les écoles supérieures

ETML

MINF
Mise en œuvre des microcontrôleurs PIC32MX

Chapitre 5

Timers, interruptions & PWM

🛠 T.P. PIC32MX

Christian HUBER (CHR)
Serge CASTOLDI (SCA)
Version 1.81 décembre 2018

CONTENU DU CHAPITRE 5

5. TIMERS, INTERRUPTIONS ET PWM

Dans ce chapitre nous allons étudier sur la base du MHC (MPLAB Harmony Configurator) : comment mettre en œuvre les timers, répondre à une interruption cyclique, ainsi que générer un signal PWM.

Les documents de référence sont :

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

5.1. CRÉATION DU PROJET AVEC LE MHC

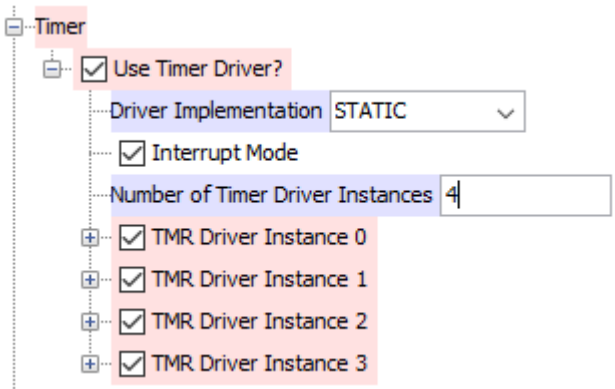
On reprend le même principe de création du projet tel que décrit dans les chapitres précédents. Par contre au niveau de Harmony Framework Configuration dans la section Drivers, nous allons utiliser la section Timer et la section OC (pour PWM).

Nous allons configurer les éléments suivants :

- Timer 1, période 50 ms, interruption niveau 3
- Timer 2, base de temps pour PWM 10 kHz → période 100 µs, pas d'interruption
- Timer 3, base de temps 10 ms pour impulsions, pas d'interruption
- Timer 4&5, période 500 ms, interruption niveau 2
- OC2 en PWM en relation avec timer 2
- OC3 en Continuous pulse en relation avec timer 3

5.1.1. SÉLECTION DES TIMERS

Sous Harmony Framework Configuration > Drivers > Timer, nous introduisons la configuration ci-dessous pour obtenir 4 timers avec des interruptions. Choix de l'implémentation STATIC.

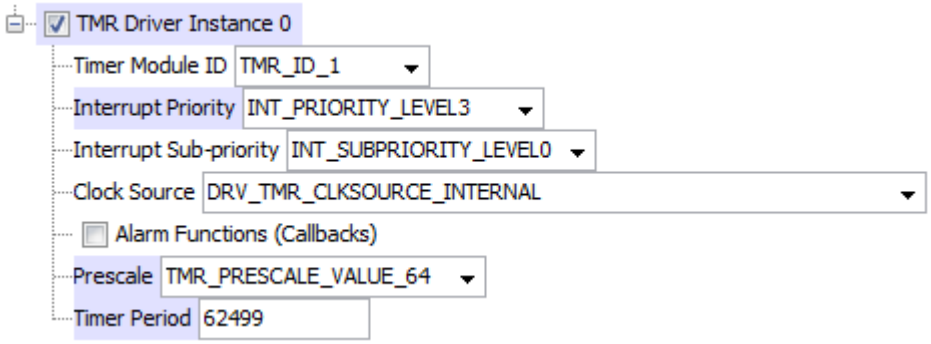


5.1.1.1. CONFIG TMR DRIVER INSTANCE 0

Configuration du timer 1 pour une période de 50 ms et priorité d'interruption 3.

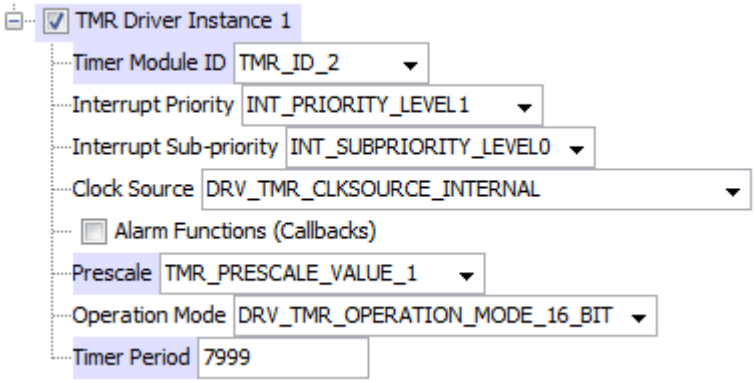
Avec PB_CLOCK 80 MHz :
TimerPeriode = 50'000 µs * 80 = 4000000 dépasse valeur maximale pour timer 16 bits.
Besoin d'un prescaler de 4000000 / 65536 = 61,03 => **64**

☝Le timer 1 ne supporte que 1, 8, 64, 256 contrairement à la liste proposée !
TimerPeriode = (50'000 µs * 80 / 64)-1 = 62'499



5.1.1.2. CONFIG TMR DRIVER INSTANCE 1

Configuration du timer 2 pour obtenir un PWM à 10 kHz, donc une période de 100 µs. Avec PB_CLOCK 80 MHz et un prescaler de 1, il faut une valeur de comparaison de (100 µs * 80)-1 = 7'999 pour obtenir 100 µs. L'interruption ne sera pas utilisée.

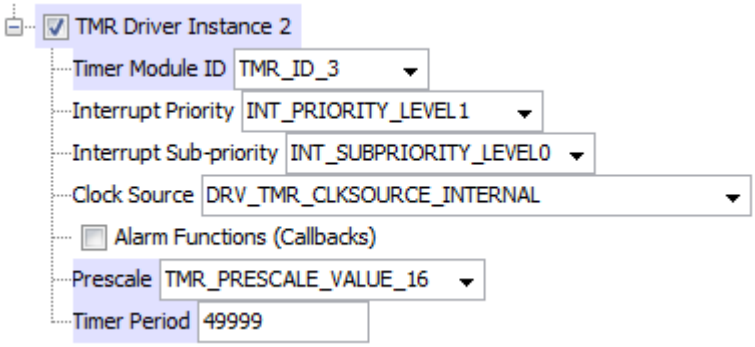


5.1.1.3. CONFIG TMR DRIVER INSTANCE 2

Configuration du timer 3 pour obtenir une période PWM de 10 ms. L'interruption ne sera pas utilisée.

Avec PB_CLOCK 80 MHz :
TimerPeriode = 10'000 µs * 80 = 800000 dépasse valeur maximale pour timer 16 bits.
Besoin d'un prescaler de 800000 / 65536 = 12,2 => **16**

La période du timer sera donc (10000 µs * 80 / 16)-1 = 49'999.

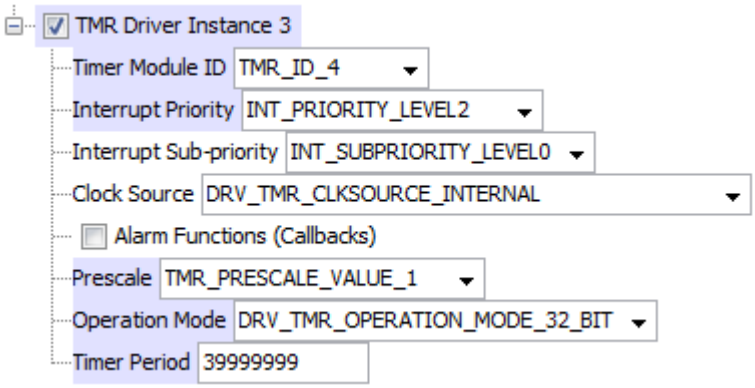


5.1.1.4. CONFIG TMR DRIVER INSTANCE 3

Configuration du timer 4 en 32 bits pour obtenir une période de 500 ms, interruption de niveau 2.

Avec PB_CLOCK 80 MHz:
TimerPeriode = (500'000 µs * 80)-1 = 39'999'999 $< 2^{32} = 4'294'967'296$.

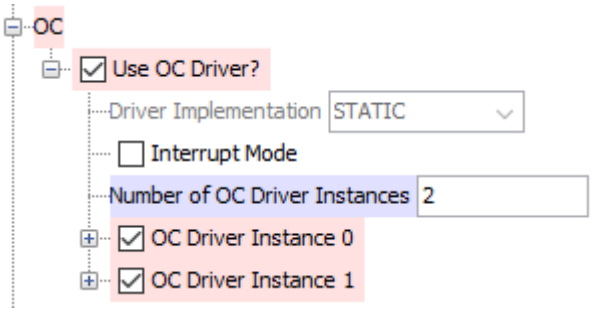
La période du timer sera donc de 39'999'999.



Remarque : en configurant le timer 4 en 32 bits, il y a utilisation de la paire T4 & T5.

5.1.2. SÉLECTION DES OC

Sous Harmony Framework Configuration Drivers, OC nous introduisons la configuration ci-dessous pour configurer 2 OC (OC = Output Compare).



Remarques :

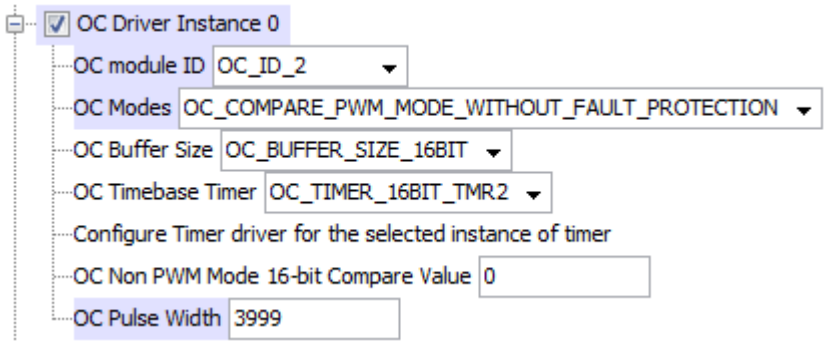
- Pour les drivers OC il n'y a que les static à disposition.
- Les interruptions des OC ne sont pas utiles dans l'application prévue.
- Le lecteur est invité à se reporter aux datasheet du PIC32 ainsi qu'à l'aide de Harmony concernant les différents modes de fonctionnement des OC.

5.1.2.1. CONFIG OC DRIVER INSTANCE 0

- Choix de l'OC2, qui correspond à PWMA_Hbridge sur le kit,
- Utilisation du timer 2 comme base de temps,
- Choix du mode OC_COMPARE_PWM_MODE_WITHOUT_FAULT_PROTECTION, qui semble correspondre au PWM souhaité.

OC_COMPARE_PWM_MODE_WITHOUT_FAULT_PROTECTION	Output Compare module output is PWM signal and is not fault protected

Choix de OC Pulse Width à 3'999, ce qui correspond à un PWM de 50%, car la période du timer 2 correspond à une valeur de 7'999.

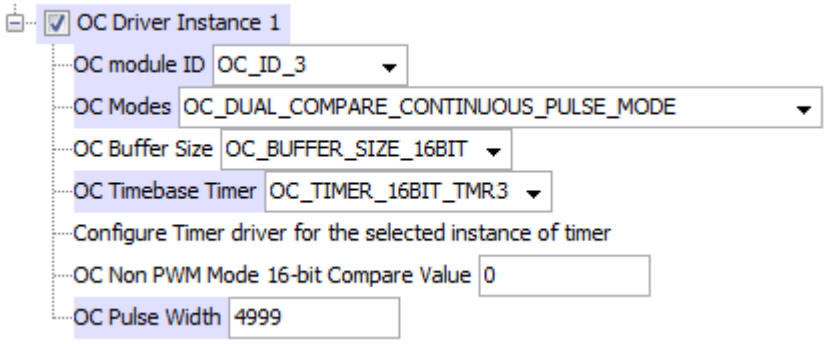


5.1.2.2. CONFIG OC DRIVER INSTANCE 1

- Choix de l'OC3, qui correspond à PWMB_Hbridge sur le kit,
- Utilisation du timer 3 comme base de temps,
- Choix du mode OC_DUAL_COMPARE_CONTINUOUS_PULSE_MODE, qui permet de générer une impulsion cyclique.

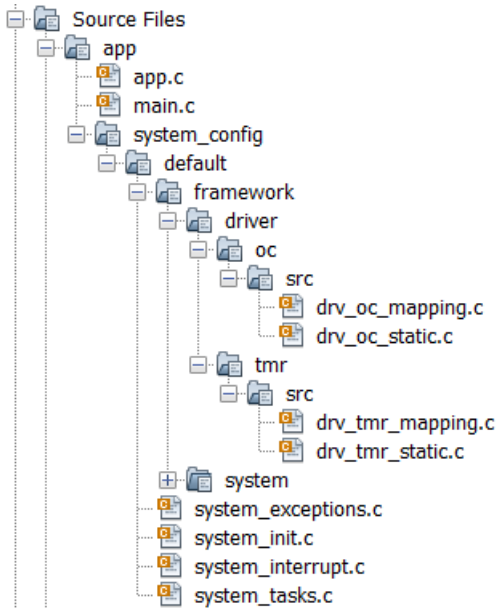
OC_DUAL_COMPARE_CONTINUOUS_PULSE_MODE	Dual Compare, Continuous Pulse mode: Output Compare module output is driven high on compare match with primary compare value and driven low on compare match with secondary compare value. A continuous stream of pulses is generated unless the compare mode is changed 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.

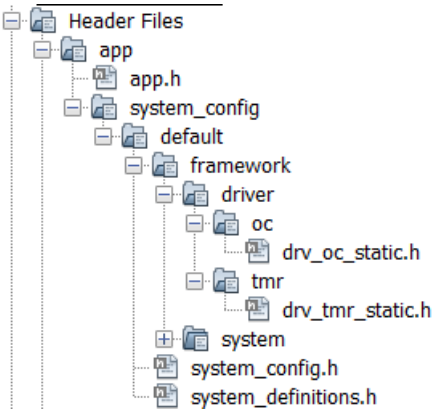
Choix de OC Pulse Width à 4'999, ce qui devrait correspondre à une impulsion de 1 ms car la période du timer 3 de 10 ms correspond à une valeur de 49'999.



5.2. EXPLOITATION DES ÉLÉMENTS GÉNÉRÉS PAR LE MHC

Après la génération suite à la configuration effectuée, nous trouvons l'arborescence suivante dans les Sources Files et les Header Files sous app\system_config.





Le fichier system_init.c contient les appels aux fonctions de configuration des timers et des OC.

5.2.1. LA FONCTION SYS_INITIALIZE

Voici le contenu de la fonction SYS_Initialize que l'on trouve dans le fichier system_init.c :

```
void SYS_Initialize ( void* data )
{
    /* Core Processor Initialization */
    SYS_CLK_Initialize( NULL );
    sysObj.sysDevcon =
              SYS_DEVCON_Initialize(SYS_DEVCON_INDEX_0,
                     (SYS_MODULE_INIT*)&sysDevconInit);
    SYS_DEVCON_PerformanceConfig
                           (SYS_CLK_SystemFrequencyGet());
    SYS_DEVCON_JTAGDisable();
    SYS_PORTS_Initialize();

    /* Board Support Package Initialization */
    BSP_Initialize();

    /* Initialize Drivers */
    /* Initialize the OC Driver */
    DRV_OC0_Initialize();
    DRV_OC1_Initialize();
```

```
    /*Initialize TMR0 */
    DRV_TMR0_Initialize();
    /*Initialize TMR1 */
    DRV_TMR1_Initialize();
    /*Initialize TMR2 */
    DRV_TMR2_Initialize();
    /*Initialize TMR3 */
    DRV_TMR3_Initialize();

    /* Initialize System Services */

    /*** Interrupt Service Initialization Code ***/

       SYS_INT_Initialize();

    /* Initialize Middleware */

    /* Enable Global Interrupts */

    SYS_INT_Enable();

    /* Initialize the Application */
    APP_Initialize();
}
```

Les fonctions d'initialisation des drivers sont automatiquement ajoutées dans la fonction SYS_Initialize.

5.2.2. INITIALISATION DU TIMER 1 (DRV_TMR0)

La fonction DRV_TMR0_Initialize nous permet de découvrir comment sont utilisées les fonctions de PLIB_TMR pour initialiser le timer, ainsi que celles de PLIB_INT pour configurer l'interruption.

La fonction DRV_TMR0_Initialize se trouve dans le fichier drv_tmr_static.c.

```
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,
                TMR_CLOCK_SOURCE_PERIPHERAL_CLOCK);
    /* Select prescalar value */
    PLIB_TMR_PrescaleSelect(TMR_ID_1,
                            TMR_PRESCALE_VALUE_64);
    /* 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, 62499);
```

```
    /* 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);
}
```

☝ La source de l'interruption n'est pas activée. Cette action est réalisée par la fonction _DRV_TMR0_Resume elle-même appelée par la fonction DRV_TMR0_Start

```
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;
}
```

Le timer est stoppé au début de la fonction. Pour lancer le timer, il sera nécessaire dans l'initialisation de l'application d'ajouter un appel à la fonction DRV_TMR0_Start.

5.2.3. INITIALISATION DU TIMER 2 (DRV_TMR1)

La fonction DRV_TMR1_Initialize effectue l'initialisation 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'activation 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;
}
```

Le timer 2 est stoppé au début de la fonction. Pour lancer le timer, il sera nécessaire d'ajouter un appel à la fonction DRV_TMR1_Start dans l'initialisation de l'application.

5.2.4. INITIALISATION DU TIMER 3 (DRV_TMR2)

La fonction DRV_TMR2_Initialize effectue l'initialisation du timer 3.

```
void DRV_TMR2_Initialize(void)
{
    /* Initialize Timer Instance2 */
    /* Disable Timer */
    PLIB_TMR_Stop(TMR_ID_3);
    /* 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);
    /*Set period */
    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'activation de la source d'interruption doit être mise en commentaire dans la fonction _DRV_TMR2_Resume.

```
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;
}
```

Le timer 3 est stoppé au début de la fonction. Pour lancer le timer, il sera nécessaire d'ajouter un appel à la fonction DRV_TMR2_Start dans l'initialisation de l'application.

5.2.5. INITIALISATION DU TIMER 4 (DRV_TMR3)

La fonction DRV_TMR3_Initialize effectue l'initialisation du timer 4, ce qui va configurer la paire de timers 4 & 5.

```
void DRV_TMR3_Initialize(void)
{
    /* Initialize Timer Instance3 */
    /* Disable Timer */
    PLIB_TMR_Stop(TMR_ID_4);
    /* Select clock source */
    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);
    /* Clear counter */
    PLIB_TMR_Counter32BitClear(TMR_ID_4);
    /*Set 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 activée 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;
}
```

Le timer 4 (poids faible) est stoppé au début de la fonction. Pour lancer le timer, il sera nécessaire d'ajouter un appel à la fonction DRV_TMR3_Start dans l'initialisation de l'application.

5.2.6. INITIALISATION DE OC2 (DRV_OC0)

La fonction DRV_OC0_Initialize utilise les fonctions de PLIB_OC pour configurer l'output compare en PWM. Elle se trouve dans le fichier drv_oc_static.c.

```
void DRV_OC0_Initialize(void)
{
    /* Setup OC0 Instance */
    // CHR : A ajouter
    PLIB_OC_Disable(OC_ID_2);

    PLIB_OC_ModeSelect(OC_ID_2,
                OC_COMPARE_PWM_EDGE_ALIGNED_MODE);
    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);
}
```

☝ Par précaution et par cohérence, il faut ajouter comme 1ère action :

```
    PLIB_OC_Disable(OC_ID_2);
```

Cela nous permet de mieux comprendre la nécessité d'activer l'OC en appelant dans l'initialisation de l'application la fonction DRV_OC0_Enable ou la fonction DRV_OC0_Start.

```
void DRV_OC0_Enable(void)
{
   PLIB_OC_Enable(OC_ID_2);
}

void DRV_OC0_Start(void)
{
   PLIB_OC_Enable(OC_ID_2);
}
```

La suite d'appels des fonctions élémentaires de la PLIB_OC est relativement facile à comprendre. Par la suite, on peut modifier le mode, changer le timer qui sert de base de temps. On utilisera la fonction **PLIB_OC_PulseWidth16BitSet** pour modifier le rapport cyclique du PWM.

5.2.7. INITIALISATION DE OC3 (DRV_OC1)

La fonction DRV_OC1_Initialize utilise les fonctions de PLIB_OC pour configurer l'output compare pour générer une impulsion cyclique. Elle se trouve dans le fichier drv_oc_static.c.

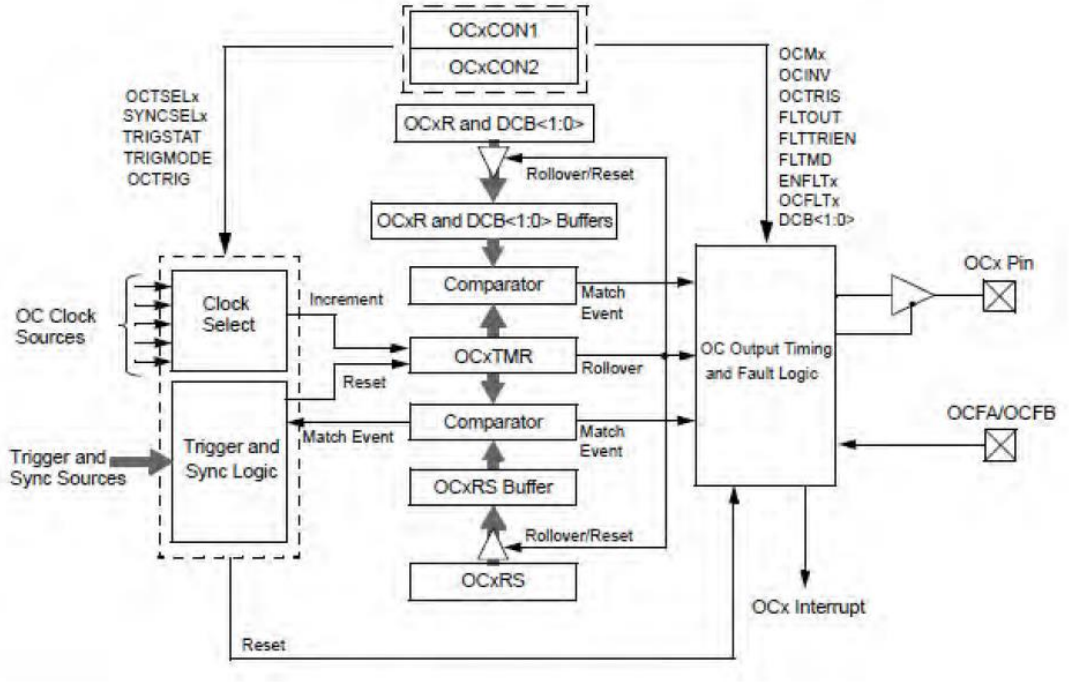
```
void DRV_OC1_Initialize(void)
{
    /* Setup OC1 Instance */
    // CHR : A ajouter
    PLIB_OC_Disable(OC_ID_3);

    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);
    PLIB_OC_Buffer16BitSet(OC_ID_3, 0);
    PLIB_OC_PulseWidth16BitSet(OC_ID_3, 4999);
}
```

☝ A nouveau, par précaution et par cohérence il faut ajouter comme 1ère action :

```
    PLIB_OC_Disable(OC_ID_3);
```

Par la suite, on utilisera les fonctions PLIB_OC_PulseWidth16BitSet et PLIB_OC_Buffer16BitSet pour modifier le rapport cyclique de l'impulsion répétitive.

5.2.8. PRINCIPE DE FONCTIONNEMENT DES OC



Le schéma de principe ci-dessus nous montre qu'il y a 2 comparateurs gérant la sortie. Dans les modes 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 correspond au début de la période du timer.
- Avec **PLIB_OC_PulseWidth16BitSet**(OC_ID_2, 3999), on établit la valeur de référence du comparateur pour le flanc descendant. Dans cet exemple, 3999 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.**

5.3. RÉALISATION DU TEST DES TIMERS & OC

Nous allons ajouter le code minimal permettant de tester la période des timers, ainsi que des OC.

5.3.1. AJOUT DANS INITIALISATION DE L'APPLICATION

Ajout des appels aux fonctions pour démarrer les timers et les OC que l'on place dans APP_Initialize. On ajoute aussi l'initialisation de l'afficheur LCD et l'affichage d'un message.

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

    // Init du LCD
    lcd_init();
    lcd_bl_on();
    // Affichage d'un message
    printf_lcd("App chap5_TimerPwm ");
    lcd_gotoxy(1,2);
    printf_lcd("C. Huber 30.11.2016");

    DRV_TMR0_Start();
    DRV_TMR1_Start();
    DRV_TMR2_Start();
    DRV_TMR3_Start();
    DRV_OC0_Start();
    DRV_OC1_Start();
}
```

5.3.2. AJOUT ACTION LED DANS RÉPONSES AUX INTERRUPTIONS

Les routines de réponses aux interruptions se trouvent dans le fichier system_interrupt.c

5.3.2.1. ACTION LED DANS INTERRUPTION TIMER 1

Dans l'ISR du timer 1 (période 50 ms), on ajoute le toggle de la LED_1.

```
void __ISR(_TIMER_1_VECTOR, ipl3AUTO)
                            _IntHandlerDrvTmrInstance0(void)
{
    PLIB_INT_SourceFlagClear(INT_ID_0,INT_SOURCE_TIMER_1);
    BSP_LEDToggle(BSP_LED_1);
}
```

Avec une inversion toutes les 50 ms, on obtient un signal d'une période de 100 ms.

5.3.2.2. ACTION LED DANS INTERRUPTION TIMER 5

Dans l'ISR du timer 5 (période 500 ms), on ajoute le toggle de la LED_5.

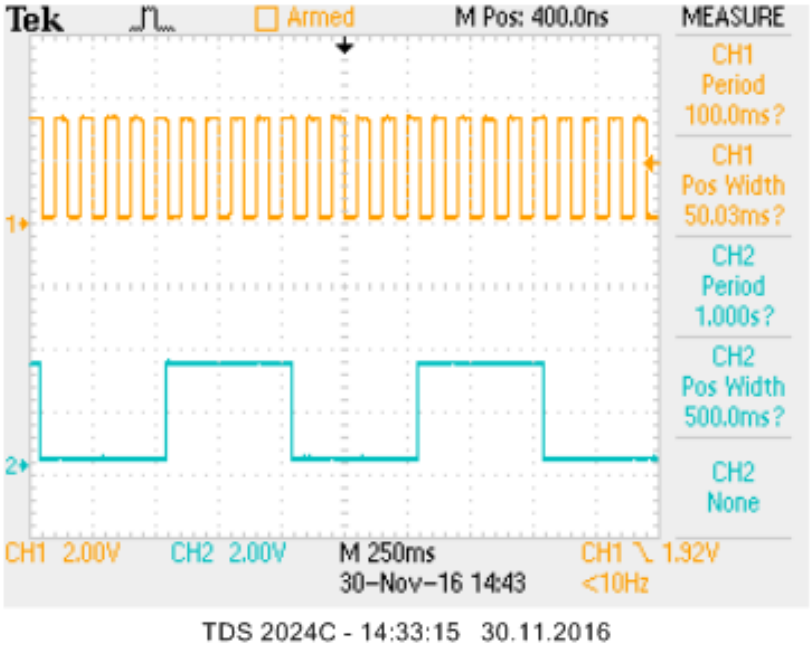
```
void __ISR(_TIMER_5_VECTOR, ipl2AUTO)
                            _IntHandlerDrvTmrInstance3(void)
{
    PLIB_INT_SourceFlagClear(INT_ID_0,INT_SOURCE_TIMER_5);
    BSP_LEDToggle(BSP_LED_5);
}
```

Avec une inversion toutes les 500 ms, on obtient un signal d'une période de 1000 ms.

5.3.3. OBSERVATION DES CYCLES SUR LES LED

Canal 1 :
Led1

Canal 2 :
LED5



On obtient bien une période de 100 ms pour la LED1 et une période de 1000 ms pour la LED5.

5.3.4. TEST FONCTIONNEMENT PWM SUR OC2

☺ Il n'y a rien à faire d'autre que de démarrer les modules initialisés :

```
DRV_TMR1_Start();
DRV_OC0_Start();
```

Pour l'OC2 en mode PWM et avec l'utilisation du timer 2 :

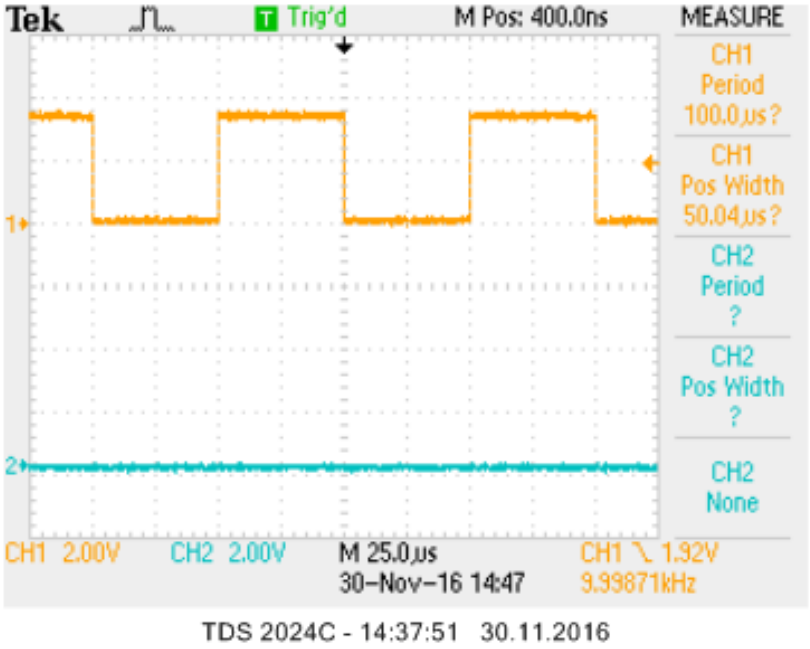
```
PLIB_OC_ModeSelect(OC_ID_2,
                   OC_COMPARE_PWM_EDGE_ALIGNED_MODE);
PLIB_OC_TimerSelect(OC_ID_2, OC_TIMER_16BIT_TMR2);
```

La configuration des 2 références de comparaison est :

```
PLIB_OC_Buffer16BitSet(OC_ID_2, 0);
PLIB_OC_PulseWidth16BitSet(OC_ID_2, 3999);
```

La période du timer 2 étant de 8000 ticks pour 100 µs, nous devons obtenir un signal d'une période de 100 µs et d'un rapport cyclique de 50 %.

Canal 1 :
Broche 76
OC2/RD1



Ce que la mesure à l'oscilloscope nous permet de vérifier.

Si on écrit PLIB_OC_Buffer16BitSet(OC_ID_2, 2000), au lieu de 0, cela n'a aucun effet sur le signal (ceci est valable pour le mode PWM).

5.3.5. TEST FONCTIONNEMENT IMPULSION OC3

☺ A nouveau, il n'y a rien à faire d'autre que de démarrer les modules initialisés :

```
DRV_TMR2_Start();
DRV_OC1_Start();
```

Pour l'OC3 en mode DUAL_COMPARE_CONTINUOUS_PULSE et avec l'utilisation du timer 3 :

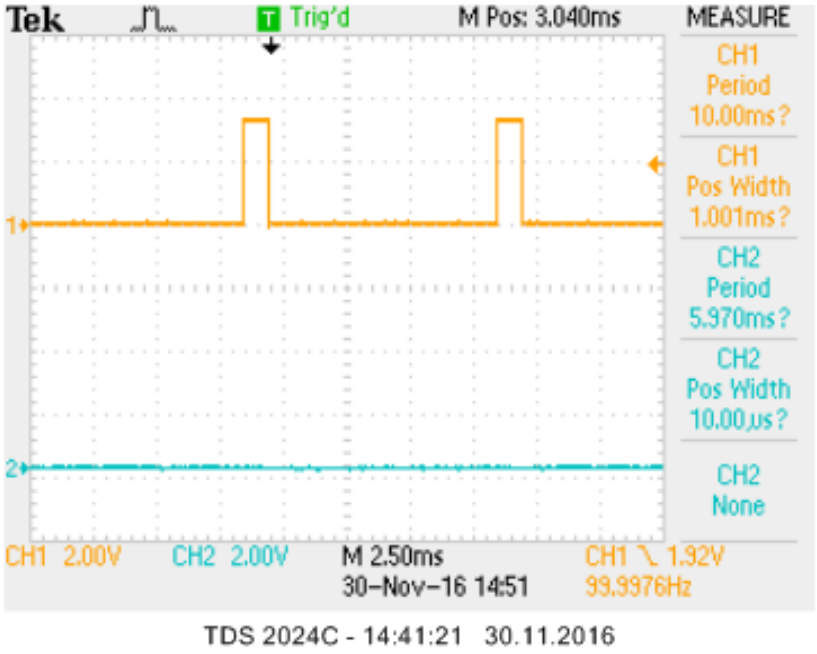
```
PLIB_OC_ModeSelect(OC_ID_3,
                   OC_DUAL_COMPARE_CONTINUOUS_PULSE_MODE);
PLIB_OC_TimerSelect(OC_ID_3, OC_TIMER_16BIT_TMR3);
```

La configuration des 2 références de comparaison est :

```
PLIB_OC_Buffer16BitSet(OC_ID_3, 0);
PLIB_OC_PulseWidth16BitSet(OC_ID_3, 4999);
```

La période du timer 3 étant de 49'999, nous devons obtenir un signal d'une période de 10 ms, avec une durée du temps haut de 1 ms soit 10% de la période du timer 3.

Canal 1 :
Broche 77
OC3/RD2

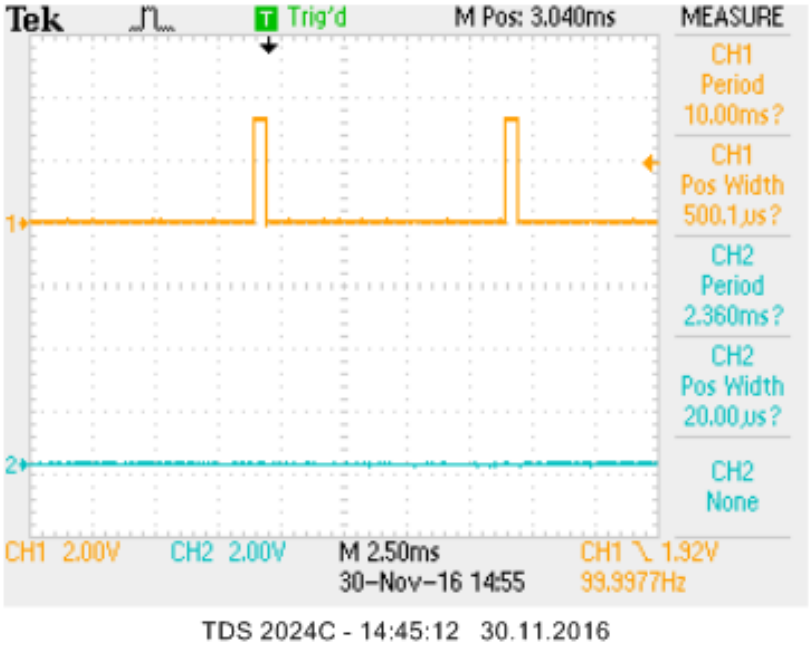


Ce que la mesure à l'oscilloscope nous permet de vérifier.

Si on écrit PLIB_OC_Buffer16BitSet(OC_ID_3, 2499), au lieu de 0, le signal est modifié.

On obtient une impulsion de seulement 500 µs :

Canal 1 :
Broche 77
OC3/RD2



Le flanc montant est retardé de 500 µs d'où un temps haut de 500 µs seulement.

5.4. COMPLÉMENT APPLICATION DE TEST DES TIMERS & OC

Pour tester les éléments mis en place par le MHC, nous allons modifier la machine d'état de l'application pour obtenir un traitement cyclique toutes les 50 ms.

Dans ce traitement cyclique, nous allons effectuer la lecture du pot0 pour varier le PWM de 0 à 100%. La lecture du pot1 va nous permettre de varier la largeur d'impulsion de 0,8 ms à 2,2 ms. Ceci dans le but de piloter un servomoteur de modélisme.

Dans la réponse à l'interruption du timer 32 bits, nous conservons l'inversion de la LED5 pour pouvoir vérifier la période. Dans celle du timer 1, nous conservons l'inversion de la LED1 et nous ajoutons l'activation de l'application à chaque 50 ms.

5.4.1. ADAPTATION INTERRUPTION TIMER 1

Complément de ISR du timer 1 au niveau du fichier system_interrupt.c, dans le but d'établir l'état de l'application à APP_STATE_SERVICE_TASKS toutes les 50 ms. Toggle de la LED_1 à chaque interruption.

```
void __ISR(_TIMER_1_VECTOR, ipl3AUTO)
                            _IntHandlerDrvTmrInstance0(void)
{
    PLIB_INT_SourceFlagClear(INT_ID_0,INT_SOURCE_TIMER_1);
    BSP_LEDToggle(BSP_LED_1);
    // Etablit état d'exécution
    APP_UpdateState (APP_STATE_SERVICE_TASKS);
}
```

5.4.2. MACHINE D'ÉTAT DE L'APPLICATION

Ajout de l'état APP_STATE_WAIT ainsi que le prototype de la fonction:

```
void APP_UpdateState ( APP_STATES NewState ) ;
```

Ceci dans app.h. Implémentation de la fonction dans app.c.

Nous ajoutons encore dans la structure :

```
typedef struct
{
    /* The application's current state */
    APP_STATES state;
    /*  TODO:  Define  any  additional  data  used  by  the
application. */
    S_ADCResults AdcRes;
    int16_t PulseWidthOC2;
    int16_t PulseWidthOC3;
} APP_DATA;
```

Il est nécessaire d'ajouter dans app.h

```
#include "Mc32DriverAdc.h"
```

5.4.3. DESCRIPTION DE APP_TASK

Voici le contenu de la fonction APP_Tasks au niveau du fichier app.c.
Remarque: avec l'introduction de la machine d'état, on déplace l'activation des timers et OC dans la section INIT du switch. D'où :

```
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 AD mode scan
            BSP_InitADC10();

            printf_lcd("App chap5_TimerPwm ");
            lcd_gotoxy(1,2);
            printf_lcd("C. Huber 30.11.2016");
            DRV_TMR0_Start();
            DRV_TMR1_Start();
            DRV_TMR2_Start();
            DRV_TMR3_Start();
            DRV_OC0_Start();
            DRV_OC1_Start();

            appData.state = APP_STATE_WAIT;
            break;
        }

        case APP_STATE_WAIT:
        {
            break;
        }

        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 (pér. 10 ms = 50'000)
                // 0.8 ms => 3999 (0 a 3999)
                // 2.2 - 0.8 = 1.4 => 7000
                appData.PulseWidthOC3 = 3999 +
                       ((7000 * appData.AdcRes.Chan1) / 1024);
                PLIB_OC_PulseWidth16BitSet(OC_ID_3,
                                          appData.PulseWidthOC3);

                lcd_gotoxy(1,4);
                printf_lcd("OC2 %5d OC2 %5d",
                           appData.PulseWidthOC2,
                           appData.PulseWidthOC3);
                appData.state = APP_STATE_WAIT;
                break;
            }

            /* The default state should never be executed. */
            default:
            {
                /* TODO: Handle error in application's
                  state machine. */
                break;
            }
        }
    }
```

5.4.3.1. DÉTAILS MODULATION DES PWM

La valeur brute du canal 0 de l'AD sert à calculer une proportion de la période du timer 2 (7'999). Cette valeur est fournie à la fonction PLIB_OC_PulseWidth16BitSet.

☺ Pour éviter d'utiliser la valeur numérique de la période, il est possible d'utiliser la fonction DRV_TMR1_PeriodValueGet.

```
uint32_t DRV_TMR1_PeriodValueGet(void)
{
    /* Get 16-bit counter value*/
    return (uint32_t) PLIB_TMR_Period16BitGet(TMR_ID_2);
}
```

☝ Il est aussi possible d'utiliser directement PLIB_TMR_Period16BitGet, mais dans ce cas il sera nécessaire d'inclure :

```
#include "peripheral/tmr/plib_tmr.h"
```

```
// Modulation PWM OC2
appData.PulseWidthOC2 =( DRV_TMR1_PeriodValueGet() *
                         appData.AdcRes.Chan0 / 1024);
PLIB_OC_PulseWidth16BitSet(OC_ID_2,
                           appData.PulseWidthOC2);
```

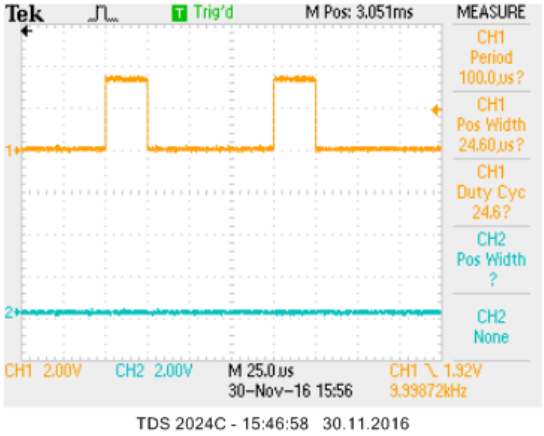
☺ Les calculs étant effectués en 32 bits, on ne rencontre pas de problème de dépassement lors des multiplications, même si AdcRes.Chan0 est 16 bits.

☝ L'appel à PLIB_OC_PulseWidth16BitSet nécessite d'inclure :

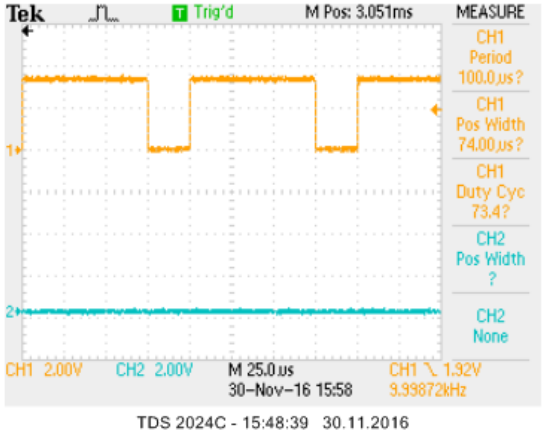
```
#include "peripheral/oc/plib_oc.h"
```

5.4.3.2. RÉSULTAT MODULATION DU PWM

Avec Ch0 = 253



Avec Ch0 = 759



- Avec Ch0 = 253, cela correspond environ à 25%. La valeur PulseWidthOC2 est de 7999 * 253 / 1024 = 1976.
- Avec Ch0 = 759, cela correspond environ à 75%, la valeur PulseWidthOC2 est de 7999 * 759 / 1024 = 5928.

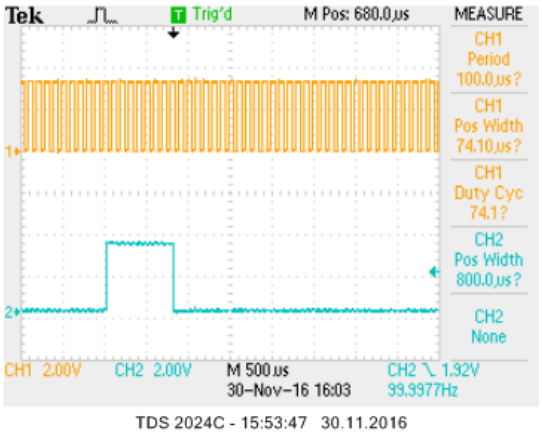
5.4.3.3. DÉTAILS MODULATION DE L'IMPULSION

La période du timer 3 de 10 ms correspond à une valeur de 49'999. Pour obtenir une impulsion qui varie de 0,8 ms à 2,2 ms, on ajoute à la valeur qui correspond à 0,8 ms (3999) une proportion de la valeur qui correspond à la plage de variation de 1,4 ms (7000). La valeur brute du canal 1 de l'AD est utilisée pour établir cette proportion. Puis on applique cette valeur à la fonction PLIB_OC_PulseWidth16BitSet.

```
// Modulation PWM OC3
// 1 ms correspond à 5000 (pér. 10 ms = 50'000)
// 0.8 ms => 3999 (0 a 3999)
// 2.2 - 0.8 = 1.4 => 7000
PulseWidthOC3 = 3999 + ((7000 * AdcRes.Chan1) / 1024);
PLIB_OC_PulseWidth16BitSet(OC_ID_3, PulseWidthOC3);
```

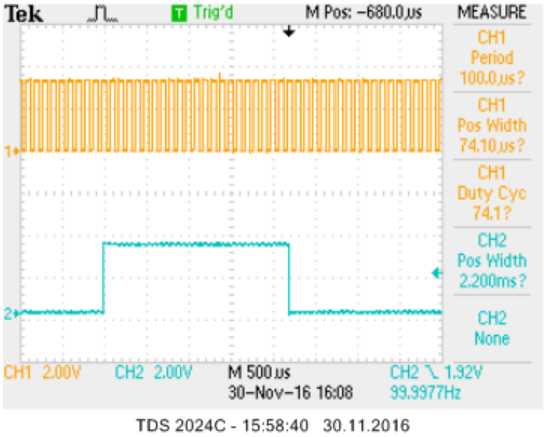
5.4.3.4. RÉSULTAT MODULATION DE L'IMPULSION

Avec Ch1 = 0



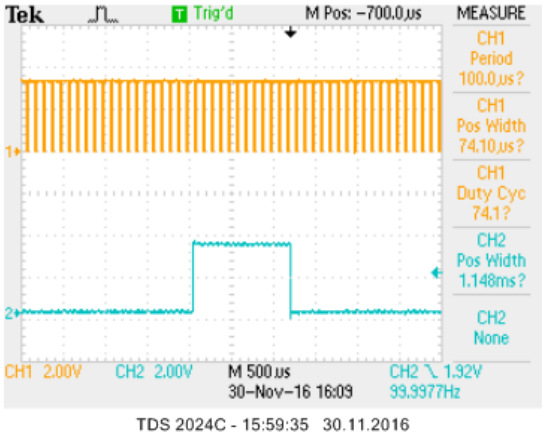
Avec Ch1 = 0, on doit obtenir une impulsion de 0,8 + (1,4 * 0) / 1024 = 0,8 ms, soit une valeur de 3999 pour OC2

Avec Ch1 = 1023



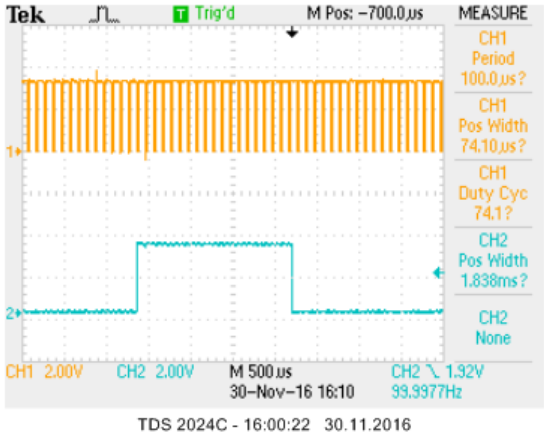
Avec Ch1 = 1023, on doit obtenir une impulsion de 0,8 + (1,4 * 1023) / 1024 = 2,199 ms, soit 3999 + (7000 * 1023) / 1024 = 10992

Avec Ch1 = 254



Avec Ch1 = 254, on doit obtenir une impulsion de 0,8 + (1,4 * 254) / 1024 = 1,147 ms

Avec Ch1 = 758



Avec Ch1 = 758, on doit obtenir une impulsion de 0,8 + (1,4 * 758) / 1024 = 1,836 ms

Mis à part la difficulté à obtenir une valeur stable sur l'ADC, on constate que l'on peut facilement varier les 2 PWM et que l'on obtient de manière assez proche les valeurs prévues.

5.5. CONCLUSION

Cette approche basée sur le MHC, qui correspond à une approche de type "recette de cuisine", nous permet déjà de mettre en œuvre les timers et de générer des signaux PWM. C'est au niveau de la théorie que seront étudiés plus en détail les mécanismes et principes de fonctionnement.

5.6. HISTORIQUE DES VERSIONS

5.6.1. V1.0 JUIN 2013

Création du document.

5.6.2. V1.1 MARS 2014

Quelques retouches mineures. (Changement numérotation des chapitres de la doc du XC32 et avec le kit version B on utilise un PIC32MX795F512L).

5.6.3. V1.5 NOVEMBRE 2014

Remaniement en profondeur pour étudier les nouvelles PLIB de Harmony 1.0 en utilisant le MHC (Microchip Harmony Configurator).

5.6.4. V1.6 NOVEMBRE 2015

Adaptation aux changements de détails (en particulier pour les drivers) introduits par Harmony 1.06. Restructuration des exemples.

5.6.5. V1.7 NOVEMBRE 2016

Adaptation aux changements de détails (en particulier pour les drivers) introduits par Harmony 1.08.

5.6.6. V1.8 NOVEMBRE 2017

Reprise et relecture par SCA.

5.6.7. V1.81 DÉCEMBRE 2018

Relecture et corrections modes OC en relation avec Harmony 2.05 et calculs valeurs OC.