les écoles supérieures

MINF
Programmation des PIC32MX

Chapitre 10

Programmation concurrente

Théorie PIC32MX

Christian HUBER (CHR)
Serge CASTOLDI (SCA)
Version 1.91 janvier 2021

CONTENU DU CHAPITRE 10

10. PROGRAMMATION CONCURRENTE (PIC32MX)

Ce chapitre traite de la programmation concurrente au niveau du PIC32MX et des possibilités offertes par Harmony. Deux approches seront étudiées :

- Les fonctions à machine d'état.
- L'usage d'un noyau temps réel (FreeRTOS).

10.1. BESOIN DE LA PROGRAMMATION CONCURRENTE

Pour mieux comprendre le besoin de la programmation concurrente et comment la réaliser, voici un exemple pratique qui sera utilisé tout au long de ce chapitre :

Supposons que l'on souhaite gérer trois mouvements d'un robot. Une programmation conventionnelle, donc séquentielle, nous amènera à gérer l'un après l'autre chaque mouvement. Par contre si on désire une exécution simultanée des 3 mouvements, il sera nécessaire de recourir à la programmation concurrente.

Remarque : Il est tout de même possible de traiter avec un seul morceau de programme les 3 mouvements en même temps, mais ce morceau de programme devient délicat à écrire. De plus, il résistera mal aux modifications comme l'ajout d'un 4ème axe.

10.2. LES FONCTIONS A MACHINE D'ETAT

Les fonctions à machine d'état sont des fonctions appelées cycliquement. Chaque machine possède une variable donnant son état. **En principe, ces fonctions ne doivent pas être bloquantes, ni effectuer des attentes par des boucles ou des fonctions de délais**. Les attentes sont réalisées par comptage des appels ou lecture d'un timer. Les fonctions à machine d'état utilisent généralement une structure switch .. case en utilisant et en faisant évoluer la valeur de la variable de situation (variable d'état).

10.2.1. GESTION D'UN MOUVEMENT, FONCTION A MACHINE D'ETAT

Voici tout d'abord la description du principe d'un mouvement :

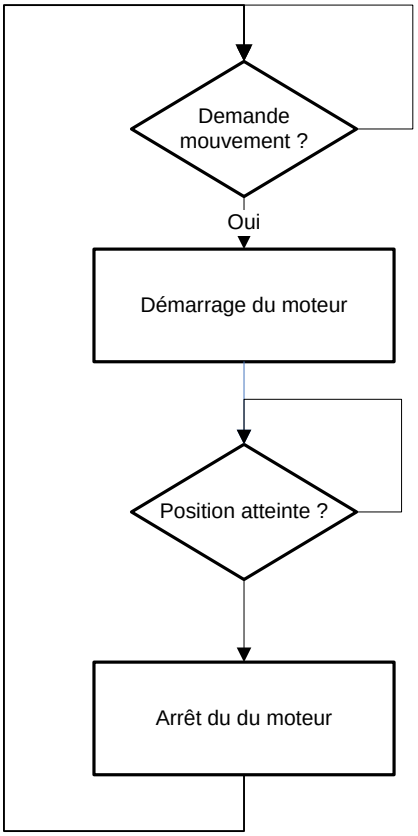


Figure 10-1

10.2.1.1. PRINCIPE GESTION DU MOUVEMENT, MACHINE D'ETAT

Le principe de gestion du mouvement devient le suivant avec une machine d'état :

StatusMove correspond au statut du mouvement (0 à l'arrêt, 1 en mouvement).

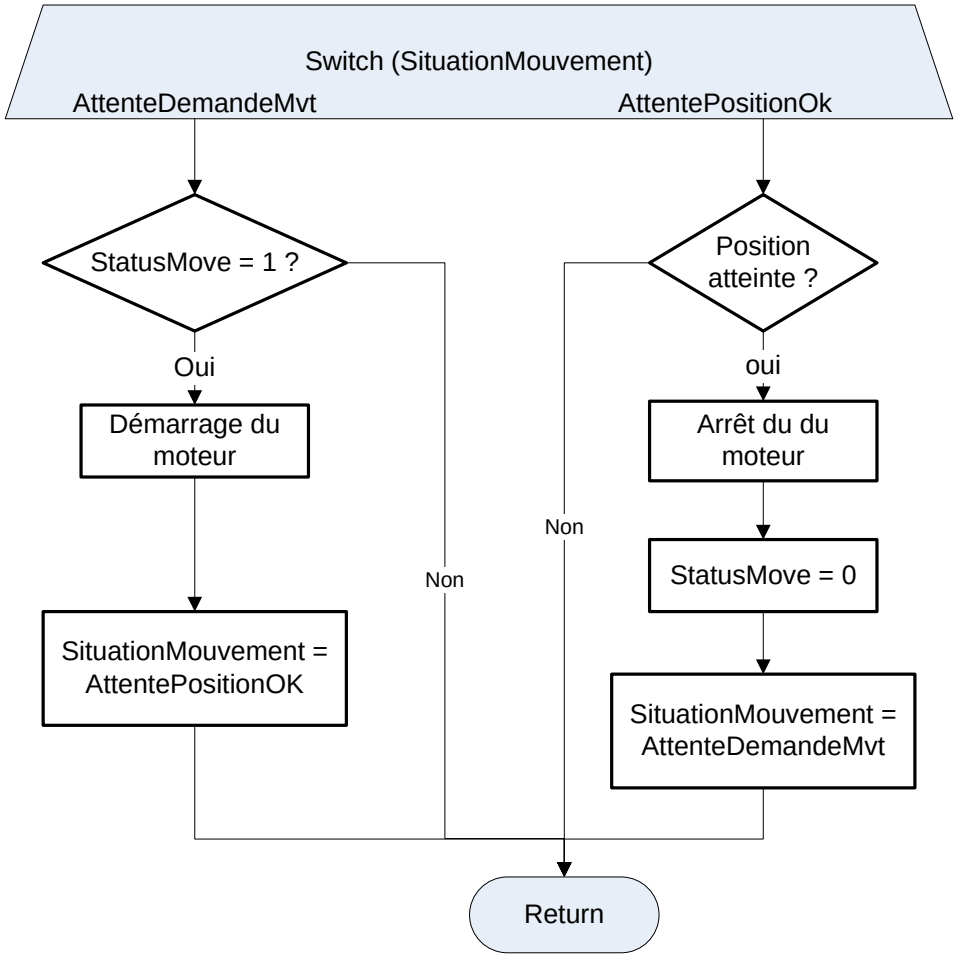


Figure 10-2

Pour démarrer le mouvement, on met à 1 le bit StartMove (action extérieure à la fonction). Il faut attendre StartMove = 1 indication de la fin du mouvement.

Les attentes sont réalisées par des tests qui ressortent de la fonction sans attentes.

10.2.2. EXEMPLE DE REALISATION DE GESTION DE 3 MOUVEMENTS

Le programme suivant illustre la gestion de 3 mouvements en réalisant une fonction unique, appelée 3 fois en utilisant un descripteur pour chaque mouvement. L'exemple simplifie la réalité en simulant la rotation du moteur par un signal généré par un OC qui simule les impulsions que fournirait un capteur incrémental. La position du moteur est gérée par comptage dans des interruptions externes. Pour simplifier, on considère de plus que le moteur tourne toujours dans le même sens.

La fonction ExecMove reprend le principe de la Figure 10-2.

10.2.2.1. FONCTIONS DE GESTION DU MOTEUR

Les fonctions et définitions permettant de simuler la gestion du moteur sont regroupées dans les fichiers GesMot.h et GesMot.c

10.2.2.1.1. Contenu du fichier GesMot.h

```
#ifndef GESMOT_H
#define GESMOT_H

#include <stdint.h>
#include <stdbool.h>

typedef enum { AttenteDemandeMvt, AttentePositionOK}
              E_MvtSituation;

// Descripteur d'un mouvement
typedef struct {
      int8_t              NoMvt;
      bool                StartToDo;
      E_MvtSituation      MvtSituation;
      int32_t             CurPosition;
      int32_t             GoalPosition;
} S_DescrMvt;

// Cette fonction initialise le descripteur de mouvement
void MvtInit(S_DescrMvt *pDescr, int8_t NoMvt);

// Fonction pour exécution d'un mouvement
void RelMove(S_DescrMvt *pDescr, int32_t Pulses);

// Lancement et arrêt du moteur (simulation)
void  StartMot(int8_t  NoMvt) ;
void  StopMot(int8_t  NoMvt) ;

// Fonction de gestion des mouvements (appel cyclique)
void ExecMove(S_DescrMvt *pDescr);

// Fonctions pour gestion de la position
bool    IsInPosition( S_DescrMvt *pDescr);
int32_t GetPosition( S_DescrMvt *pDescr);
void    IncPosition (S_DescrMvt *pDescr);

#endif
```

10.2.2.2. FONCTION D'INITIALISATION DES MOUVEMENTS

Cette fonction initialise le descripteur de mouvement.

```
void MvtInit(S_DescrMvt *pDescr, int8_t NoMvt)
{
   pDescr->NoMvt = NoMvt;
   pDescr->StartToDo = false;
   pDescr->MvtSituation = AttenteDemandeMvt;
   pDescr->CurPosition = 0;
   pDescr->GoalPosition = 0;
}
```

10.2.2.3. FONCTION D'EXECUTION D'UN DEPLACEMENT RELATIF

Cette fonction établit la position de destination dans le descripteur et met à un le bit de demande de mouvement.

```
void RelMove(S_DescrMvt *pDescr, int32_t DeltaPos)
{
   // Prépare le mouvement
   pDescr->GoalPosition =  pDescr->CurPosition + DeltaPos;
   pDescr->StartToDo = true;
}
```

10.2.2.4. FONCTIONS SIMULANT LE START ET STOP DU MOTEUR

Ces fonctions simulent le lancement et l'arrêt d'un moteur en agissant sur l'OC correspondant.

```
// Lance le moteur (simulation)
// Corresponds à activer l'OC correspondant
void  StartMot(int8_t  NoMvt)
{
     switch (NoMvt) {
         case 1 : DRV_OC0_Start(); break;
         case 2 : DRV_OC1_Start(); break;
         case 3 : DRV_OC2_Start(); break;
     }
}

// stop le moteur (simulation)
void  StopMot(int8_t  NoMvt)
{
     switch (NoMvt) {
         case 1 : DRV_OC0_Stop(); break;
         case 2 : DRV_OC1_Stop(); break;
         case 3 : DRV_OC2_Stop(); break;
     }
}
```

10.2.2.5. FONCTIONS POUR LA GESTION DE LA POSITION

Ces fonctions sont prévues pour être utilisées depuis l'application. Elles exploitent le contenu du descripteur.

```
bool IsInPosition( S_DescrMvt *pDescr)
{
    bool stat = false;
    if ( pDescr->MvtSituation == AttenteDemandeMvt)
           stat = true;
    return stat;
}

int32_t GetPosition( S_DescrMvt *pDescr)
{
    return pDescr->CurPosition;
}

void IncPosition (S_DescrMvt *pDescr)
{
    pDescr->CurPosition++;
}
```

10.2.2.6. FONCTION D'EXECUTION DES MOUVEMENTS

Cette fonction, prévue pour un appel cyclique, assure l'exécution d'un mouvement.

```
void ExecMove(S_DescrMvt *pDescr)
{
    switch (pDescr->MvtSituation) {
      case AttenteDemandeMvt :
         if ( pDescr->StartToDo) {
             pDescr->StartToDo = false;
            // Demande rotation
            StartMot( pDescr->NoMvt) ;
            pDescr->MvtSituation = AttentePositionOK;
         }
      break;
      case AttentePositionOK:
         // Test si position atteinte
         // (la position augmente toujours)
         if (pDescr->CurPosition >= pDescr->GoalPosition) {
              // Position atteinte
              // Stop la rotation
              StopMot( pDescr->NoMvt) ;
              pDescr->MvtSituation = AttenteDemandeMvt;
         }
      break;
   } // end switch
} // end ExecMove
```

10.2.2.7. MISE A JOUR DE LA POSITION (INTERRUPTIONS EXTERNES)

Utilisation de fonctions de l'application.

```
void __ISR(_EXTERNAL_1_VECTOR, IPL5AUTO)
                _IntHandlerExternalInterruptInstance0(void)
{
    PLIB_INT_SourceFlagClear(INT_ID_0,
                             INT_SOURCE_EXTERNAL_1);
    APP_UpdatePos1();
}

void __ISR(_EXTERNAL_2_VECTOR, IPL5AUTO)
                _IntHandlerExternalInterruptInstance1(void)
{
    PLIB_INT_SourceFlagClear(INT_ID_0,
                             INT_SOURCE_EXTERNAL_2);
    APP_UpdatePos2();
}

void __ISR(_EXTERNAL_3_VECTOR, IPL5AUTO)
                _IntHandlerExternalInterruptInstance2(void)
{
    PLIB_INT_SourceFlagClear(INT_ID_0,
                             INT_SOURCE_EXTERNAL_3);
    APP_UpdatePos3();
}
```

10.2.2.8. APPEL CYCLIQUE DES FONCTIONS EXECMOVE

L'appel cyclique des fonctions ExecMove est placé dans la réponse à l'interruption du timer 1 (cycle 1 ms). La fonction de l'application APP_ExecMovements effectue l'appel des 3 fonctions ExecMove.

```
// Réponse interruption Timer1 (cycle = 1 ms)
void __ISR(_TIMER_1_VECTOR, ipl4AUTO)
                          _IntHandlerDrvTmrInstance0(void)
{
    static int count = 0;

    PLIB_INT_SourceFlagClear(INT_ID_0,INT_SOURCE_TIMER_1);
    count++;
    APP_ExecMovements();
    if (count >= 2000) {
        APP_UpdateState(APP_STATE_SERVICE_TASKS);
        count = 1980;
    }
}
```

10.2.2.9. TRAITEMENTS AU NIVEAU DE L'APPLICATION

Voici les différentes actions au niveau de l'application.

10.2.2.9.1. Variable globales

Les trois descripteurs de mouvement et une variable pour la gestion des séquences de mouvements.

```
// Descripteurs de mouvements
S_DescrMvt DescrMot1;
S_DescrMvt DescrMot2;
S_DescrMvt DescrMot3;
// Variable pour gestion séquences de mouvements
int APP_SituationMouvement;
```

10.2.2.9.2. Initialisations

Dans la section case APP_STATE_INIT, on effectue l'initialisation des descripteurs et le lancement des timers. Et aussi ce qui est nécessaire à l'affichage sur le lcd.

```
case APP_STATE_INIT:
     lcd_init();
     lcd_bl_on();

     // Init des descripteurs de mouvements
     MvtInit(&DescrMot1, 1);
     MvtInit(&DescrMot2, 2);
     MvtInit(&DescrMot3, 3);
     APP_SituationMouvement = 0;
     DRV_TMR0_Start(); // Start Timer1 (cycle application)
     DRV_TMR1_Start(); // Start Timer2 (base temps des OC)

     printf_lcd("Chap10 prog concu.");
     lcd_gotoxy(1,2);
     printf_lcd("C. Huber 04.06.2015");

     appData.state = APP_STATE_WAIT;
break;
```

10.2.2.9.3. Exécution des séquences de mouvements

Dans la section case APP_STATE_SERVICE_TASKS, on introduit une machine d'état pour réaliser des séquences de mouvements.

```
case APP_STATE_SERVICE_TASKS:
    // Machine d'état traitement des 3 mouvements
    switch (APP_SituationMouvement) {

        case 0 :
            // Lance 1er deplacement d'ensemble
            // Exécution de trois mouvements
            RelMove(&DescrMot1, 3000);
            RelMove(&DescrMot2, 5000);
            RelMove(&DescrMot3, 7000);
            APP_SituationMouvement = 1;
         break;
```

```
            case 1 :
                // Attente fin des 3 mouvements
                if ( IsInPosition(&DescrMot1) &&
                          IsInPosition(&DescrMot2) &&
                          IsInPosition(&DescrMot3)) {
                     APP_SituationMouvement = 2;
                    // Affiche la position des 3 moteurs
                    lcd_gotoxy(1,3);
                    printf_lcd("M1:%06d M2:%06d ",
                                GetPosition(&DescrMot1),
                                GetPosition(&DescrMot2) );
                    lcd_gotoxy(1,4);
                    printf_lcd("M3:%06d  ",
                                GetPosition(&DescrMot3) );
                 }
            break;

            case 2 :
                // Lance 2ème déplacement d'ensemble
                // Exécution de trois mouvements
                RelMove(&DescrMot1, 500);
                RelMove(&DescrMot2, 4000);
                RelMove(&DescrMot3, 2000);
                APP_SituationMouvement = 3;
            break;

            case 3 :
                // Attente fin des 3 mouvement
                if ( IsInPosition(&DescrMot1) &&
                     IsInPosition(&DescrMot2) &&
                     IsInPosition(&DescrMot3))    {

                      APP_SituationMouvement = 0;
                      // Affiche la position des 3 moteurs
                      lcd_gotoxy(1,3);
                      printf_lcd("M1:%06d M2:%06d ",
                                GetPosition(&DescrMot1),
                                GetPosition(&DescrMot2) );
                      lcd_gotoxy(1,4);
                      printf_lcd("M3:%06d  ",
                                GetPosition(&DescrMot3) );
                      if (GetPosition(&DescrMot3) > 150000 ) {
                                 MvtInit(&DescrMot1, 1);
                                 MvtInit(&DescrMot2, 2);
                                 MvtInit(&DescrMot3, 3);
                      }
                  }
            break;
        }
    appData.state = APP_STATE_WAIT;
    break;
```

10.2.2.10. LA FONCTION APP_EXECMOVEMENTS

Cette fonction, prévue pour un appel cyclique, effectue l'appel des fonctions ExecMove.

```
void APP_ExecMovements(void)
{
      ExecMove(&DescrMot1);
      ExecMove(&DescrMot2);
      ExecMove(&DescrMot3);
}
```

☝ C'est ici que se réalise en quelque sorte l'exécution en parallèle.

10.2.2.11. LES FONCTIONS APP_UPDATEPOSX

Ces fonctions servent d'interface aux fonctions de gestion des moteurs.

```
void APP_UpdatePos1(void)
{
      IncPosition(&DescrMot1);
}

void APP_UpdatePos2(void)
{
      IncPosition(&DescrMot2);
}

void APP_UpdatePos3(void)
{
      IncPosition(&DescrMot3);
}
```

10.2.2.12. TEST DU FONCTIONNEMENT

Il faut câbler les sorties OC sur les Int Ext :

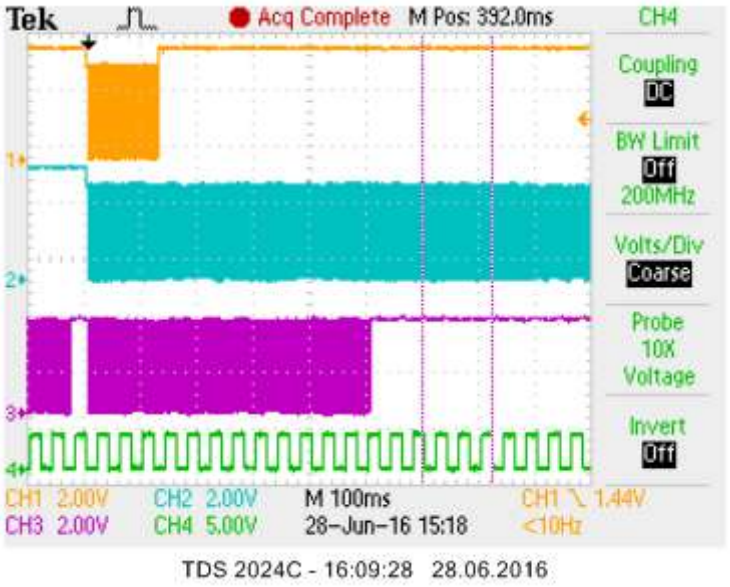
- OC2 → Int1 76 → 18
- OC3 → Int2 77 → 19
- OC4 → Int3 78 → 66

Ch1 : OC2 (moteur 1)

Ch2 : OC3 (moteur 2)

Ch3 : OC4 (moteur 3)

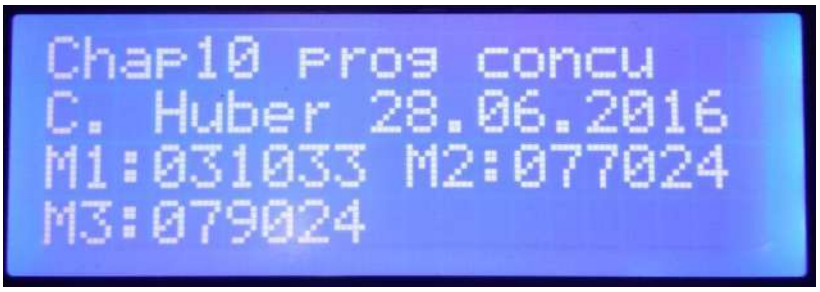
Ch4 : LED_2
(toggle dans application)



On peut observer les séquences de mouvement avec les périodes d'arrêt (signal au niveau haut).

10.2.2.12.1. Respect des positions

Le cycle de traitement des "moteurs" étant de 1 ms et la période des OC de 250 us, il est normal que la position affichée dépasse de quelques impulsions.



10.3. PROGRAMMATION CONCURRENTE PAR MULTI-THREADING

Ce type de programmation requiert l'existence d'un noyau ou système d'exploitation capable de gérer des processus fonctionnant en pseudo parallélisme selon le principe suivant :

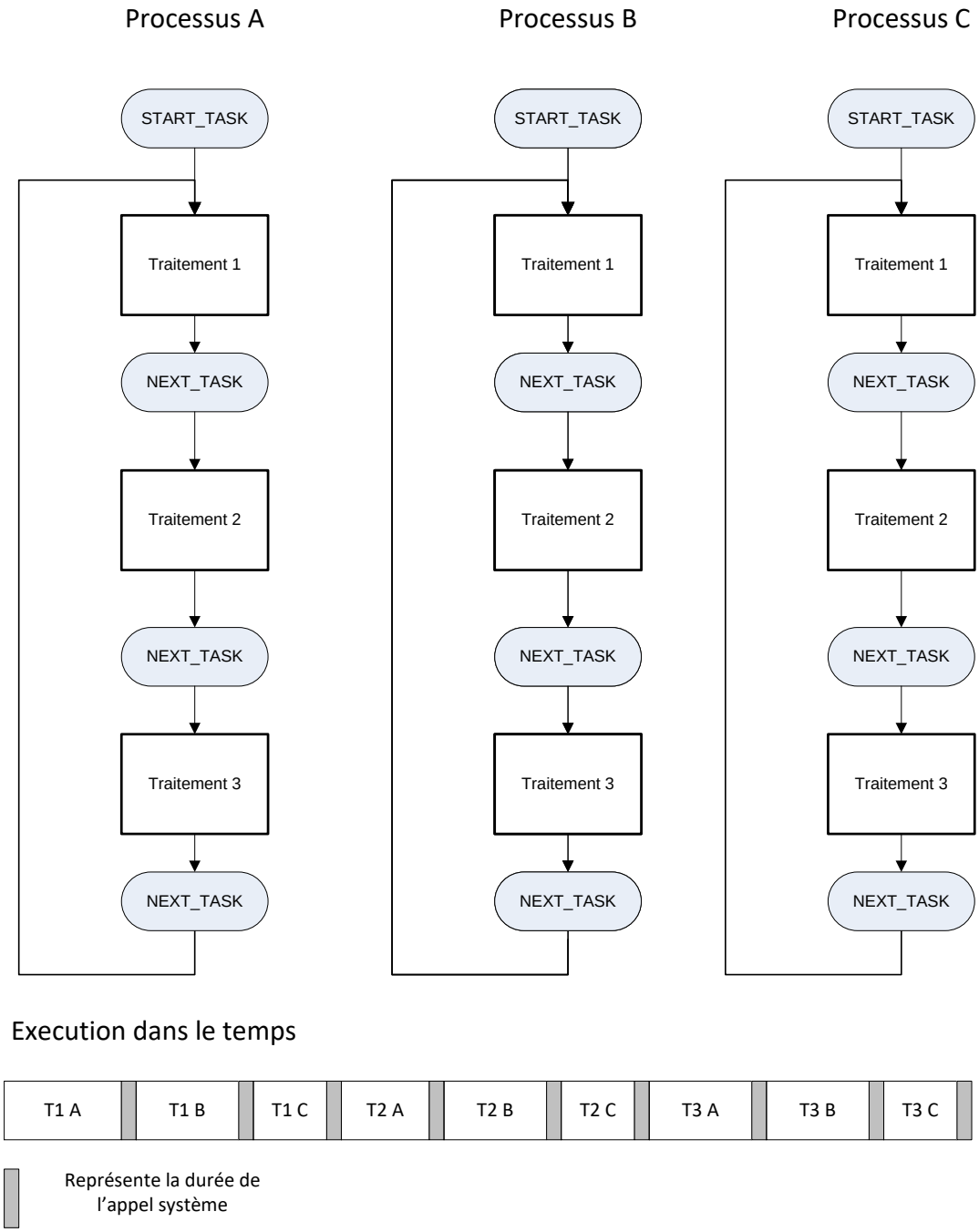


Figure 10-3

Chaque processus s'exécute pour lui-même, mais pour assurer la répartition du temps CPU par tranches, chaque processus doit, lorsqu'une action est accomplie, exécuter un appel au système pour que le processus suivant puisse s'exécuter. Ceci s'appelle un noyau **non-préemptif ou coopératif**.

Il existe aussi des systèmes n'utilisant pas d'appel explicite. Dans ce cas, c'est le système qui attribue le temps CPU d'un processus à l'autre sur la base d'une interruption à intervalle régulier. On appelle cela un noyau **préemptif**.

10.3.1. GESTION DES 3 MOUVEMENTS AVEC MULTI-PROCESSUS

Si on reprend notre exemple de gestion des 3 mouvements dans un cadre multiprocessus, le traitement d'un mouvement par un processus devient :

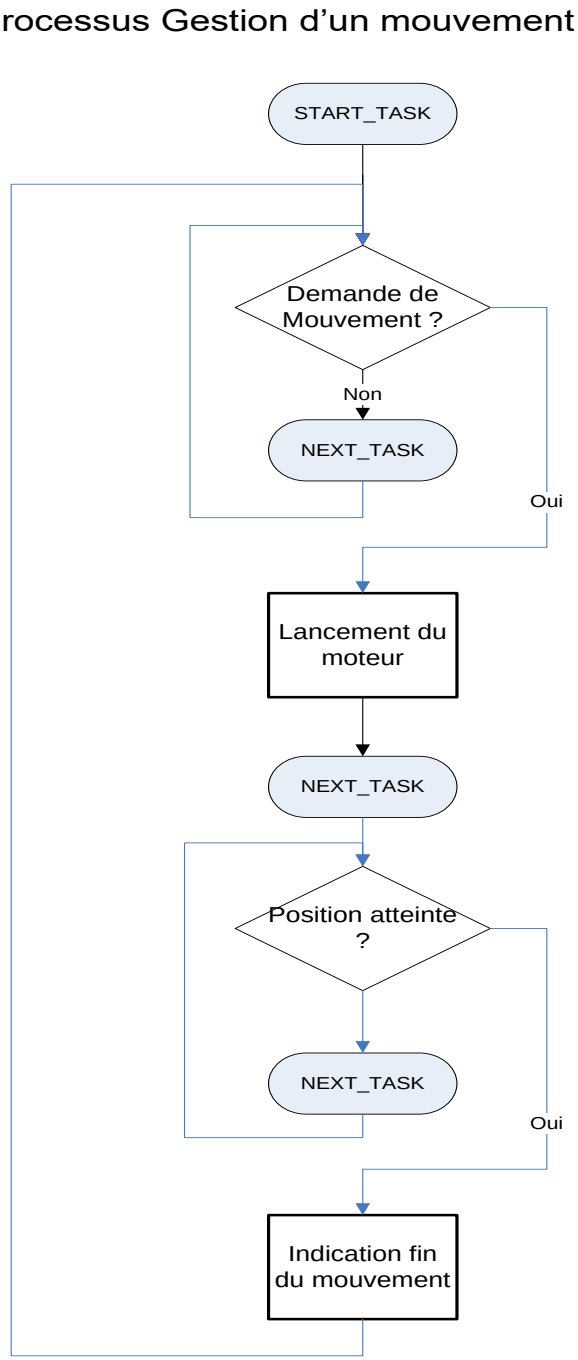


Figure 10-4

On constate que mis à part les appels système (Next_Task), on retrouve une écriture plus naturelle de la séquence.

10.3.2. FREERTOS AVEC HARMONY

Harmony, dans sa section "Third Party", supporte différents RTOS (Real Time Operating System) : notre choix se porte sur FreeRTOS.

FreeRTOS est un système d'exploitation temps réel faible empreinte, portable, préemptif et libre pour microcontrôleur. Il a été porté sur plusieurs dizaines d'architectures différentes.
Il est parmi les plus utilisés dans le marché des systèmes d'exploitation temps réel. Les domaines d'applications sont assez larges, car les principaux avantages de FreeRTOS sont l'exécution temps réel, un code source ouvert et une taille très faible. On le retrouvera dans des systèmes embarqués qui ont des contraintes d'espace pour le code, mais aussi pour des systèmes de traitement vidéo et des applications réseau qui ont des contraintes temps réel.

En étudiant et modifiant une copie du projet
<Répertoire Harmony>\v<n>\apps\rtos\freertos\basic, nous arrivons à la conclusion que l'on peut utiliser vTaskDelay() pour réaliser un équivalent du NEXT_TASK de la Figure 10-4.

Cette partie a été réalisée avec les logiciels suivants :

- Harmony v1_06
- MPLABX IDE v3.10
- XC32 v1.40

10.3.2.1. LA FONCTION VTASKDELAY

Voici le prototype de la fonction vTaskDelay :

```
void vTaskDelay( const TickType_t xTicksToDelay );
```

Et les commentaires qui vont avec :
Delay a task for a given number of ticks. The actual time that the task remains blocked depends on the tick rate. The constant portTICK_PERIOD_MS can be used to calculate real time from the tick rate - with the resolution of one tick period.

vTaskDelay() specifies a time at which the task wishes to unblock **relative to** the time at which vTaskDelay() is called. For example, specifying a block period of 100 ticks will cause the task to unblock 100 ticks after vTaskDelay() is called.

10.3.2.2. LA FUNCTION VTASKDELAYUNTIL

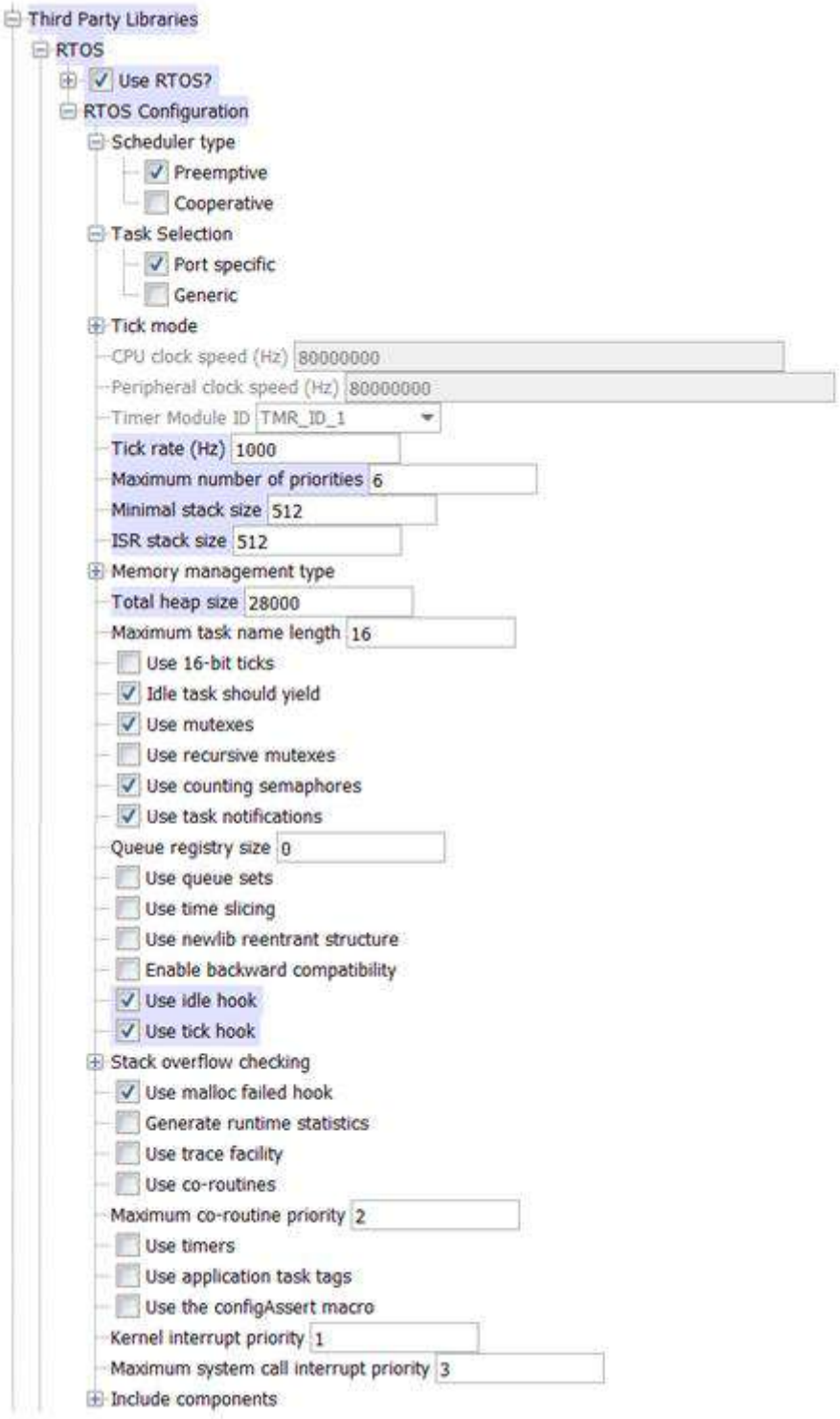
L'utilisation de la fonction vTaskDelay() n'est pas appropriée pour contrôler la fréquence d'exécution d'une tâche périodique. En effet, le temps d'exécution de la tâche elle-même (qui peut être variable), ainsi que les autres tâches et les interruptions vont affecter la fréquence à laquelle vTaskDelay() sera appelée, et par conséquent le prochain instant d'exécution de la tâche.
La fonction **vTaskDelayUntil()** permet une exécution de tâche à fréquence fixe. Ceci est rendu possible en lui transmettant en paramètre un temps absolu (au lieu de relatif pour vTaskDelay()) auquel le délai se terminera (et donc la tâche appelante sera débloquée).

Exemple d'utilisation :

```
// Perform an action every 10 ticks.
void vTaskFunction(void)
{
  TickType_t xLastWakeTime;
  const TickType_t xPeriod = 10;

  //Init. the xLastWakeTime variable with the current time
  xLastWakeTime = xTaskGetTickCount();

  for( ;; )
  {
    // Wait for the next cycle.
    vTaskDelayUntil( &xLastWakeTime, xPeriod );

    // Perform action here.
  }
}
```

10.3.3. AJOUT RTOS DANS LA CONFIGURATION

Au niveau des Third Party Librairies on coche Use RTOS et on configure en s'inspirant de l'exemple.

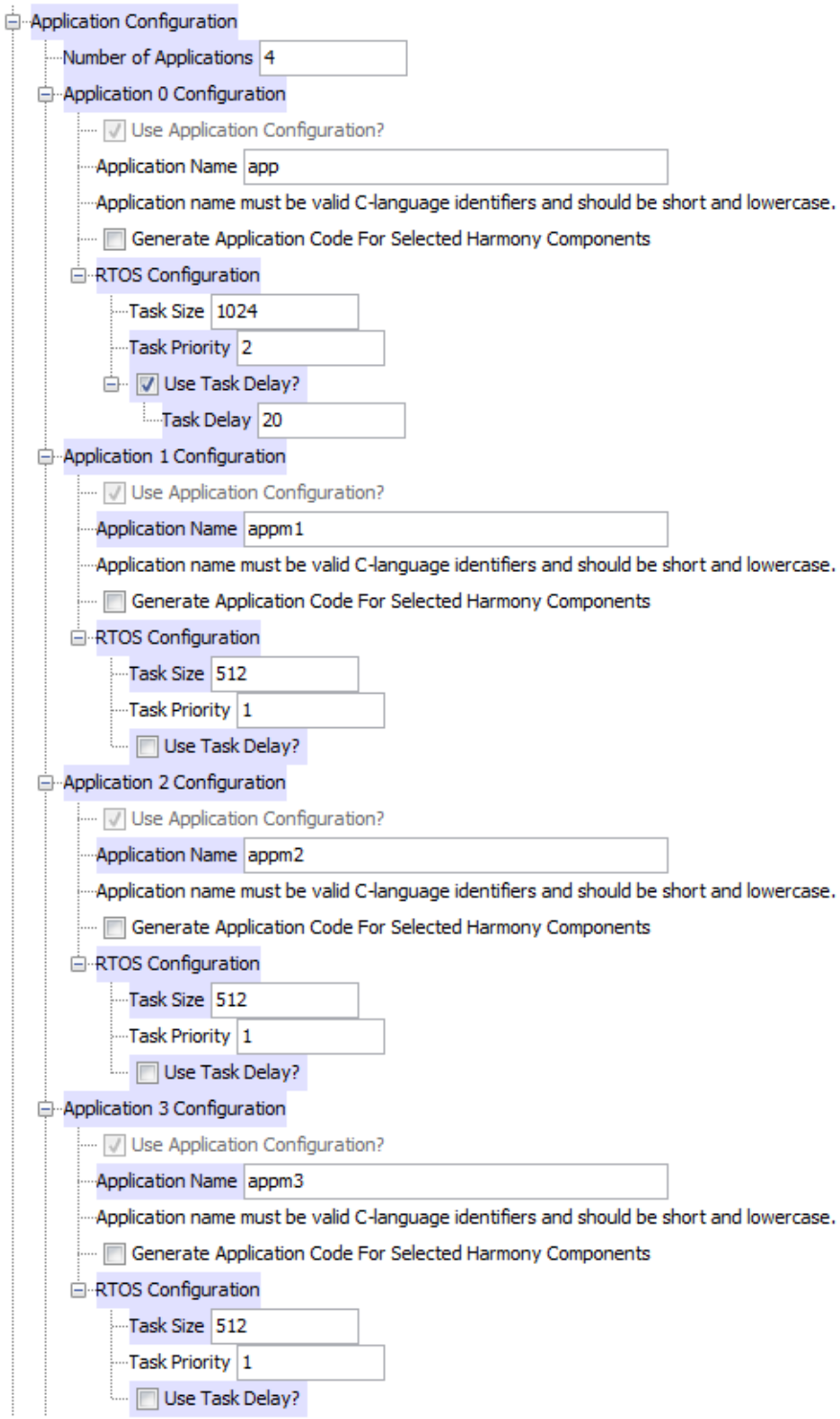


En se documentant, on trouve que le timer 1 est utilisé par FreeRTOS. On ne peut donc pas l'utiliser !

Cette configuration fonctionne.

10.3.3.1. AJOUT DE 3 TACHES SUPPLEMENTAIRES

- Attribution de la priorité 1 aux 3 tâches moteur et 2 à la tâche application.
- Task Delay de valeur 20 ticks pour l'application principale. La configuration de FreeRTOS ci-dessus fixe le tick rate à 1 kHz.



10.3.3.2. SITUATION DE SYS_TASKS

Avec l'ajout du RTOS, on constate un changement dans la gestion des tâches.

```
void SYS_Tasks ( void )
{
    /* Create OS Thread for Sys Tasks. */
    xTaskCreate((TaskFunction_t) _SYS_Tasks,
                "Sys Tasks",
                1024, NULL, 1, NULL);

    /* Create OS Thread for APP Tasks. */
    xTaskCreate((TaskFunction_t) _APP_Tasks,
                "APP Tasks",
                1024, NULL, 1, NULL);

    /* Create OS Thread for APPM1 Tasks. */
    xTaskCreate((TaskFunction_t) _APPM1_Tasks,
                "APPM1 Tasks",
                512, NULL, 1, NULL);

    /* Create OS Thread for APPM2 Tasks. */
    xTaskCreate((TaskFunction_t) _APPM2_Tasks,
                "APPM2 Tasks",
                512, NULL, 1, NULL);

    /* Create OS Thread for APPM3 Tasks. */
    xTaskCreate((TaskFunction_t) _APPM3_Tasks,
                "APPM3 Tasks",
                512, NULL, 1, NULL);

    /**************
     * Start RTOS *
     **************/
    vTaskStartScheduler(); /* This function never returns.
        */
}
```

10.3.3.3. REALISATION DES TACHES

A la suite de SYS_Tasks, on trouve l'implémentation des tâches sous la forme suivante :

```
static void _SYS_Tasks ( void )
{
    while(1)
    {
        /* Maintain system services */
        SYS_DEVCON_Tasks(sysObj.sysDevcon);

        /* Maintain Device Drivers */

        /* Maintain Middleware */

        /* Task Delay */
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

10.3.3.3.1. Appel de APP_Tasks

Voici la situation de APP_Tasks, avec son delay de 20 ms entre chaque appel :

```
static void _APP_Tasks(void)
{
    while(1)
    {
        APP_Tasks();
        vTaskDelay(20 / portTICK_PERIOD_MS);
    }
}
```

10.3.3.3.2. Appel de APPM1_Tasks

Voici la situation de l'appel de **APPM1_Tasks()** avec l'ajout d'une led pour observation :

```
static void _APPM1_Tasks(void)
{
    while(1)
    {
        APPM1_Tasks();
        BSP_LEDToggle(BSP_LED_3);
    }
}
```

10.3.3.3.3. Appel de APPM2_Tasks

Voici la situation de l'appel de **APPM2_Tasks()** :

```
static void _APPM2_Tasks(void)
{
    while(1)
    {
        APPM2_Tasks();
    }
}
```

10.3.3.3.4. Appel de APPM3_Tasks

Voici la situation de l'appel de **APPM3_Tasks() :**

```
static void _APPM3_Tasks(void)
{
    while(1)
    {
        APPM3_Tasks();
    }
}
```

☺ Il va donc être possible d'introduire le traitement du mouvement dans chacune des applications.

10.3.4. REALISATION DES TACHES MOTEURS

Pour la réalisation, nous allons utiliser des fonctions d'attente utilisant vTaskDelay() avec une valeur de 1ms.

10.3.4.1. REALISATION DE APPM1_TASKS

Voici la réalisation de APPM1_Tasks ainsi que la fonction d'initialisation. Le descripteur de mouvement a été placé dans la structure appm1Data.

```
void APPM1_Initialize ( void )
{
    // la variable d'état n'est pas utilisée
    appm1Data.state = APPM1_STATE_INIT;
    MvtInit(&appm1Data.DescrMot1, 1);
}

void APPM1_Tasks ( void )
{
    // Pas besoin de machine d'état
    // ----------------------------

    // Attente start
    while ( appm1Data.DescrMot1.StartToDo == false) {
        vTaskDelay(1 / portTICK_PERIOD_MS);
        BSP_LEDToggle(BSP_LED_1);
    }

    // quittance et annonce MvtEnCours
    appm1Data.DescrMot1.MvtSituation = MvtInProgress;
    appm1Data.DescrMot1.StartToDo = false;
    // Demande rotation
    StartMot( appm1Data.DescrMot1.NoMvt) ;

    // Attente position atteinte
    while ( appm1Data.DescrMot1.CurPosition
                    < appm1Data.DescrMot1.GoalPosition) {
        vTaskDelay(1 / portTICK_PERIOD_MS);
        BSP_LEDToggle(BSP_LED_1);
    }

    // position atteinte
    // Stop la rotation
    StopMot( appm1Data.DescrMot1.NoMvt) ;
    appm1Data.DescrMot1.MvtSituation = PositionReached;
    LED0_W = 0;
    vTaskDelay(5 / portTICK_PERIOD_MS);  // pause

}
```

Pas de machine d'état, mais des boucles avec vTaskDelay.

10.3.4.2. REALISATION DE APPM2_TASKS

Voici la réalisation de APPM2_Tasks ainsi que la fonction d'initialisation. Le descripteur de mouvement a été placé dans la structure appm2Data.

```
void APPM2_Initialize ( void )
{
    // la variable d'état n'est pas utilisée
    appm2Data.state = APPM2_STATE_INIT;
     MvtInit(&appm2Data.DescrMot2, 2);
}

void APPM2_Tasks ( void )
{
   // Attente start
    while ( appm2Data.DescrMot2.StartToDo == false) {
        vTaskDelay(1 / portTICK_PERIOD_MS);
    }

    // quittance et annonce MvtEnCours
    appm2Data.DescrMot2.MvtSituation = MvtInProgress;
    appm2Data.DescrMot2.StartToDo = false;
    // Demande rotation
    StartMot( appm2Data.DescrMot2.NoMvt) ;

    // Attente position atteinte
    while ( appm2Data.DescrMot2.CurPosition
                    < appm2Data.DescrMot2.GoalPosition) {
        vTaskDelay(1 / portTICK_PERIOD_MS);
    }
    // position atteinte
    // Stop la rotation
    StopMot( appm2Data.DescrMot2.NoMvt) ;
    appm2Data.DescrMot2.MvtSituation = PositionReached;

    vTaskDelay(5 / portTICK_PERIOD_MS);  // pause
}
```

10.3.4.3. REALISATION DE APPM3_TASKS

Voici la réalisation de APPM3_Tasks ainsi que la fonction d'initialisation. Le descripteur de mouvement a été placé dans la structure appm3Data.

```
void APPM3_Initialize ( void )
{
    // la variable d'état n'est pas utilisée
    appm3Data.state = APPM3_STATE_INIT;
    MvtInit(&appm3Data.DescrMot3, 3);
}
```

```
void APPM3_Tasks ( void )
{
    // Attente start
    while ( appm3Data.DescrMot3.StartToDo == false) {
        vTaskDelay(1 / portTICK_PERIOD_MS);

    }

    // quittance et annonce MvtEnCours
    appm3Data.DescrMot3.MvtSituation = MvtInProgress;
    appm3Data.DescrMot3.StartToDo = false;
    // Demande rotation
    StartMot( appm3Data.DescrMot3.NoMvt) ;

    // Attente position atteinte
    while ( appm3Data.DescrMot3.CurPosition
                    < appm3Data.DescrMot3.GoalPosition) {
        vTaskDelay(1 / portTICK_PERIOD_MS);
    }
    // position atteinte
    // Stop la rotation
    StopMot( appm3Data.DescrMot3.NoMvt) ;
    appm3Data.DescrMot3.MvtSituation = PositionReached;

    vTaskDelay(5 / portTICK_PERIOD_MS);  // pause

}
```

10.3.5. MODIFICATION DE GESMOT

Au niveau de GesMot.h, modification du type énuméré pour donner une notion de situation du mouvement :

```
typedef enum { MvtInProgress, PositionReached
             } E_MvtSituation;
```

Au niveau de GesMot.c, suppression de la fonction pour appel cyclique et retouche de la fonction **MvtInit** ainsi que de la fonction :

```
void MvtInit(S_DescrMvt *pDescr, int8_t NoMvt)
{
   pDescr->NoMvt = NoMvt;
   pDescr->StartToDo = false;
   pDescr->MvtSituation = PositionReached;
   pDescr->CurPosition = 0;
   pDescr->GoalPosition = 0;
}
```

```
bool IsInPosition( S_DescrMvt *pDescr)
{
    bool stat = false;
    if ( pDescr->MvtSituation == PositionReached)
          stat = true;
    return stat;
}
```

10.3.6. MODIFICATION DE SYSTEM_INTERRUPT

L'interruption pour le cycle de l'application, ainsi que pour l'appel de **ExecMove** a été supprimée, ce qui permet de n'avoir qu'un timer sans interruption au niveau de la configuration.

System_interrupt.c ne contient plus que les réponses aux interruptions externes qui ont été modifiées par freeRTOS.

```
void IntHandlerExternalInterruptInstance0(void)
{
    PLIB_INT_SourceFlagClear(INT_ID_0,
                             INT_SOURCE_EXTERNAL_1);
    APP_UpdatePos1();
}

void IntHandlerExternalInterruptInstance1(void)
{
    PLIB_INT_SourceFlagClear(INT_ID_0,
                             INT_SOURCE_EXTERNAL_2);
    APP_UpdatePos2();

}

void IntHandlerExternalInterruptInstance2(void)
{
    PLIB_INT_SourceFlagClear(INT_ID_0,
                             INT_SOURCE_EXTERNAL_3);
    APP_UpdatePos3();
}
```

10.3.7. EVOLUTION DE APP.C

Voici l'évolution de app.c, dans lequel on utilise principalement le case APP_STATE_SERVICE_TASKS.

☝ Il faut initialiser le lcd dans le APP_Initialize qui s'exécute avant que le FreeRTOS soit activé.

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

    lcd_init();
    lcd_bl_on();

    printf_lcd("Chap10 avec RTOS  ");
    lcd_gotoxy(1,2);
    printf_lcd("C. Huber 28.06.2016");
}

void APP_Tasks ( void )
{
    /* Check the application's current state. */
    switch ( appData.state )
    {
        /* Application's initial state. */
        case APP_STATE_INIT:
        {

            // L'initialisation des descripteurs de
            // est faite dans chaque ammmx
            appData.SituationMouvement = 0;

            // Start Timer2 (Base temps OC)
            DRV_TMR0_Start();

            // pas de WAIT
            appData.state = APP_STATE_SERVICE_TASKS;
            break;
        }

        case APP_STATE_WAIT :
          // nothing to do
        break;
```

```
case APP_STATE_SERVICE_TASKS:
   // Inversion LED_2
   BSP_LEDToggle(BSP_LED_2);

   // Machine d'état traitement des 3 mouvements
   switch (appData.SituationMouvement) {
       case 0 :
          // lance 1er deplacement d'ensemble
          // Execution de trois mouvements
          RelMove(&appm1Data.DescrMot1, 300);
          RelMove(&appm2Data.DescrMot2, 500);
          RelMove(&appm3Data.DescrMot3, 700);
          appData.SituationMouvement = 1;
       break;

       case 1 :
           // Attente fin des 3 mouvement
           if (IsInPosition(&appm1Data.DescrMot1)
             && IsInPosition(&appm2Data.DescrMot2)
             && IsInPosition(&appm3Data.DescrMot3)
                                              ) {
               appData.SituationMouvement = 2;
               // Affiche la position
               // des 3 moteurs
               lcd_gotoxy(1,3);
               printf_lcd("M1:%06d M2:%06d ",
               GetPosition(&appm1Data.DescrMot1),
               GetPosition(&appm2Data.DescrMot2));
               lcd_gotoxy(1,4);
               printf_lcd("M3:%06d  ",
               GetPosition(&appm3Data.DescrMot3));
           }
       break;

       case 2 :
          // lance 2e deplacement d'ensemble
          // Execution de trois mouvements
          LED4_W = 1;
          RelMove(&appm1Data.DescrMot1, 500);
          RelMove(&appm2Data.DescrMot2, 750);
          RelMove(&appm3Data.DescrMot3, 1000);
          appData.SituationMouvement = 3;
       break;

       case 3 :
           // Attente fin des 3 mouvement
           if (IsInPosition(&appm1Data.DescrMot1)
             && IsInPosition(&appm2Data.DescrMot2)
             && IsInPosition(&appm3Data.DescrMot3)
                                            )   {
               appData.SituationMouvement = 0;
```

```
                            LED4_W = 0;
                            // Affiche la position
                            // des 3 moteurs
                            lcd_gotoxy(1,3);
                            printf_lcd("M1:%06d M2:%06d ",
                            GetPosition(&appm1Data.DescrMot1),
                            GetPosition(&appm2Data.DescrMot2));
                            lcd_gotoxy(1,4);
                            printf_lcd("M3:%06d  ",
                            GetPosition(&appm3Data.DescrMot3));

                           if (GetPosition(&appm3Data.DescrMot3)
                                                    > 150000 ) {
                              MvtInit(&appm1Data.DescrMot1, 1);
                              MvtInit(&appm2Data.DescrMot2, 2);
                              MvtInit(&appm3Data.DescrMot3, 3);
                            }
                        }
                    break;
                }

                // reste en exec
                // appData.state = APP_STATE_WAIT;
             break;

            /* The default state should never be executed. */
            default:
            {
                break;
            }
        }
    }

    void APP_UpdateState ( APP_STATES NewState )
    {
        appData.state = NewState;
    }

    void APP_UpdatePos1(void)
    {
         IncPosition(&appm1Data.DescrMot1);
    }
    void APP_UpdatePos2(void)
    {
         IncPosition(&appm2Data.DescrMot2);
    }
    void APP_UpdatePos3(void)
    {
         IncPosition(&appm3Data.DescrMot3);
    }
```

10.3.8. CONTROLE DU FONCTIONNEMENT

Il faut câbler les sorties OC sur les Int Ext :

- OC2 → Int1 76 → 18
- OC3 → Int2 77 → 19
- OC4 → Int3 78 → 66

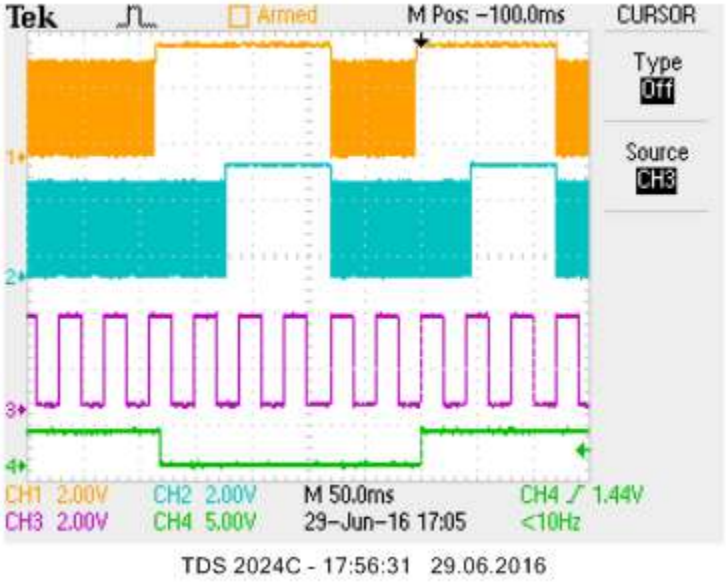
```
static void _APPM1_Tasks(void)
{
    while(1)
    {
        APPM1_Tasks();
        // vTaskDelay(1000 / portTICK_PERIOD_MS);
        BSP_LEDToggle(BSP_LED_3);
    }
}
```

Ch1 : OC2 (moteur 1)

Ch2 : OC3 (moteur 2)

Ch3 : LED_2
(toggle dans application)

Ch4 : LED_3
(toggle dans boucle
_APPM1_Task)



Si on met en relation le canal 1 avec le canal 4, on observe une longue attente avant l'exécution du mouvement. Le système fonctionne correctement.

10.4. COMMUNICATION INTER-TACHES

Pour une applications donnée, les différentes tâches auront inévitablement besoin de communications entre elles (passage de variables, réservation de ressource, etc.). Il y a donc un besoin pour plusieurs tâches d'accéder aux mêmes données. Dans un environnement multitâches, cela demande quelques précautions.

En effet, étant donné le fonctionnement multitâche de FreeRTOS (auquel peuvent se rajouter des interruptions "utilisateur"), une donnée ne doit être accédée que par une tâche à la fois, et de manière atomique (d'un seul bloc, sans être interrompu au milieu de l'opération).
FreeRTOS met à disposition le nécessaire à cela. On parle de fonctionnement "thread-safe".

10.4.1. QUEUE

Une queue permet de transférer des données entre tâches. Dans son utilisation standard, une queue est un FIFO thread-safe.

Une tâche peut :

- Placer un élément dans la queue (give). La tâche est suspendue si la queue est déjà pleine (et donc le contrôle va aux autres tâches).
- Y prendre un élément (take). La tâche est suspendue s'il n'y a aucun élément à prendre.

Exemple d'utilisation d'une queue :

```
#include "queue.h"

QueueHandle_t queueTx  = NULL; //déclaration

//Initialisation
queueTx  = xQueueCreate( 16, sizeof(char));

//Placement d'un élément dans la queue
xQueueSend( queueTx , &car, 0U );


// ... ailleurs dans le code :
//Réception d'un élément
//Reste bloqué tant que caractère pas reçu
if (xQueueReceive( queueTx, &car, portMAX_DELAY )) ...
```

Le troisième paramètre permet de spécifier le temps d'attente maximum en ticks.

10.4.2. SEMAPHORE

Un sémaphore peut être utilisé à des fins d'exclusion mutuelle ou de synchronisation de tâches. Une exclusion mutuelle est un mécanisme utilisé pour éviter que des ressources partagées (par exemple un port série, ou l'accès à un disque) ne soient utilisées en même temps par plusieurs tâches.

10.4.2.1. BINARY SEMAPHORE

Un sémaphore binaire peut être vu comme une queue de longueur unitaire où la valeur passée n'a pas d'intérêt. La queue est soit pleine, soit vide (binaire). La tâche qui veut prendre le sémaphore (take) sera donc suspendue jusqu'à ce que la tâche concernée donne le sémaphore (give).

- Exemple de cas d'exclusion mutuelle :
  La tâche qui a le sémaphore peut contrôler une ressource, puis céder son sémaphore et donc le contrôle de la ressource à une autre tâche lorsqu'elle a terminé.
- Exemple de cas de synchronisation de tâches :
  La tâche qui doit effectuer un traitement de donnée peut céder son sémaphore à une autre tâche dès qu'elle a terminé. L'autre tâche pourra alors lire le résultat du traitement et l'utiliser pour son traitement.

Exemple d'utilisation d'un sémaphore :

```
#include "semphr.h"

SemaphoreHandle_t semIntTimer = NULL; //déclaration

//Initialisation
semIntTimer =  xSemaphoreCreateBinary();

//pour débloquer la tâche qui attend le semaphore
xSemaphoreGiveFromISR(semIntTimer, NULL);


// ... ailleurs dans le code :
//attente semaphore (portMAX_DELAY = indéfiniment)
xSemaphoreTake(semIntTimer, portMAX_DELAY);
```

10.4.2.2. COUNTING SEMAPHORE

Un counting semaphore peut être vu comme une queue de longueur supérieure à 1. A nouveau, on ne se soucie pas de transmettre des valeurs, mais de combien de valeurs sont présentes dans le "FIFO" de transmission.

Au lieu d'avoir une simple valeur binaire, chaque appel à take incrémente un compteur, et chaque appel à give décrémente.

- Exemple de cas de comptage d'événements :
  Dans ce cas, une tâche va faire un "give" à chaque événement (incrémentation de la valeur du semaphore).
  Dans une autre tâche, pour chaque évènement, le même traitement doit être effectué. Un "take" est donc effectué (décrémentation de la valeur du sémaphore).
  La valeur de compteur est donc égale à la différence entre le nombre d'évènements survenus et le nombre d'évènements effectivement traités.
- Gestion de ressources
  Dans ce cas, la valeur du sémaphore indique le nombre de ressources disponibles.
  Lorsqu'une tâche veut obtenir une ressource, elle fait un "take" (décrémentation de la valeur), et lorsqu'elle a terminé, elle fait un "give" (incrémentation).
  Si la valeur vaut zéro, cela signifie qu'aucune ressource n'est disponible.

10.4.3. MUTEX

MutEx est une contraction de "MUTual EXclusion", ou exclusion mutuelle. Notons que le sémaphore peut également être utilisé à cette fin.

Un mutex est similaire à un sémaphore, excepté que le mutex inclut un mécanisme de changement de priorité de tâche que le sémaphore n'a pas.

10.4.3.1. MUTEX

Dans le cas d'un mutex simple, si une tâche fait un "take" et que le mutex n'est pas disponible et occupé par une tâche de plus faible priorité :

- Comme dans le cas du semaphore binaire, la tâche qui a fait le "take" sera suspendue.
- La différence par rapport au semaphore se trouve dans la gestion des priorités des tâches :
  La priorité de la tâche qui détient le mutex sera augmentée à la valeur de la priorité de la tâche qui demande le mutex. Ceci a pour effet de laisser la tâche de plus haute priorité (qui demande le mutex) le moins longtemps possible dans l'état suspendu.

10.4.3.2. RECURSIVE MUTEX

Dans le cas d'un mutex récursif, plusieurs "take" peuvent être effectués par une tâche. Le mutex ne deviendra disponible que lorsque la tâche qui a le mutex aura fait le même nombre d'appel "give".

10.5. CONCLUSION

Ce document devrait permettre, en s'inspirant des principes exposés et de l'exemple, de comprendre les principes d'un système d'exploitation temps-réel pour microcontrôleur et de mettre en œuvre FreeRTOS sur PIC32.

10.6. SOURCES

- Using The FreeRTOS Real Time Kernel - Microchip PIC32 Edition, Richard Barry, ISBN 978-1-4461-7108-0
- Wikipédia
- https://www.freertos.org
- https://www.freertos.org/Inter-Task-Communication.html
- http://www.chibios.org/dokuwiki/doku.php?id=chibios:articles:semaphores_mutexes

10.7. HISTORIQUE DES VERSIONS

10.7.1. VERSION 1.5 JUIN 2015

Création du document (version 1.5 = Harmony) par migration du chapitre 15 PIC18. L'exemple avec FreeRTOS n'est pas encore à disposition.

10.7.2. VERSION 1.7 JUIN 2016

Exemple avec machine d'état refait avec Harmony 1.06 et MPLABX 3.10. Essais avec FreeRTOS en chantier.

10.7.3. VERSION 1.71 JUIN 2016

Exemple avec machine d'état refait avec Harmony 1.06 et MPLABX 3.10. Mise au point de la partie FreeRTOS.

10.7.4. VERSION 1.8 MAI 2017

Relecture générale par SCA. Remise en forme. Ajout communications inter-processus.

10.7.5. VERSION 1.9 FEVRIER 2018

Ajout sources.

10.7.6. VERSION 1.91 JANVIER 2021

Compléments vTaskDelayUntil et sémaphores.