les écoles supérieures

MINF
Programmation des PIC32MX

Chapitre 6

Timers, PWM & capture

Théorie PIC32MX

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

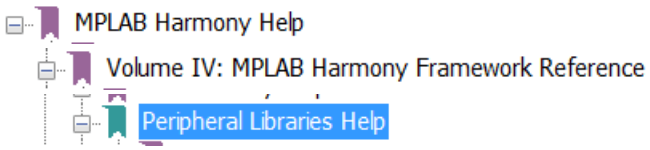
CONTENU DU CHAPITRE 6

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 librairies de Harmony. Il était supporté par le compilateur XC32 (version ≤ 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 coprocessor 0. Le core timer est une des fonctionnalités apportée par le coprocessor 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 coprocessor 0, abrégé CP0.

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

- Registre "Count" (CPO 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 (incrémentation du registre count), la seule exception étant une suspension de l'incrémentation 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 :

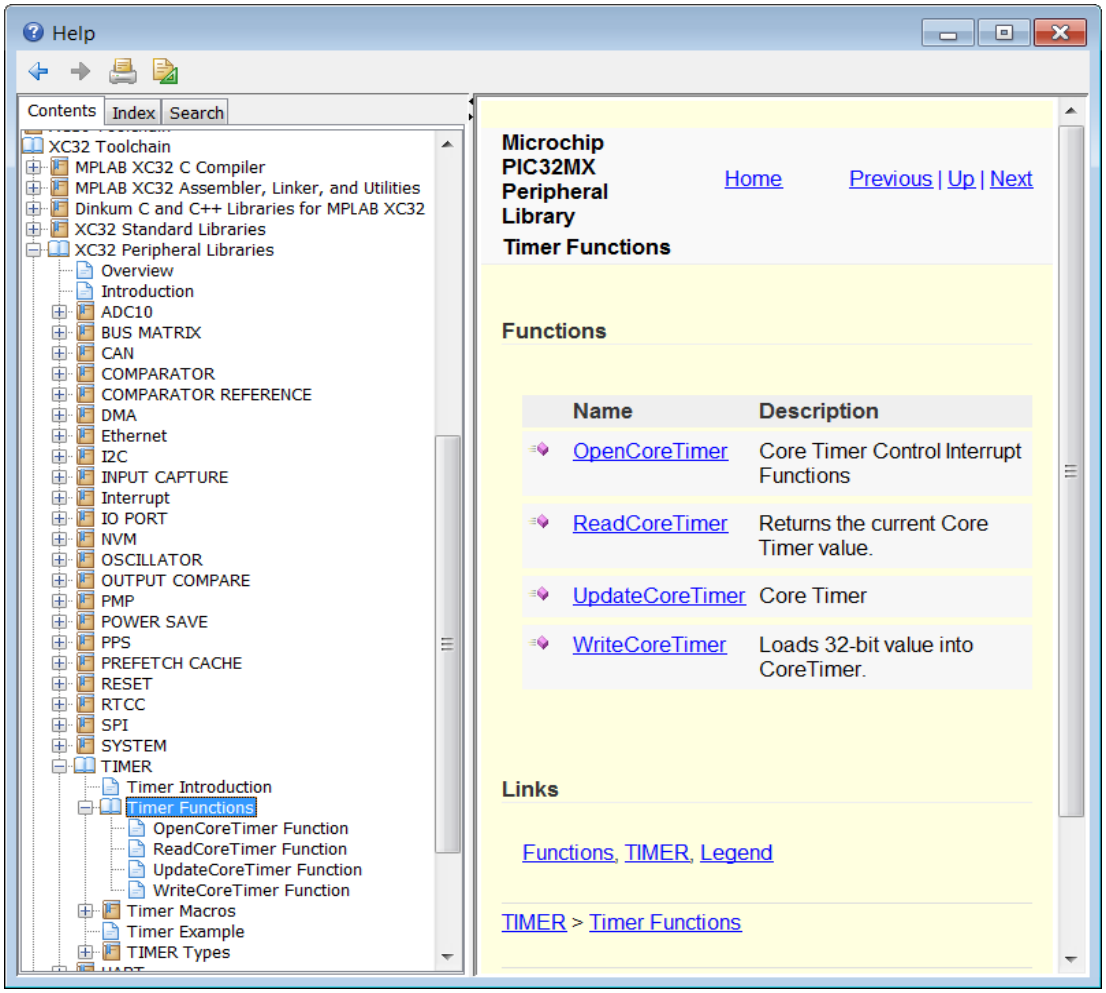
```
#include <cp0defs.h>
```

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 introuvables. 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. $f_{CT} = f_{SYSCLK}/2 = 80MHz / 2 = 40$ 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, ipl5AUTO)
                                    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 discrétion 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

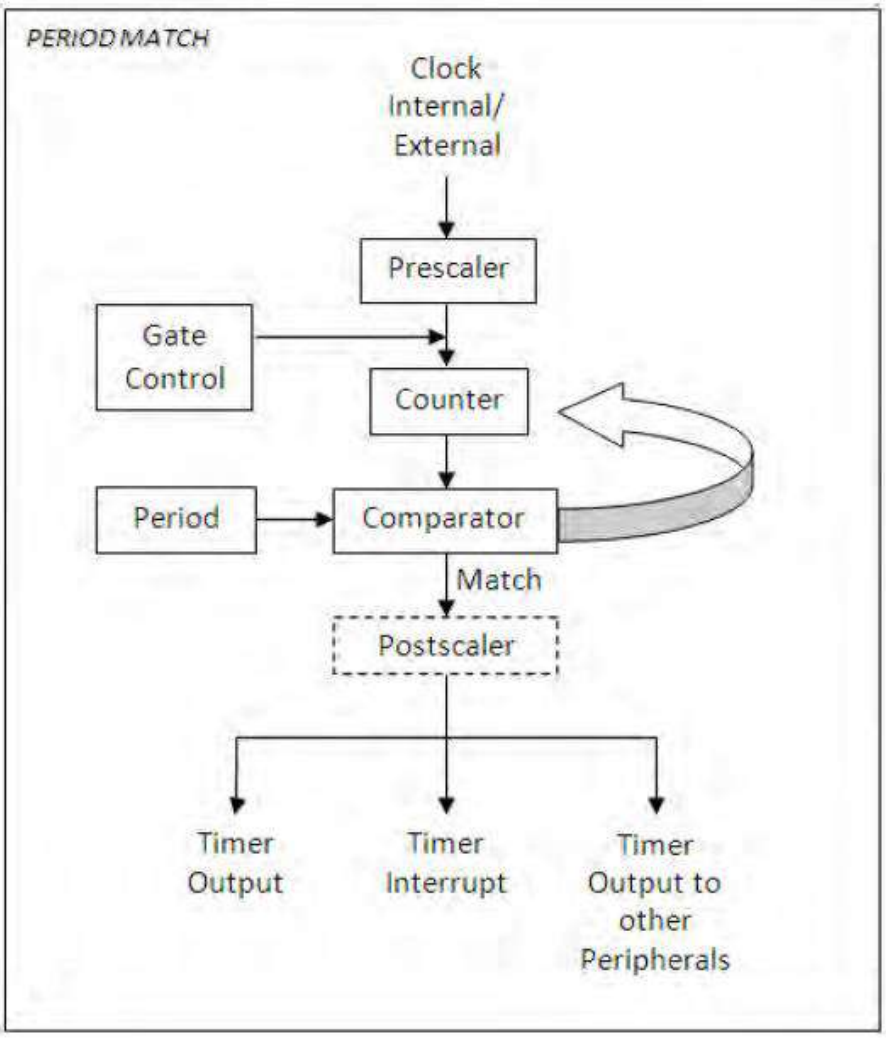
Au niveau d'Harmony, pour gérer les timers périphériques, il est nécessaire d'inclure **plib_tmr.h**

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.

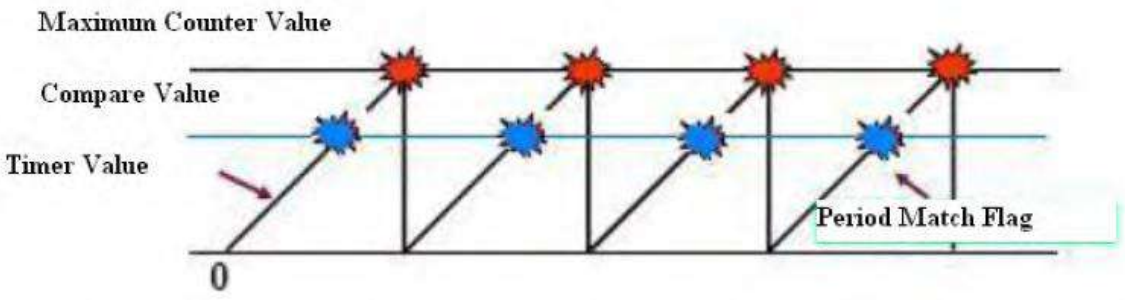
6.2.2. PRINCIPE FONCTIONNEMENT DES TIMERS

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

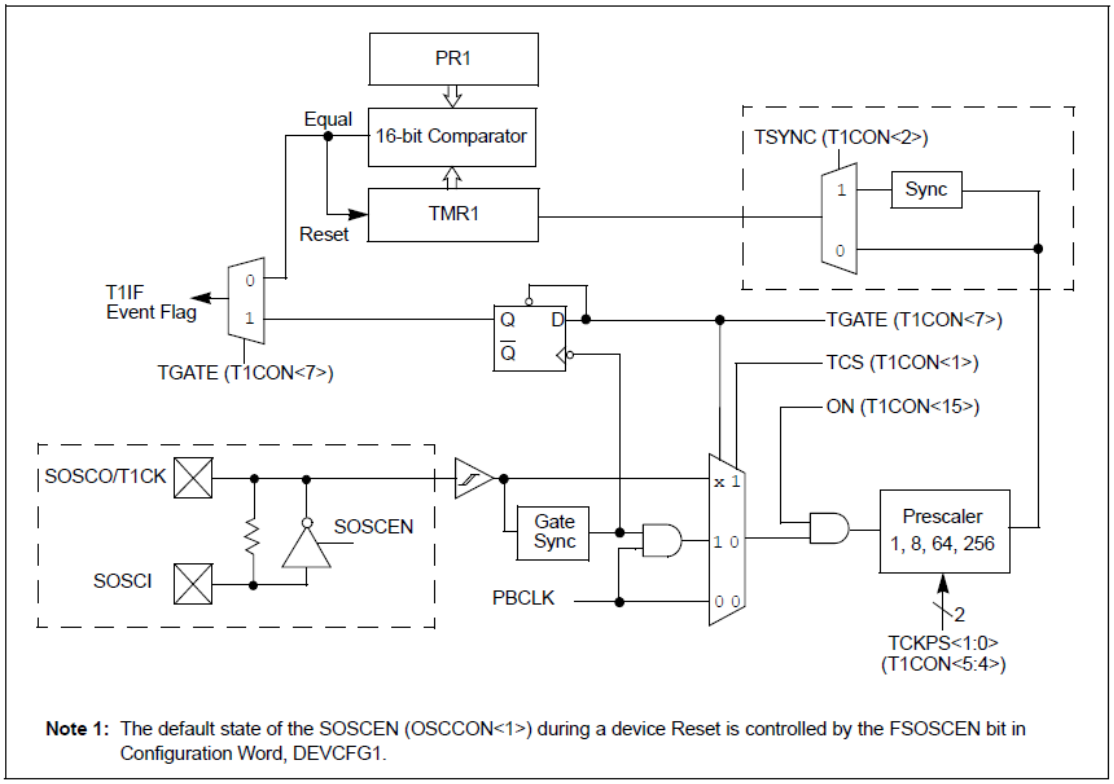
Le diagramme ci-dessous, que l'on trouve dans la documentation Harmony, illustre le principe de fonctionnement des timers :.



6.2.3. LE TIMER 1

Voici le schéma bloc du timer 1 :

FIGURE 13-1: TIMER1 BLOCK DIAGRAM(1)



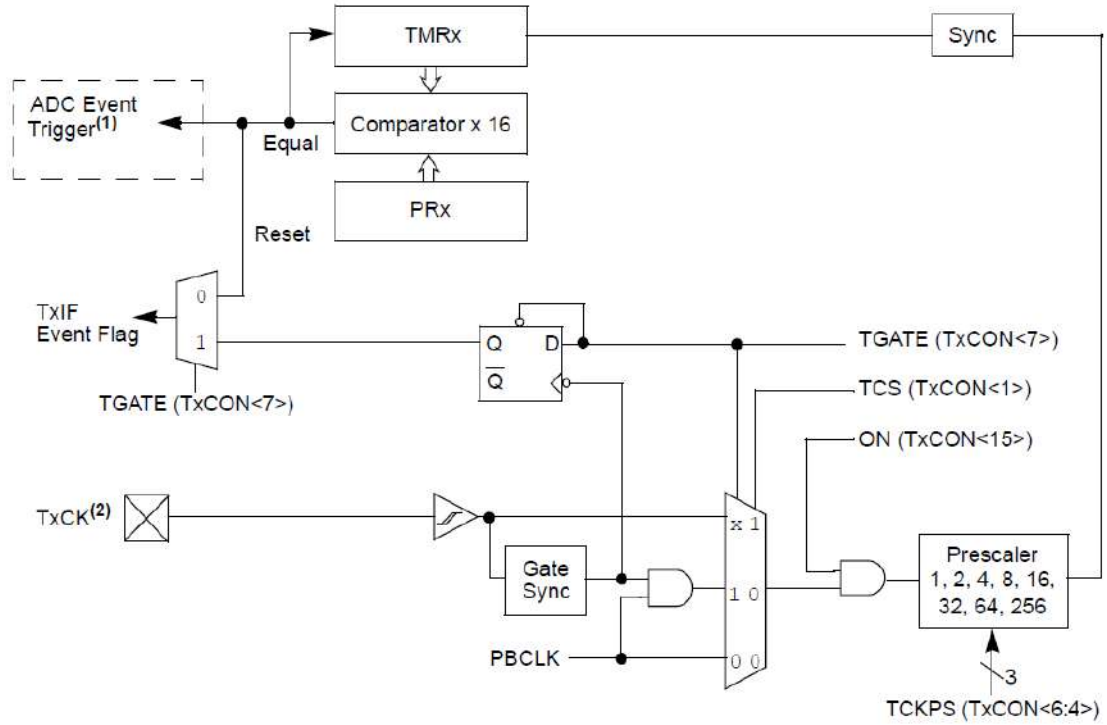
Comme on peut l'observer, le timer 1 est remis à 0 lorsque sa valeur correspond à celle de PR1. Le timer 1 est un timer 16 bits

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.

Cest 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
PLIB_TMR_Mode16BitEnable	Enables the Timer module for 16-bit operation and disables all other modes.
PLIB_TMR_Mode32BitEnable	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 µs = 12,5 ns.

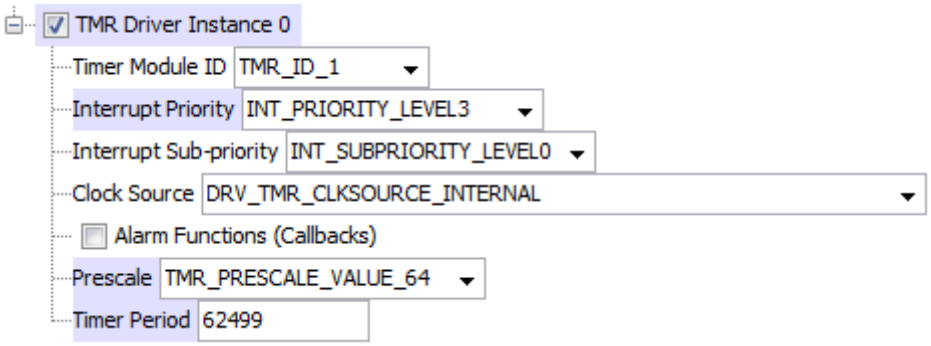
Pour obtenir une période de 50 ms, il faut compter 50'000 / 0,0125 = 4'000'000 ce qui est beaucoup trop grand pour un compteur 16 bits.
Remarque : au lieu de diviser par 0,0125 [µs] il est plus simple de multiplier par 80 [MHz] !

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

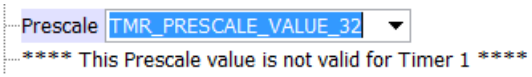
6.2.6.1. CONFIGURATION TIMER 1 AU NIVEAU MHC

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;
}
```

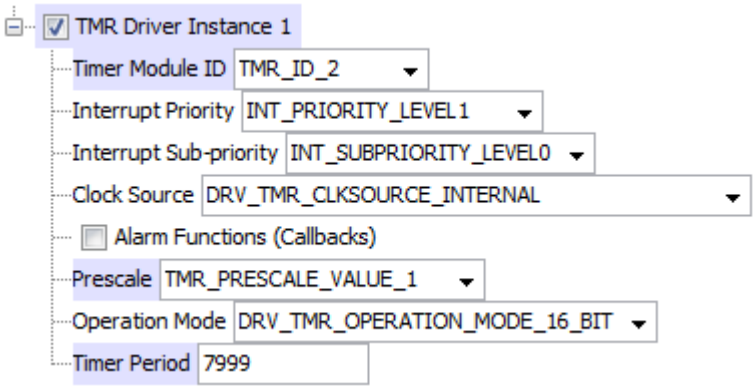
6.2.7. EXEMPLE CONFIGURATION DU TIMER 2

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 paramétrer :
N_{MAX} = (T_{VOULU} / T_{TIMER2})-1 = (100 / 0,0125)-1 = 7'999 ou (100 µs * 80 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.

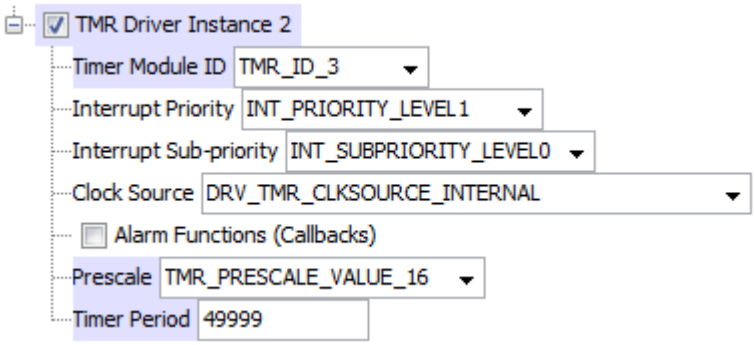
Pour obtenir une période de 10 ms, il faut compter (10'000 * 80)-1 = 799'999 , ce qui est beaucoup trop grand pour un compteur 16 bits.

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;
}
```

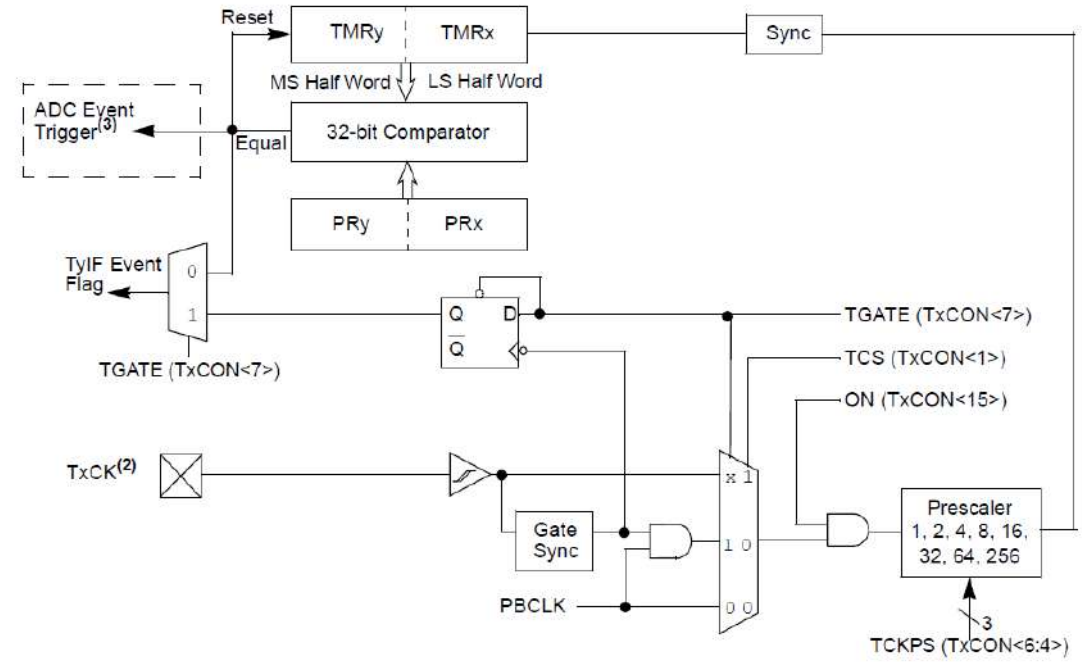
6.2.8.3. LANCEMENT DU TIMER 3

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 ns = 53,7 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.

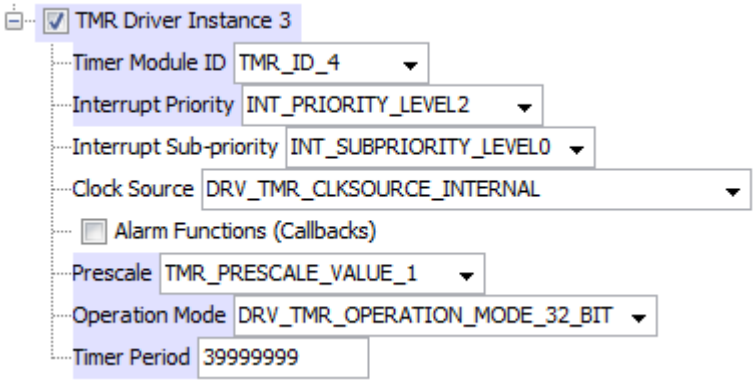
☝ Pour la paire des timers 4 & 5 on configure le timer 4 avec interruption sur timer 5.

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

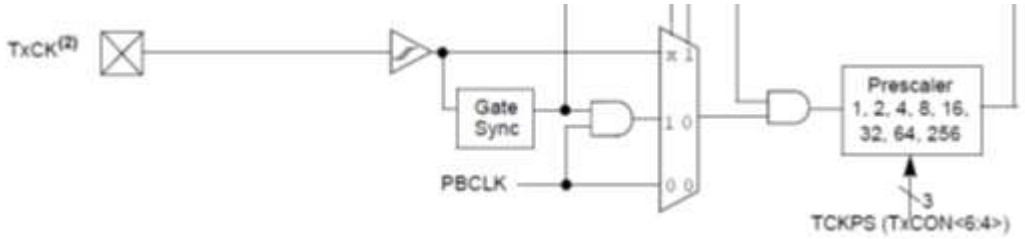
Voici la routine de réponse à l'interruption de la paire timers 4 &5. L'interruption est liée au timer 5, qui correspond au poids fort.

```
// 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
PLIB_TMR_Counter16BitClear	Clears the 16-bit timer value.
PLIB_TMR_Counter16BitGet	Gets the 16-bit timer value.
PLIB_TMR_Counter16BitSet	Sets the 16-bit timer value.
PLIB_TMR_Counter32BitClear	Clears the 32-bit timer value.
PLIB_TMR_Counter32BitGet	Gets the 32-bit timer value.
PLIB_TMR_Counter32BitSet	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 :

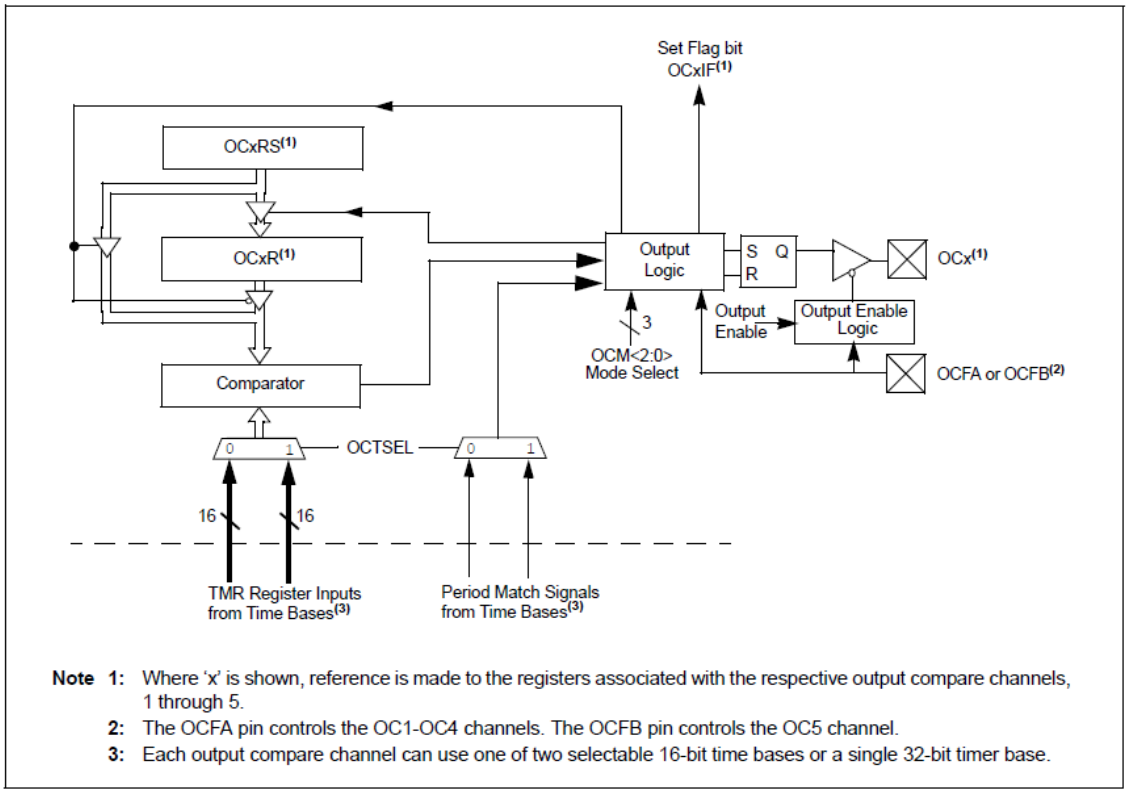
```
uint16_t Timer3Value = DRV_TMR2_CounterValueGet();
```

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



Note 1: Where 'x' is shown, reference is made to the registers associated with the respective output compare channels, 1 through 5.
2: The OCFA pin controls the OC1-OC4 channels. The OCFB pin controls the OC5 channel.
3: Each output compare channel can use one of two selectable 16-bit time bases or a single 32-bit timer base.

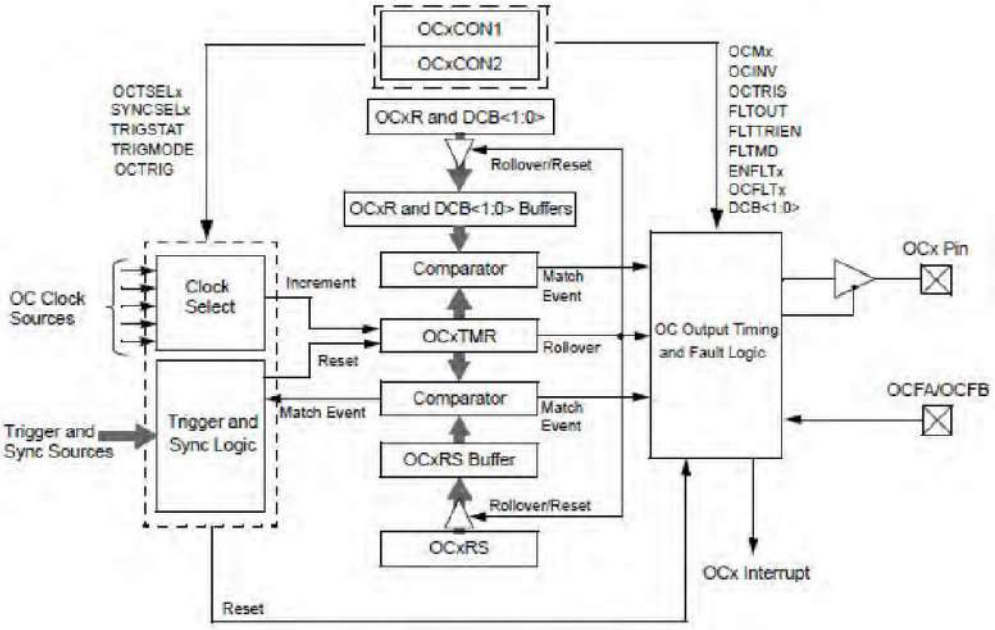
☹ Seulement 2 timers 16 bits possibles comme source : timer 2 et timer 3 !

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 :

PLIB_OC_Disable	Disable the OC module
PLIB_OC_Enable	Enables the OC module.
PLIB_OC_ModeSelect	Selects the compare mode for the OC module.
PLIB_OC_TimerSelect	Selects a clock source for the OC module.
PLIB_OC_Buffer16BitSet	Sets a 16-bit primary compare value for compare operations.
PLIB_OC_Buffer32BitSet	Sets a 32-bit primary compare value for compare operations.
PLIB_OC_BufferSizeSelect	Sets the buffer size and pulse width to 16-bits or 32-bits.
PLIB_OC_PulseWidth16BitSet	Sets a 16-bit pulse width for OC module output.
PLIB_OC_PulseWidth32BitSet	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_COMPARES_MODES

Le type énuméré OC_COMPARES_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	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 â€˜0â€™ 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 PLIB_OC_FaultHasOccurred. The Output Compare will be disabled until the following conditions are met: 1. The external Fault condition has been removed 2. The PWM mode is re-enabled
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) :

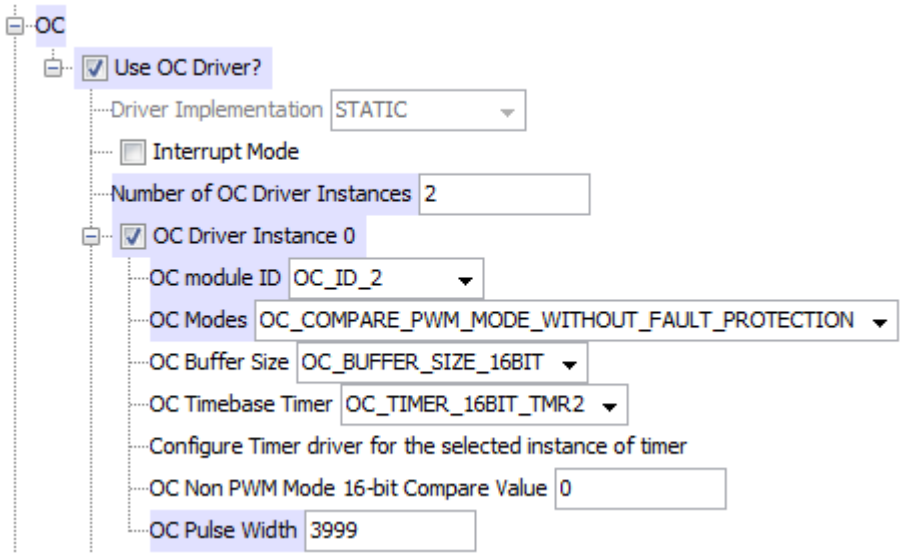
100 µs * 80 = 8'000. Donc valeur maximale de comptage 8'000-1 = 7'999.

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 °

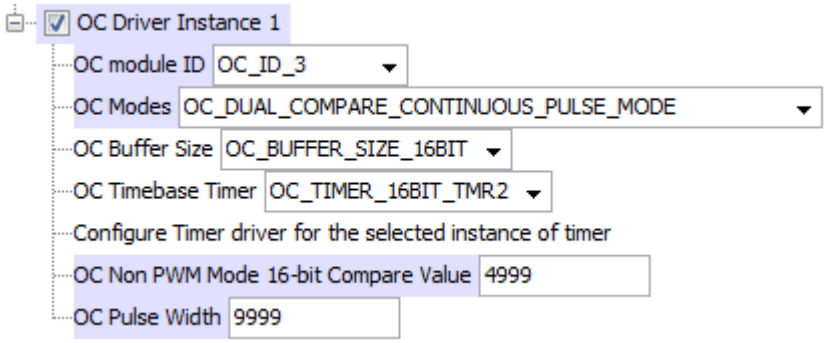
Pour la génération d'impulsions, on utilise le mode **OC_DUAL_COMPARE_CONTINUOUS_PULSE_MODE**. Il faut indiquer le timer 3 comme source pour la comparaison.

☺ 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 apparait 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 1ère 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) :

Canal 1 :
signal en Led3 (toggle à chaque interruption du timer 3)

Canal 2 :
Signal en OC3 (broche 77)

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

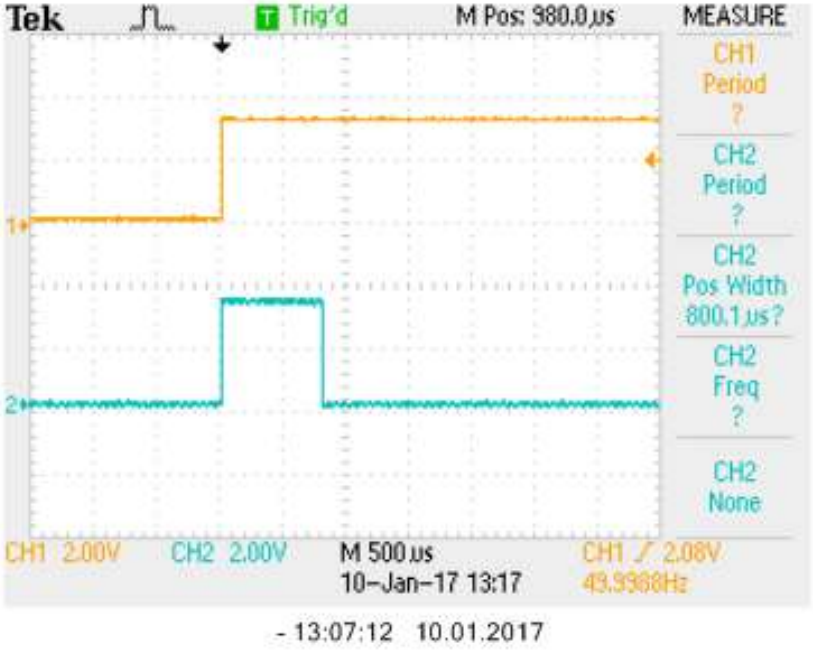
Dans une 2ème phase, l'application modifie la valeur signaux PWM en utilisant les valeurs de l'AD.

```
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;
}
```

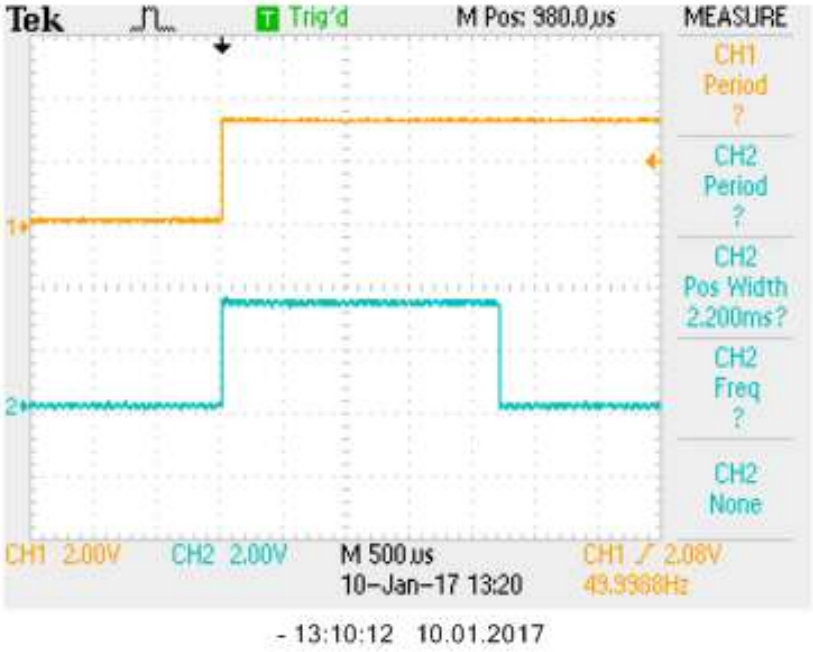
6.3.8.1. OBSERVATION DES RESULTATS

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. EVÉNEMENTS 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 1er flanc.

6.4.1.3. EVENEMENTS MULTIPLES

En utilisant le prescaler :

- Capture lors du 4ème flanc montant sur l'entrée ICx.
- Capture lors du 16ème flanc montant sur l'entrée ICx.

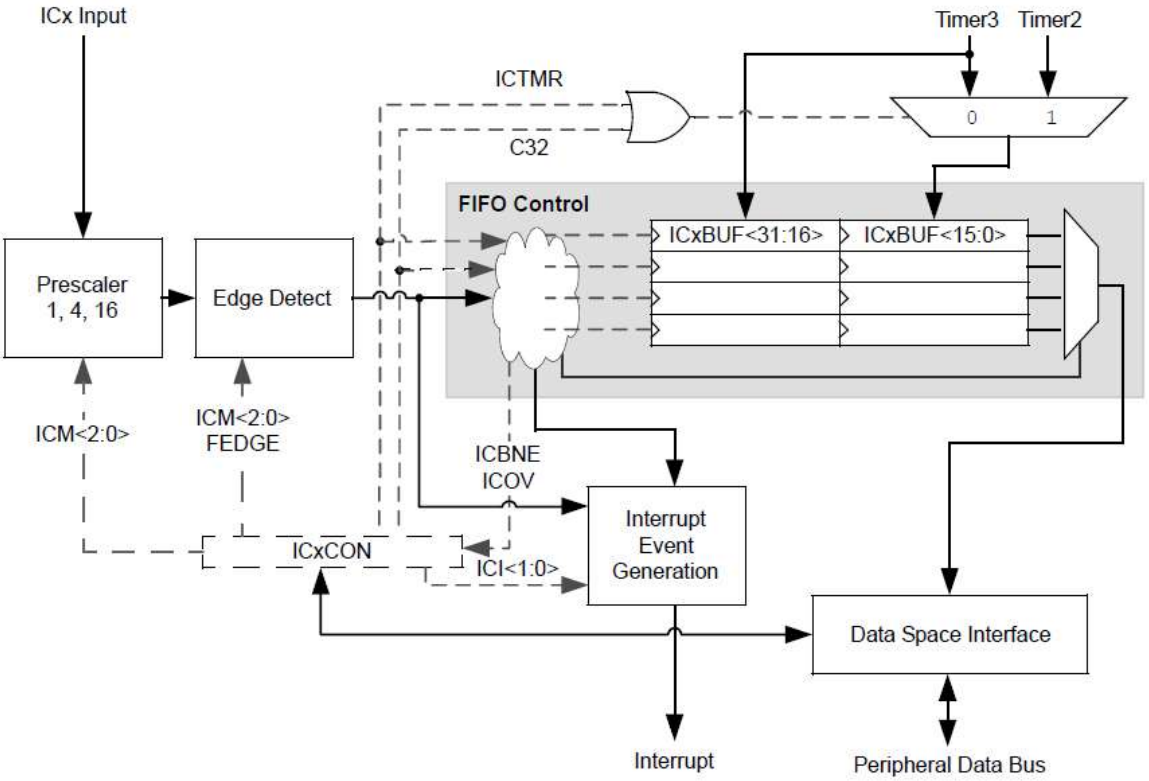
6.4.2. LISTE DES ENTRÉES DE CAPTURES

Pour le PIC32MX795FL512L 100 pin TQFP :

ICx	Nom de la broche	No de la broche
IC1	RTCC/EMDIO/AEMDIO/**IC1**/RD8	**68**
IC2	_SS1/**IC2**/RD9	**69**
IC3	SCK1/**IC3**/PMCS2/PMA15/RD10	**70**
IC4	EMDC/AEMDC/**IC4**/PMCS1/PMA14/RD11	**71**
IC5	ETXD2/**IC5**/PMD12/RD12	**79**

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 1er 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 1er 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 1er 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 1ère 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 **ON:** 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

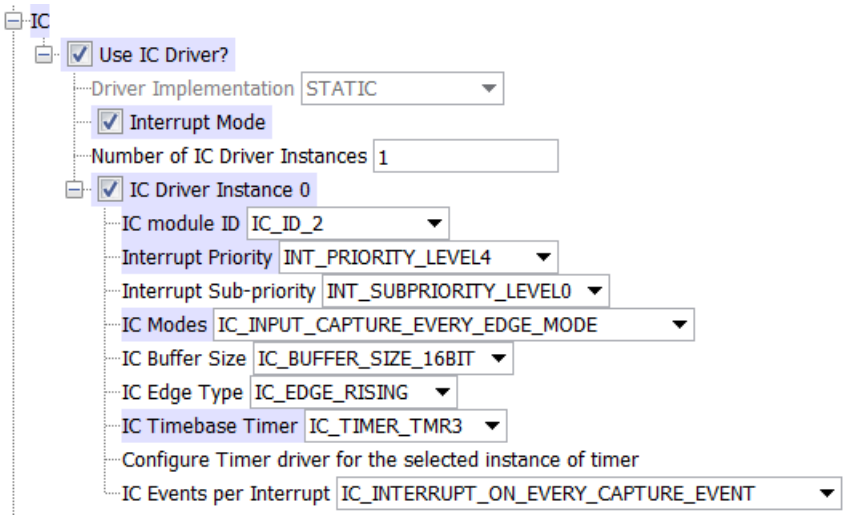
La fonction **PLIB_IC_Enable** active le module IC spécifié.

```
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
PLIB_IC_Buffer16BitGet	Obtains the 16-bit input capture buffer value.
PLIB_IC_Buffer32BitGet	Obtains the 32-bit input capture buffer value.
PLIB_IC_BufferIsEmpty	Checks whether the input capture buffer is empty.
PLIB_IC_BufferOverflowHasOccurred	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 :

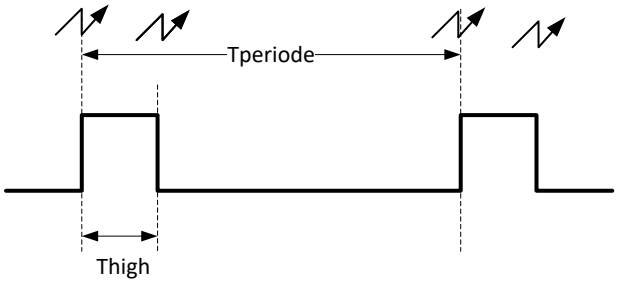
```
void DRV_IC0_Open(void)
{
    PLIB_IC_Enable(IC_ID_2);
    while (!PLIB_IC_BufferIsEmpty(IC_ID_2))
    {
        PLIB_IC_Buffer16BitGet(IC_ID_2);
    }
    PLIB_INT_SourceFlagClear(INT_ID_0,
                             INT_SOURCE_INPUT_CAPTURE_2);
    PLIB_INT_SourceEnable(INT_ID_0,
                          INT_SOURCE_INPUT_CAPTURE_2);
}
```

☹ Il est nécessaire d'effectuer un ajout dans le fichier drv_ic_static.h

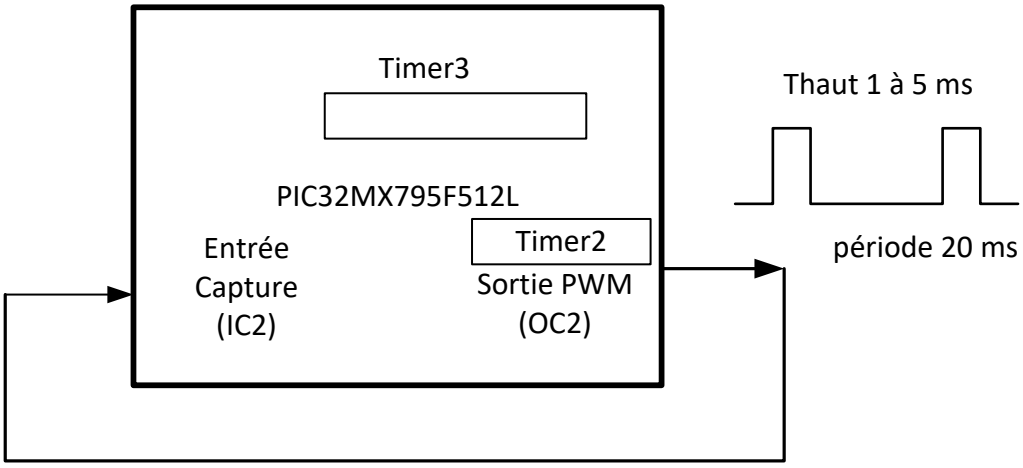
```
// *****************************************************
// Section: Interface Headers for Instance 0 for the static
driver
// *****************************************************

// 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.

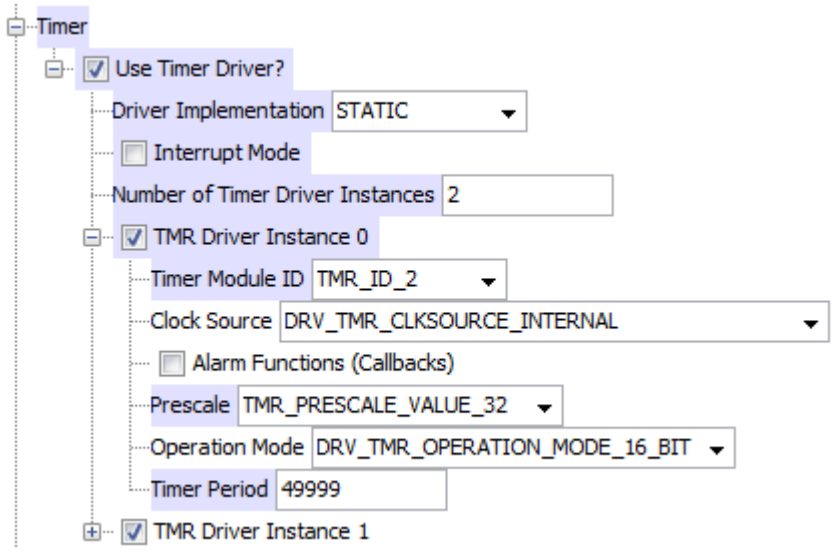


Le signal généré par OC2 (broche 76) est câblé sur IC2 (broche 69). Le timer 3 est le timer capturé.

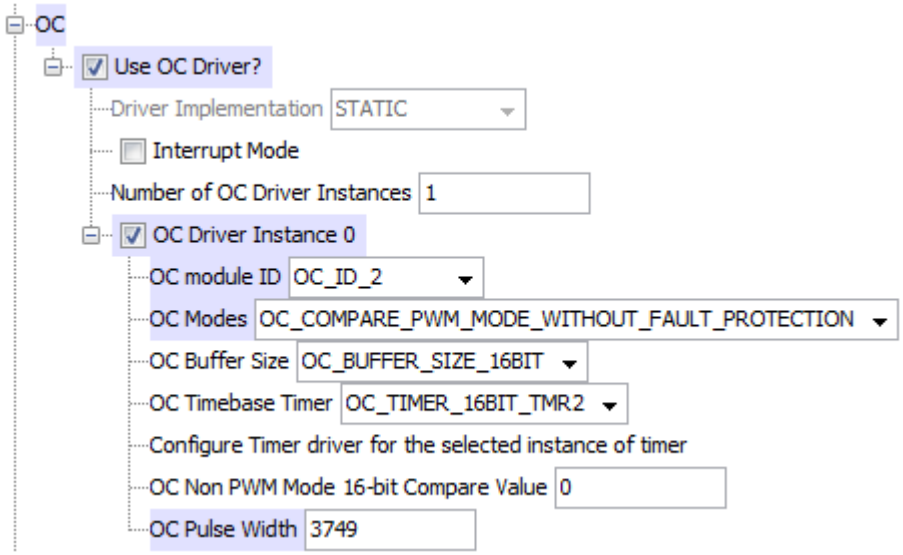
Le timer 2 est configuré pour une période de 20 ms. Le timer 3 travaille avec la période max soit 26.214 ms. Tous les 2 ont un prescaler de 32.

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.

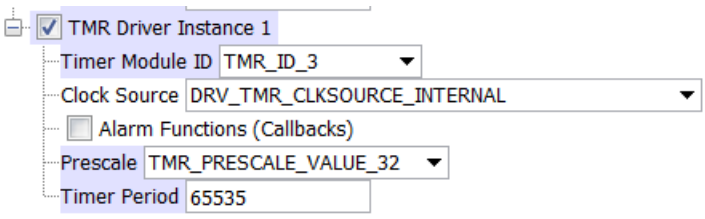


L'OC2 est configuré en PWM en générant une impulsion d'une largeur de 1,5 ms au niveau de la configuration. Valeur de OC_Pulse_Width de (50'000 / 20) * 1,5 = 3750.



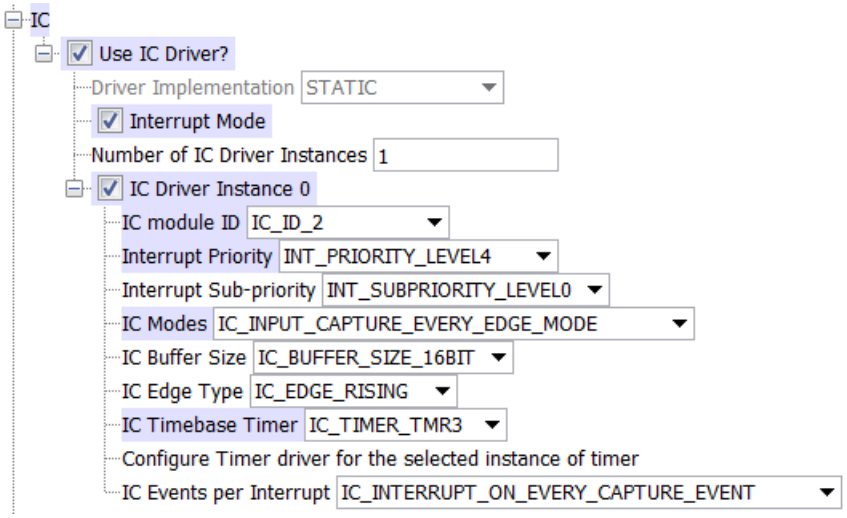
6.4.10.2. CONFIGURATION DU TIMER 3

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

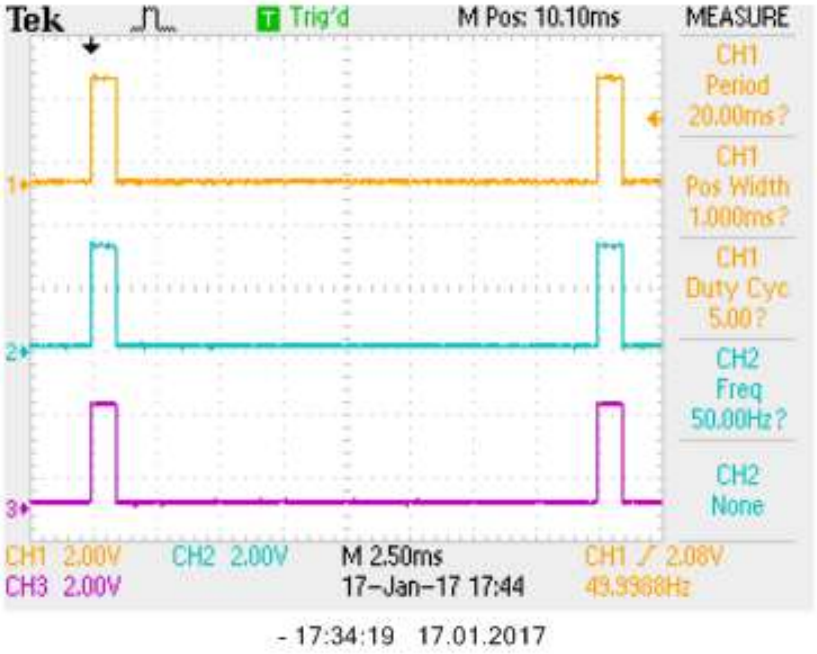
Les observations ci-dessous nous permettent de vérifier que l'on obtient bien une interruption à chaque capture.

💣 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)

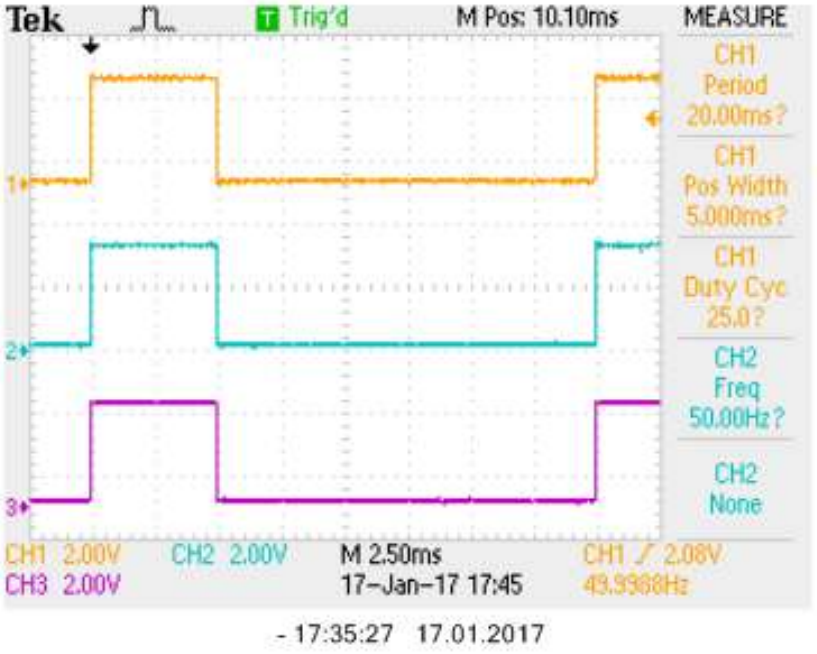


L'affichage sur le lcd nous fournit : Période = 20,000 ms et Thaut = 1,000 ms.

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

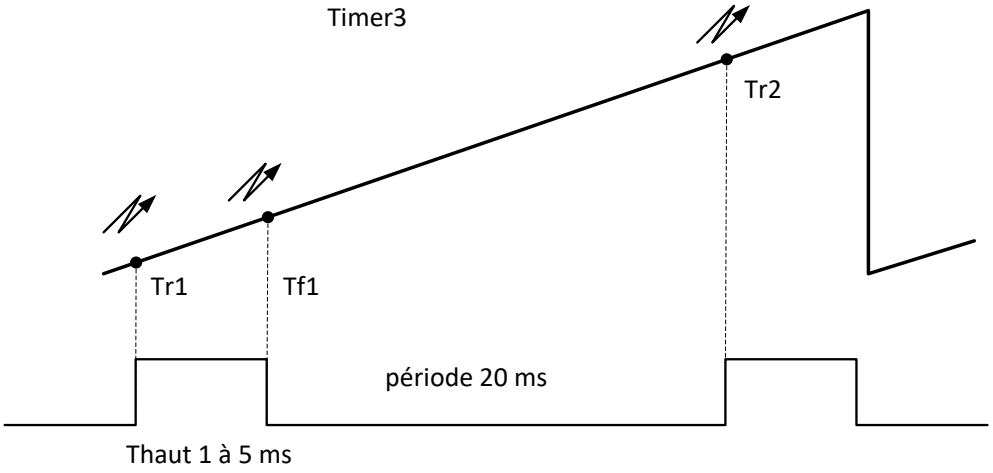
Comme on dispose d'une seule capture à chaque interruption il est possible de lire le FIFO de capture avec la fonction **PLIB_IC_Buffer16BitGet** sans se préoccuper de la situation du tampon de capture.

☝ 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.

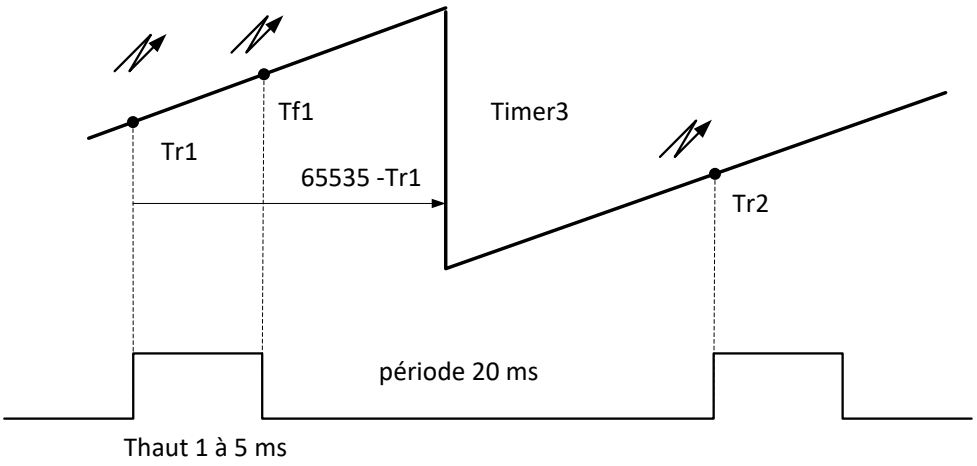
6.4.10.8. GESTION DU REBOUCLEMENT DU TIMER 3

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 rebouclement. La période du signal vaut Tr2–Tr1.



Le diagramme ci-dessous illustre la situation de rebouclement. 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 rebouclement du timer 3 peut aussi se produire entre Tr1 et Tf1.

Si plusieurs rebouclements 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 µs, puis à diviser par 1'000. Avec un prescaler de 32, la durée du tick est de 32 * 0,0125 = 0,4 µs. (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. DETAIL 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 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 rebouclement.

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.