE SYS\_TASKS

Avec l'ajout du RTOS, on constate un changement dans la gestion des tâches.

```
void SYS_Tasks ( void )
{
    /* Create OS Thread for Sys Tasks. */
    xTaskCreate((TaskFunction_t) _SYS_Tasks,
                  "Sys Tasks",
                  1024, NULL, 1, NULL);

    /* Create OS Thread for APP Tasks. */
    xTaskCreate((TaskFunction_t) _APP_Tasks,
                  "APP Tasks",
                  1024, NULL, 1, NULL);

    /* Create OS Thread for APPM1 Tasks. */
    xTaskCreate((TaskFunction_t) _APPM1_Tasks,
                  "APPM1 Tasks",
                  512, NULL, 1, NULL);

    /* Create OS Thread for APPM2 Tasks. */
    xTaskCreate((TaskFunction_t) _APPM2_Tasks,
                  "APPM2 Tasks",
                  512, NULL, 1, NULL);

    /* Create OS Thread for APPM3 Tasks. */
    xTaskCreate((TaskFunction_t) _APPM3_Tasks,
                  "APPM3 Tasks",
                  512, NULL, 1, NULL);

    /*****
     * Start RTOS *
     *****/
    vTaskStartScheduler(); /* This function never returns.
                                */
}
```

### 10.3.3.3. REALISATION DES TACHES

A la suite de SYS\_Tasks, on trouve l'implémentation des tâches sous la forme suivante :

```
static void _SYS_Tasks ( void )
{
    while(1)
    {
        /* Maintain system services */
        SYS_DEVCON_Tasks(sysObj.sysDevcon);

        /* Maintain Device Drivers */

        /* Maintain Middleware */

        /* Task Delay */
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

#### 10.3.3.3.1. Appel de APP\_Tasks

Voici la situation de APP\_Tasks, avec son delay de 20 ms entre chaque appel :

```
static void _APP_Tasks(void)
{
    while(1)
    {
        APP_Tasks();
        vTaskDelay(20 / portTICK_PERIOD_MS);
    }
}
```

#### 10.3.3.3.2. Appel de APPM1\_Tasks

Voici la situation de l'appel de APPM1\_Tasks() avec l'ajout d'une led pour observation :

```
static void _APPM1_Tasks(void)
{
    while(1)
    {
        APPM1_Tasks();
        BSP_LEDToggle(BSP_LED_3);
    }
}
```

### 10.3.3.3. Appel de APPM2\_Tasks

Voici la situation de l'appel de **APPM2\_Tasks()** :

```
static void _APPM2_Tasks(void)
{
    while(1)
    {
        APPM2_Tasks();
    }
}
```

### 10.3.3.4. Appel de APPM3\_Tasks

Voici la situation de l'appel de **APPM3\_Tasks()** :

```
static void _APPM3_Tasks(void)
{
    while(1)
    {
        APPM3_Tasks();
    }
}
```

⌚ Il va donc être possible d'introduire le traitement du mouvement dans chacune des applications.

### 10.3.4. REALISATION DES TACHES MOTEURS

Pour la réalisation, nous allons utiliser des fonctions d'attente utilisant vTaskDelay() avec une valeur de 1ms.

#### 10.3.4.1. REALISATION DE APPM1\_TASKS

Voici la réalisation de APPM1\_Tasks ainsi que la fonction d'initialisation. Le descripteur de mouvement a été placé dans la structure appm1Data.

```
void APPM1_Initialize ( void )
{
    // la variable d'état n'est pas utilisée
    appm1Data.state = APPM1_STATE_INIT;
    MvtInit(&appm1Data.DescrMot1, 1);
}

void APPM1_Tasks ( void )
{
    // Pas besoin de machine d'état
    // -----
    // Attente start
    while ( appm1Data.DescrMot1.StartToDo == false) {
        vTaskDelay(1 / portTICK_PERIOD_MS);
        BSP_LEDToggle(BSP_LED_1);
    }

    // quittance et annonce MvtEnCours
    appm1Data.DescrMot1.MvtSituation = MvtInProgress;
    appm1Data.DescrMot1.StartToDo = false;
    // Demande rotation
    StartMot( appm1Data.DescrMot1.NoMvt) ;

    // Attente position atteinte
    while ( appm1Data.DescrMot1.CurPosition
            < appm1Data.DescrMot1.GoalPosition) {
        vTaskDelay(1 / portTICK_PERIOD_MS);
        BSP_LEDToggle(BSP_LED_1);
    }

    // position atteinte
    // Stop la rotation
    StopMot( appm1Data.DescrMot1.NoMvt) ;
    appm1Data.DescrMot1.MvtSituation = PositionReached;
    LED0_W = 0;
    vTaskDelay(5 / portTICK_PERIOD_MS); // pause
}
```

Pas de machine d'état, mais des boucles avec vTaskDelay.

#### 10.3.4.2. REALISATION DE APPM2\_TASKS

Voici la réalisation de APPM2\_Tasks ainsi que la fonction d'initialisation. Le descripteur de mouvement a été placé dans la structure appm2Data.

```
void APPM2_Initialize ( void )
{
    // la variable d'état n'est pas utilisée
    appm2Data.state = APPM2_STATE_INIT;
    MvtInit(&appm2Data.DescrMot2, 2);
}

void APPM2_Tasks ( void )
{
    // Attente start
    while ( appm2Data.DescrMot2.StartToDo == false) {
        vTaskDelay(1 / portTICK_PERIOD_MS);
    }

    // quittance et annonce MvtEnCours
    appm2Data.DescrMot2.MvtSituation = MvtInProgress;
    appm2Data.DescrMot2.StartToDo = false;
    // Demande rotation
    StartMot( appm2Data.DescrMot2.NoMvt) ;

    // Attente position atteinte
    while ( appm2Data.DescrMot2.CurPosition
            < appm2Data.DescrMot2.GoalPosition) {
        vTaskDelay(1 / portTICK_PERIOD_MS);
    }
    // position atteinte
    // Stop la rotation
    StopMot( appm2Data.DescrMot2.NoMvt) ;
    appm2Data.DescrMot2.MvtSituation = PositionReached;
    vTaskDelay(5 / portTICK_PERIOD_MS); // pause
}
```

#### 10.3.4.3. REALISATION DE APPM3\_TASKS

Voici la réalisation de APPM3\_Tasks ainsi que la fonction d'initialisation. Le descripteur de mouvement a été placé dans la structure appm3Data.

```
void APPM3_Initialize ( void )
{
    // la variable d'état n'est pas utilisée
    appm3Data.state = APPM3_STATE_INIT;
    MvtInit(&appm3Data.DescrMot3, 3);
}
```

```

void APPM3_Tasks ( void )
{
    // Attente start
    while ( appm3Data.DescrMot3.StartToDo == false) {
        vTaskDelay(1 / portTICK_PERIOD_MS);

    }

    // quittance et annonce MvtEnCours
    appm3Data.DescrMot3.MvtSituation = MvtInProgress;
    appm3Data.DescrMot3.StartToDo = false;
    // Demande rotation
    StartMot( appm3Data.DescrMot3.NoMvt) ;

    // Attente position atteinte
    while ( appm3Data.DescrMot3.CurPosition
            < appm3Data.DescrMot3.GoalPosition) {
        vTaskDelay(1 / portTICK_PERIOD_MS);
    }
    // position atteinte
    // Stop la rotation
    StopMot( appm3Data.DescrMot3.NoMvt) ;
    appm3Data.DescrMot3.MvtSituation = PositionReached;
    vTaskDelay(5 / portTICK_PERIOD_MS); // pause

}

```

### 10.3.5. MODIFICATION DE GESMOT

Au niveau de GesMot.h, modification du type énuméré pour donner une notion de situation du mouvement :

```

typedef enum { MvtInProgress, PositionReached
} E_MvtSituation;

```

Au niveau de GesMot.c, suppression de la fonction pour appel cyclique et retouche de la fonction **MvtInit** ainsi que de la fonction :

```

void MvtInit(S_DescrMvt *pDescr, int8_t NoMvt)
{
    pDescr->NoMvt = NoMvt;
    pDescr->StartToDo = false;
    pDescr->MvtSituation = PositionReached;
    pDescr->CurPosition = 0;
    pDescr->GoalPosition = 0;
}

```

```

bool IsInPosition( S_DescrMvt *pDescr)
{
    bool stat = false;
    if ( pDescr->MvtSituation == PositionReached)
        stat = true;
    return stat;
}

```

### 10.3.6. MODIFICATION DE SYSTEM\_INTERRUPT

L'interruption pour le cycle de l'application, ainsi que pour l'appel de **ExecMove** a été supprimée, ce qui permet de n'avoir qu'un timer sans interruption au niveau de la configuration.

System\_interrupt.c ne contient plus que les réponses aux interruptions externes qui ont été modifiées par freeRTOS.

```

void IntHandlerExternalInterruptInstance0(void)
{
    PLIB_INT_SourceFlagClear(INT_ID_0,
                             INT_SOURCE_EXTERNAL_1);
    APP_UpdatePos1();
}

void IntHandlerExternalInterruptInstance1(void)
{
    PLIB_INT_SourceFlagClear(INT_ID_0,
                             INT_SOURCE_EXTERNAL_2);
    APP_UpdatePos2();
}

void IntHandlerExternalInterruptInstance2(void)
{
    PLIB_INT_SourceFlagClear(INT_ID_0,
                             INT_SOURCE_EXTERNAL_3);
    APP_UpdatePos3();
}

```

### 10.3.7. EVOLUTION DE APP.C

Voici l'évolution de app.c, dans lequel on utilise principalement le case APP\_STATE\_SERVICE\_TASKS.

¶ Il faut initialiser le lcd dans le APP\_Initialize qui s'exécute avant que le FreeRTOS soit activé.

```
void APP_Initialize ( void )
{
    /* Place the App state machine in its initial state. */
    appData.state = APP_STATE_INIT;

    lcd_init();
    lcd_b1_on();

    printf_lcd("Chap10 avec RTOS    ");
    lcd_gotoxy(1,2);
    printf_lcd("C. Huber 28.06.2016");
}

void APP_Tasks ( void )
{
    /* Check the application's current state. */
    switch ( appData.state )
    {
        /* Application's initial state. */
        case APP_STATE_INIT:
        {

            // L'initialisation des descripteurs de
            // est faite dans chaque ammmx
            appData.SituationMouvement = 0;

            // Start Timer2 (Base temps OC)
            DRV_TMR0_Start();

            // pas de WAIT
            appData.state = APP_STATE_SERVICE_TASKS;
            break;
        }

        case APP_STATE_WAIT :
            // nothing to do
            break;
    }
}
```

```

        case APP_STATE_SERVICE_TASKS:
            // Inversion LED_2
            BSP_LEDToggle(BSP_LED_2);

            // Machine d'état traitement des 3 mouvements
            switch (appData.SituationMouvement) {
                case 0 :
                    // lance 1er deplacement d'ensemble
                    // Execution de trois mouvements
                    RelMove (&appm1Data.DescrMot1, 300);
                    RelMove (&appm2Data.DescrMot2, 500);
                    RelMove (&appm3Data.DescrMot3, 700);
                    appData.SituationMouvement = 1;
                    break;

                case 1 :
                    // Attente fin des 3 mouvement
                    if (IsInPosition(&appm1Data.DescrMot1)
                        && IsInPosition(&appm2Data.DescrMot2)
                        && IsInPosition(&appm3Data.DescrMot3)
                        ) {
                        appData.SituationMouvement = 2;
                        // Affiche la position
                        // des 3 moteurs
                        lcd_gotoxy(1,3);
                        printf_lcd("M1:%06d M2:%06d ",
                        GetPosition(&appm1Data.DescrMot1),
                        GetPosition(&appm2Data.DescrMot2));
                        lcd_gotoxy(1,4);
                        printf_lcd("M3:%06d ",
                        GetPosition(&appm3Data.DescrMot3));
                    }
                    break;

                case 2 :
                    // lance 2e deplacement d'ensemble
                    // Execution de trois mouvements
                    LED4_W = 1;
                    RelMove (&appm1Data.DescrMot1, 500);
                    RelMove (&appm2Data.DescrMot2, 750);
                    RelMove (&appm3Data.DescrMot3, 1000);
                    appData.SituationMouvement = 3;
                    break;

                case 3 :
                    // Attente fin des 3 mouvement
                    if (IsInPosition(&appm1Data.DescrMot1)
                        && IsInPosition(&appm2Data.DescrMot2)
                        && IsInPosition(&appm3Data.DescrMot3)
                        ) {
                        appData.SituationMouvement = 0;

```

```
        LED4_W = 0;
        // Affiche la position
        // des 3 moteurs
        lcd_gotoxy(1,3);
        printf_lcd("M1:%06d M2:%06d ",
       GetPosition(&appm1Data.DescrMot1),
GetPosition(&appm2Data.DescrMot2));
        lcd_gotoxy(1,4);
        printf_lcd("M3:%06d ",
       GetPosition(&appm3Data.DescrMot3));

        if (GetPosition(&appm3Data.DescrMot3)
            > 150000 ) {
            MvtInit(&appm1Data.DescrMot1, 1);
            MvtInit(&appm2Data.DescrMot2, 2);
            MvtInit(&appm3Data.DescrMot3, 3);
        }
    }
break;
}

// reste en exec
// appData.state = APP_STATE_WAIT;
break;

/* The default state should never be executed. */
default:
{
    break;
}
}

void APP_UpdateState ( APP_STATES NewState )
{
    appData.state = NewState;
}

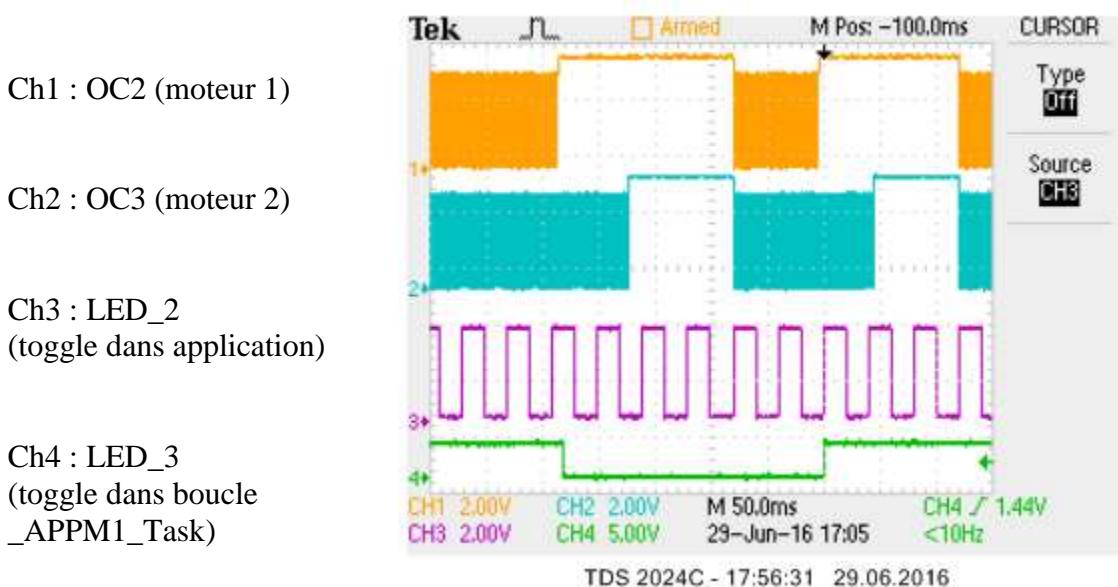
void APP_UpdatePos1(void)
{
    IncPosition(&appm1Data.DescrMot1);
}
void APP_UpdatePos2(void)
{
    IncPosition(&appm2Data.DescrMot2);
}
void APP_UpdatePos3(void)
{
    IncPosition(&appm3Data.DescrMot3);
}
```

### 10.3.8. CONTROLE DU FONCTIONNEMENT

Il faut câbler les sorties OC sur les Int Ext :

- OC2 → Int1 76 → 18
- OC3 → Int2 77 → 19
- OC4 → Int3 78 → 66

```
static void _APPM1_Tasks(void)
{
    while(1)
    {
        APPM1_Tasks();
        // vTaskDelay(1000 / portTICK_PERIOD_MS);
        BSP_LEDToggle(BSP_LED_3);
    }
}
```



Si on met en relation le canal 1 avec le canal 4, on observe une longue attente avant l'exécution du mouvement. Le système fonctionne correctement.

## 10.4. COMMUNICATION INTER-TACHES

Pour une applications donnée, les différentes tâches auront inévitablement besoin de communications entre elles (passage de variables, réservation de ressource, etc.). Il y a donc un besoin pour plusieurs tâches d'accéder aux mêmes données. Dans un environnement multitâches, cela demande quelques précautions.

En effet, étant donné le fonctionnement multitâche de FreeRTOS (auquel peuvent se rajouter des interruptions "utilisateur"), une donnée ne doit être accédée que par une tâche à la fois, et de manière atomique (d'un seul bloc, sans être interrompu au milieu de l'opération).

FreeRTOS met à disposition le nécessaire à cela. On parle de fonctionnement "thread-safe".

### 10.4.1. QUEUE

Une queue permet de transférer des données entre tâches. Dans son utilisation standard, une queue est un FIFO thread-safe.

Une tâche peut :

- Placer un élément dans la queue (give). La tâche est suspendue si la queue est déjà pleine (et donc le contrôle va aux autres tâches).
- Y prendre un élément (take). La tâche est suspendue s'il n'y a aucun élément à prendre.

Exemple d'utilisation d'une queue :

```
#include "queue.h"

QueueHandle_t queueTx = NULL; //déclaration

//Initialisation
queueTx = xQueueCreate( 16, sizeof(char));

//Placement d'un élément dans la queue
xQueueSend( queueTx , &car, 0U );

// ... ailleurs dans le code :
//Réception d'un élément
//Reste bloqué tant que caractère pas reçu
if (xQueueReceive( queueTx, &car, portMAX_DELAY )) ...
```

Le troisième paramètre permet de spécifier le temps d'attente maximum en ticks.

## 10.4.2. SEMAPHORE

Un sémaphore peut être utilisé à des fins d'exclusion mutuelle ou de synchronisation de tâches. Une exclusion mutuelle est un mécanisme utilisé pour éviter que des ressources partagées (par exemple un port série, ou l'accès à un disque) ne soient utilisées en même temps par plusieurs tâches.

### 10.4.2.1. BINARY SEMAPHORE

Un sémaphore binaire peut être vu comme une queue de longueur unitaire où la valeur passée n'a pas d'intérêt. La queue est soit pleine, soit vide (binaire). La tâche qui veut prendre le sémaphore (take) sera donc suspendue jusqu'à ce que la tâche concernée donne le sémaphore (give).

- Exemple de cas d'exclusion mutuelle :  
La tâche qui a le sémaphore peut contrôler une ressource, puis céder son sémaphore et donc le contrôle de la ressource à une autre tâche lorsqu'elle a terminé.
- Exemple de cas de synchronisation de tâches :  
La tâche qui doit effectuer un traitement de donnée peut céder son sémaphore à une autre tâche dès qu'elle a terminé. L'autre tâche pourra alors lire le résultat du traitement et l'utiliser pour son traitement.

Exemple d'utilisation d'un sémaphore :

```
#include "semphr.h"

SemaphoreHandle_t semIntTimer = NULL; //déclaration

//Initialisation
semIntTimer = xSemaphoreCreateBinary();

//pour débloquer la tâche qui attend le semaphore
xSemaphoreGiveFromISR(semIntTimer, NULL);

// ... ailleurs dans le code :
//attente semaphore (portMAX_DELAY = indéfiniment)
xSemaphoreTake(semIntTimer, portMAX_DELAY);
```

#### 10.4.2.2. COUNTING SEMAPHORE

Un counting semaphore peut être vu comme une queue de longueur supérieure à 1. A nouveau, on ne se soucie pas de transmettre des valeurs, mais de combien de valeurs sont présentes dans le "FIFO" de transmission.

Au lieu d'avoir une simple valeur binaire, chaque appel à take incrémente un compteur, et chaque appel à give décrémente.

- Exemple de cas de comptage d'événements :

Dans ce cas, une tâche va faire un "give" à chaque événement (incrémantation de la valeur du semaphore).

Dans une autre tâche, pour chaque évènement, le même traitement doit être effectué. Un "take" est donc effectué (décrémantation de la valeur du sémaphore).

La valeur de compteur est donc égale à la différence entre le nombre d'évènements survenus et le nombre d'évènements effectivement traités.

- Gestion de ressources

Dans ce cas, la valeur du sémaphore indique le nombre de ressources disponibles.

Lorsqu'une tâche veut obtenir une ressource, elle fait un "take" (décrémantation de la valeur), et lorsqu'elle a terminé, elle fait un "give" (incrémantation).

Si la valeur vaut zéro, cela signifie qu'aucune ressource n'est disponible.

#### 10.4.3. MUTEX

MutEx est une contraction de "MUTual EXclusion", ou exclusion mutuelle. Notons que le sémaphore peut également être utilisé à cette fin.

Un mutex est similaire à un sémaphore, excepté que le mutex inclut un mécanisme de changement de priorité de tâche que le sémaphore n'a pas.

##### 10.4.3.1. MUTEX

Dans le cas d'un mutex simple, si une tâche fait un "take" et que le mutex n'est pas disponible et occupé par une tâche de plus faible priorité :

- Comme dans le cas du semaphore binaire, la tâche qui a fait le "take" sera suspendue.
- La différence par rapport au semaphore se trouve dans la gestion des priorités des tâches :

La priorité de la tâche qui détient le mutex sera augmentée à la valeur de la priorité de la tâche qui demande le mutex. Ceci a pour effet de laisser la tâche de plus haute priorité (qui demande le mutex) le moins longtemps possible dans l'état suspendu.

##### 10.4.3.2. RECURSIVE MUTEX

Dans le cas d'un mutex récursif, plusieurs "take" peuvent être effectués par une tâche. Le mutex ne deviendra disponible que lorsque la tâche qui a le mutex aura fait le même nombre d'appel "give".

## 10.5. CONCLUSION

Ce document devrait permettre, en s'inspirant des principes exposés et de l'exemple, de comprendre les principes d'un système d'exploitation temps-réel pour microcontrôleur et de mettre en œuvre FreeRTOS sur PIC32.

## 10.6. SOURCES

- Using The FreeRTOS Real Time Kernel - Microchip PIC32 Edition, Richard Barry, ISBN 978-1-4461-7108-0
- Wikipédia
- <https://www.freertos.org>
- <https://www.freertos.org/Inter-Task-Communication.html>
- [http://www.chibios.org/dokuwiki/doku.php?id=chibios:articles:semaphores\\_mutexes](http://www.chibios.org/dokuwiki/doku.php?id=chibios:articles:semaphores_mutexes)

## 10.7. HISTORIQUE DES VERSIONS

### 10.7.1. VERSION 1.5 JUIN 2015

Création du document (version 1.5 = Harmony) par migration du chapitre 15 PIC18. L'exemple avec FreeRTOS n'est pas encore à disposition.

### 10.7.2. VERSION 1.7 JUIN 2016

Exemple avec machine d'état refait avec Harmony 1.06 et MPLABX 3.10. Essais avec FreeRTOS en chantier.

### 10.7.3. VERSION 1.71 JUIN 2016

Exemple avec machine d'état refait avec Harmony 1.06 et MPLABX 3.10. Mise au point de la partie FreeRTOS.

### 10.7.4. VERSION 1.8 MAI 2017

Relecture générale par SCA. Remise en forme. Ajout communications inter-processus.

### 10.7.5. VERSION 1.9 FEVRIER 2018

Ajout sources.

### 10.7.6. VERSION 1.91 JANVIER 2021

Compléments vTaskDelayUntil et sémaphores.