les écoles supérieures

ETML

MINF
Mise en œuvre des microcontrôleurs PIC32MX

Chapitre 8

Gestion du bus I2C, fonctions à machine d'état

🛠 T.P. PIC32MX

Christian HUBER (CHR)
Serge CASTOLDI (SCA)
Version v1.71 février 2022

CONTENU DU CHAPITRE 8

8. GESTION PAR MACHINE D'ÉTAT DU BUS I2C PIC32MX

Ce chapitre a pour objectif de découvrir le driver I2C fourni par Harmony.

L'objectif est de disposer de fonctions avec machine d'état, permettant un traitement cyclique et concurrent de composants I2C, ceci en utilisant au mieux les éléments fournis par Harmony.

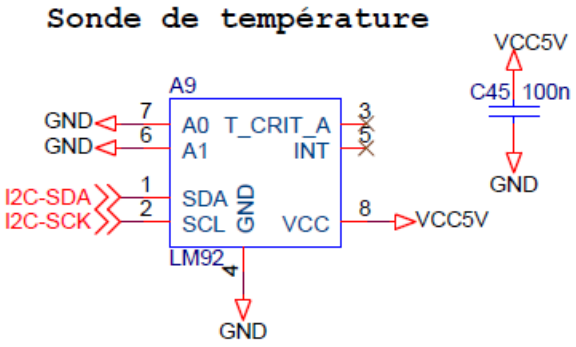
8.1. RÉALISATION DU BUS I2C AVEC LE PIC32MX95F512L

Le PIC32MX dispose de modules spécifiques pour la gestion du bus I2C. Dans le cadre du kit PIC32MX795F512L c'est le module I2C no 2 qui est utilisé.

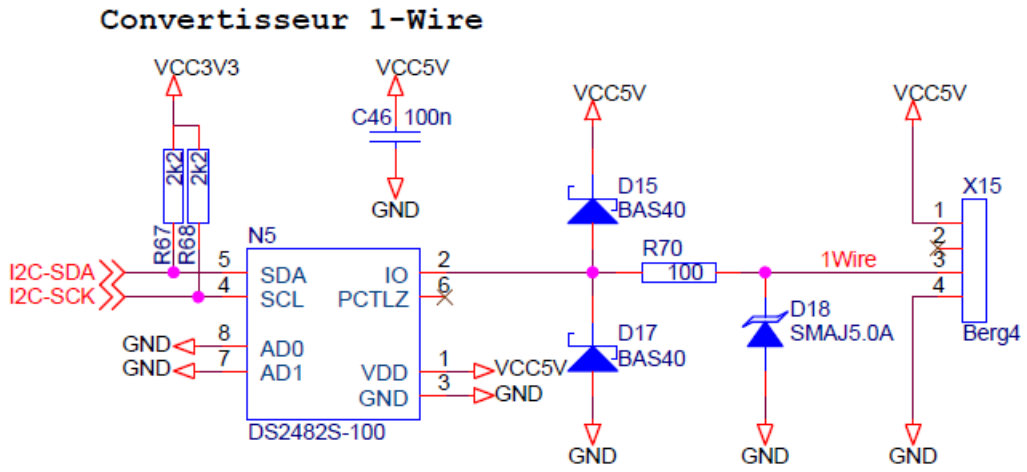
8.1.1. COMPOSANT I2C DU KIT PIC32MX795F512L

Il y a trois composants I2C sur le Kit PIC32MX795F512L.

8.1.1.1. SONDE DE TEMPÉRATURE LM92



8.1.1.2. CONVERTISSEUR 1-WIRE DS2482S-100

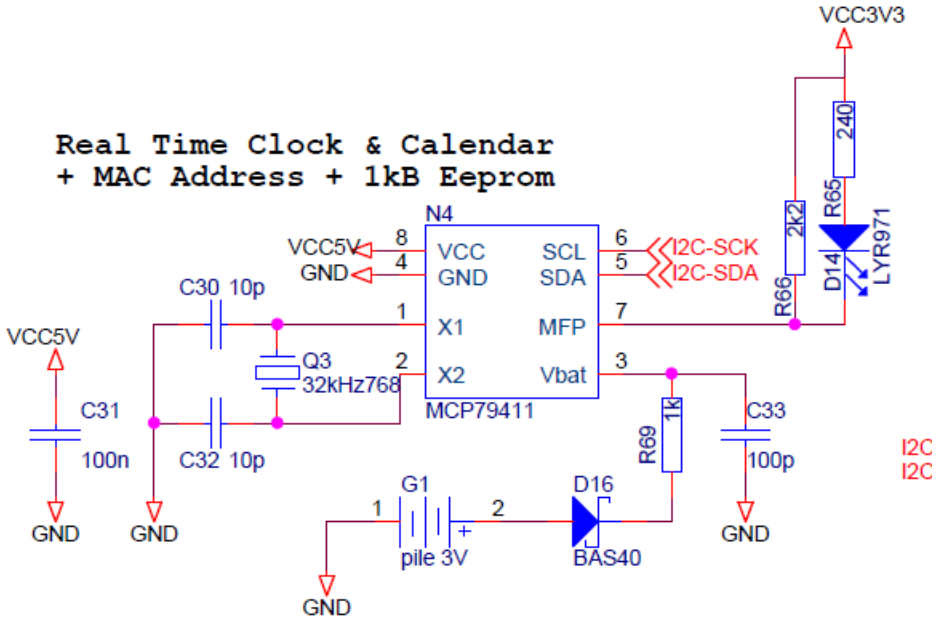


On remarque la paire de résistances de pull-up pour le bus I2C.

8.1.1.3. RTC, MAC ADRESSE ET SEEPROM

Le MCP79411 intègre :

- un Real Time Clock/Calendar,
- une adresse MAC préprogrammée d'usine,
- une EEPROM de 1 Kbits, soit 128 octets.

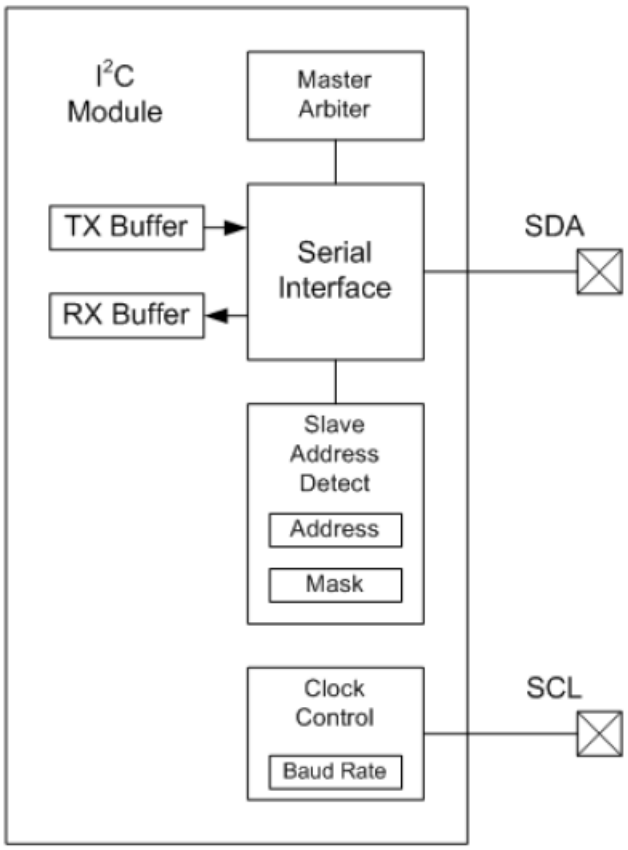


8.2. LES FONCTIONS DE LA PLIB_I2C

La PLIB_I2C fournit des fonctions de bas niveau qu'il faut combiner pour obtenir une action de base comme l'écriture ou la lecture d'un octet.

Cela nous permettra de découvrir les fonctions utilisées par le driver I2C fourni.

Dans l'aide de Harmony, section I2C Peripheral Library, on trouve le schéma de principe du module I2C.



8.2.1. VUE D'ENSEMBLE DES FONCTIONS DE LA PLIB_I2C

Voici une partie des fonctions de la plib_i2c. Les fonctions d'existence et de gestion du slave ne sont pas présentées.

Baud Rate Generator Control Functions

	Name	Description
	PLIB_I2C_BaudRateGet	Calculates the I2C module's current SCL clock frequency.
	PLIB_I2C_BaudRateSet	Sets the desired baud rate.

General Initialization Functions

	Name	Description
	PLIB_I2C_Disable	Disables the specified I2C module.
	PLIB_I2C_Enable	Enables the specified I2C module.
	PLIB_I2C_GeneralCallDisable	Disables the I2C module from recognizing the general call address.
	PLIB_I2C_GeneralCallEnable	Enables the I2C module to recognize the general call address.
	PLIB_I2C_HighFrequencyDisable	Disables the I2C module from using high frequency (400 kHz or 1 MHz) signaling.
	PLIB_I2C_HighFrequencyEnable	Enables the I2C module to use high frequency (400 kHz or 1 MHz) signaling.
	PLIB_I2C_IPMIDisable	Disables the I2C module's support for the IPMI specification
	PLIB_I2C_IPMIEnable	Enables the I2C module to support the Intelligent Platform Management Interface (IPMI) specification (see Remarks).
	PLIB_I2C_ReservedAddressProtectDisable	Disables the I2C module from protecting reserved addresses, allowing it to respond to them.
	PLIB_I2C_ReservedAddressProtectEnable	Enables the I2C module to protect (not respond to) reserved addresses.
	PLIB_I2C_SlaveClockStretchingDisable	Disables the I2C module from stretching the slave clock.
	PLIB_I2C_SlaveClockStretchingEnable	Enables the I2C module to stretch the slave clock.
	PLIB_I2C_SMBDisable	Disable the I2C module support for SMBus electrical signaling levels.
	PLIB_I2C_SMBEnable	Enables the I2C module to support System Management Bus (SMBus) electrical signaling levels.
	PLIB_I2C_StopInIdleDisable	Disables the Stop-in-Idle feature.
	PLIB_I2C_StopInIdleEnable	Enables the I2C module to stop when the processor enters Idle mode

General Status Functions

	Name	Description
	PLIB_I2C_ArbitrationLossClear	Clears the arbitration loss status flag
	PLIB_I2C_ArbitrationLossHasOccurred	Identifies if bus arbitration has been lost.
	PLIB_I2C_BusIsIdle	Determines whether the I2C bus is idle or busy.
	PLIB_I2C_StartClear	Clears the start status flag
	PLIB_I2C_StartWasDetected	Identifies when a Start condition has been detected.
	PLIB_I2C_StopClear	Clears the stop status flag
	PLIB_I2C_StopWasDetected	Identifies when a Stop condition has been detected

Master Control Functions

	Name	Description
	PLIB_I2C_MasterReceiverClock1Byte	Drives the bus clock to receive 1 byte of data from a slave device.
	PLIB_I2C_MasterStart	Sends an I2C Start condition on the I2C bus in Master mode.
	PLIB_I2C_MasterStartRepeat	Sends a repeated Start condition during an ongoing transfer in Master mode.
	PLIB_I2C_MasterStop	Sends an I2C Stop condition to terminate a transfer in Master mode.

Receiver Control Functions

	Name	Description
	PLIB_I2C_ReceivedByteAcknowledge	Allows a receiver to acknowledge a that a byte of data has been received.
	PLIB_I2C_ReceivedByteGet	Gets a byte of data received from the I2C bus interface.
	PLIB_I2C_ReceivedByteIsAvailable	Detects whether the receiver has data available.
	PLIB_I2C_ReceiverByteAcknowledgeHasCompleted	Determines if the previous acknowledge has completed.
	PLIB_I2C_ReceiverOverflowClear	Clears the receiver overflow status flag.
	PLIB_I2C_ReceiverOverflowHasOccurred	Identifies if a receiver overflow error has occurred.
	PLIB_I2C_MasterReceiverReadyToAcknowledge	Checks whether the hardware is ready to acknowledge.

Transmitter Control Functions

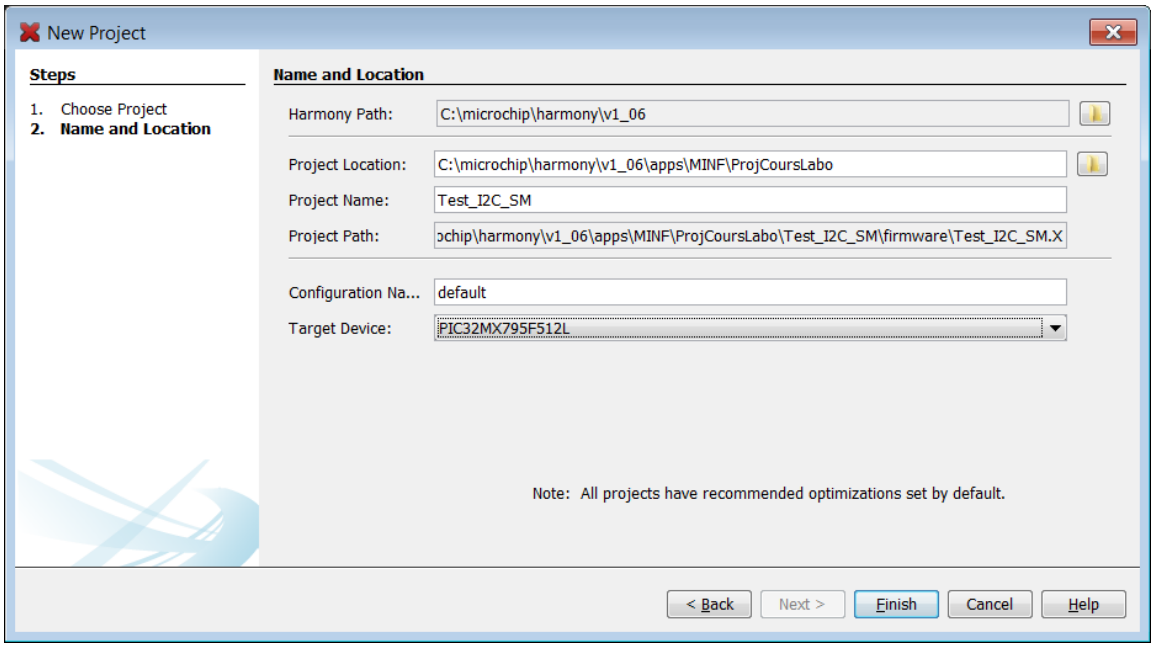
	Name	Description
	PLIB_I2C_TransmitterByteHasCompleted	Detects whether the module has finished transmitting the most recent byte.
	PLIB_I2C_TransmitterByteSend	Sends a byte of data on the I2C bus.
	PLIB_I2C_TransmitterByteWasAcknowledged	Determines whether the most recently sent byte was acknowledged.
	PLIB_I2C_TransmitterIsBusy	Identifies if the transmitter of the specified I2C module is currently busy (unable to accept more data).
	PLIB_I2C_TransmitterIsReady	Detects if the transmitter is ready to accept data to transmit.
	PLIB_I2C_TransmitterOverflowClear	Clears the transmitter overflow status flag.
	PLIB_I2C_TransmitterOverflowHasOccurred	Identifies if a transmitter overflow error has occurred.

8.3. PROJET AVEC DRIVER I2C

L'exemple suivant a été réalisé avec les logiciels suivants :

- Harmony v1.08
- MPLABX IDE v3.40
- XC32 v1.42

8.3.1. CRÉATION DU PROJET

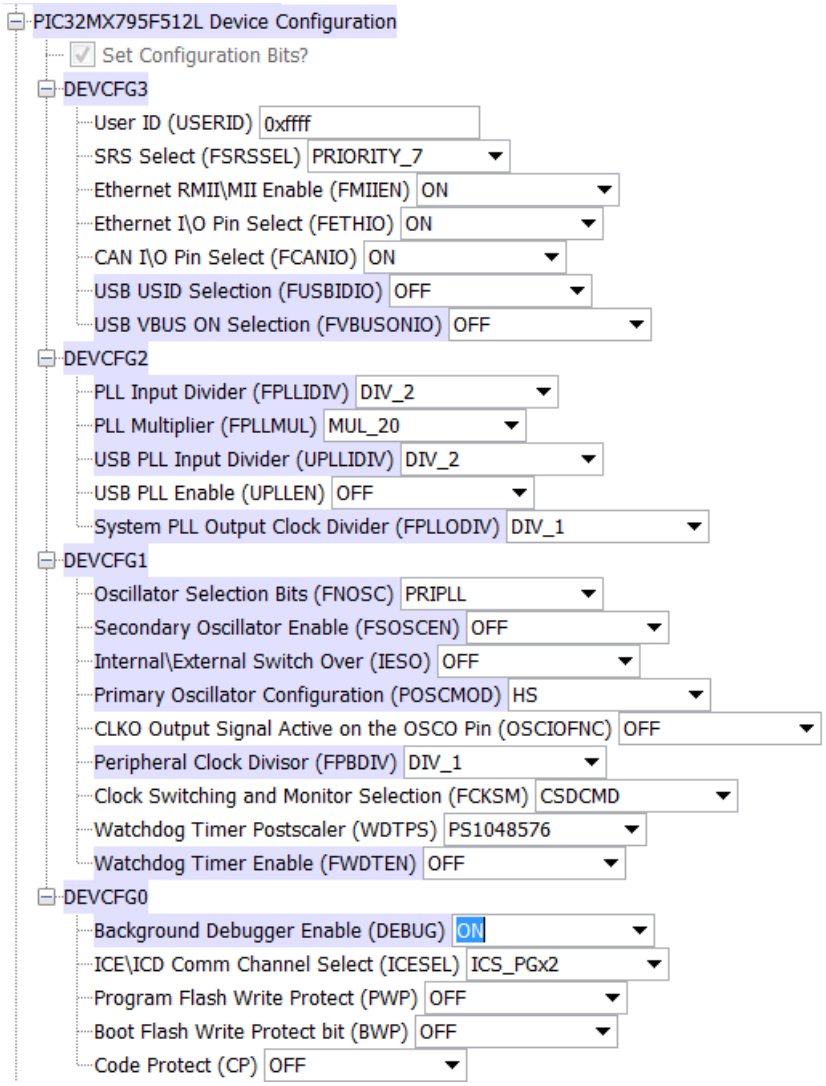


8.3.2. CHOIX DES ELÉMENTS

8.3.2.1. CHOIX BSP

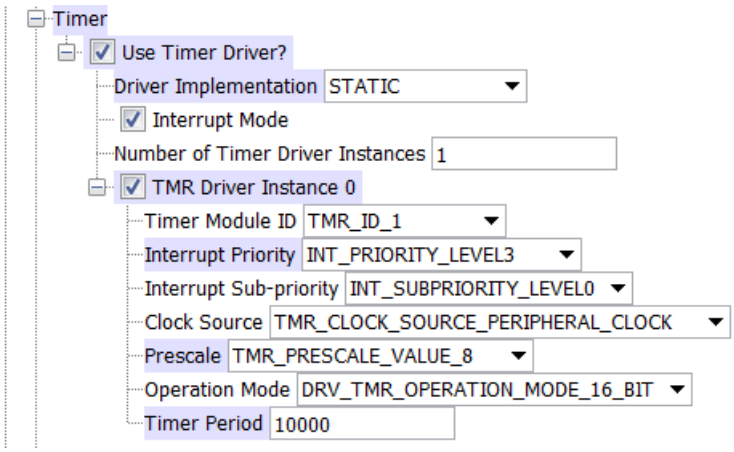
8.3.2.2. DEVICE CONFIGURATION

On retrouve les 4 sections déjà connues :



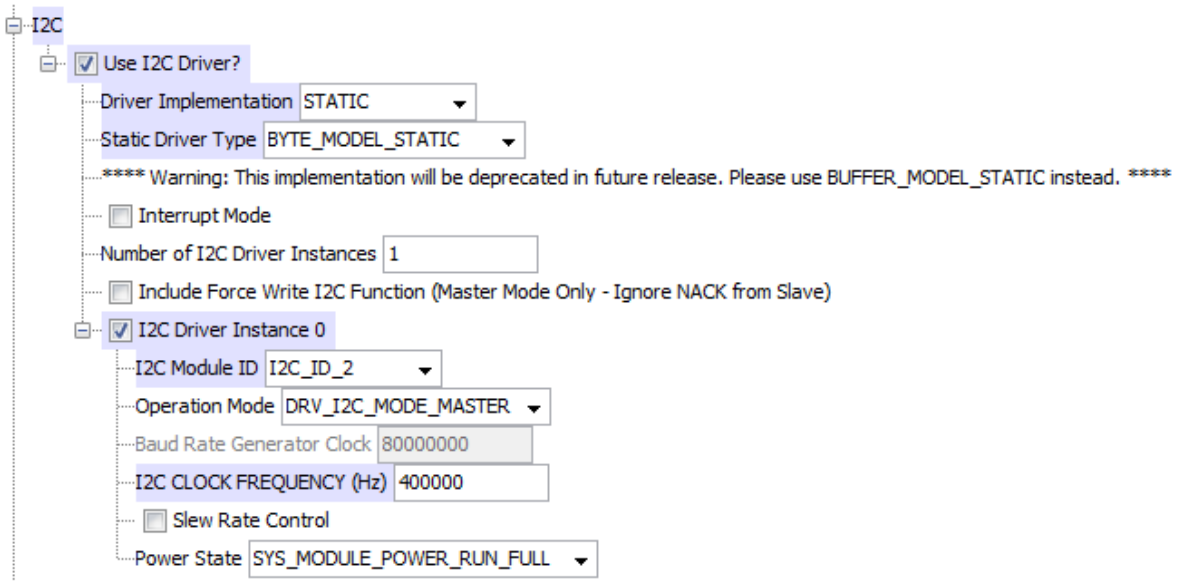
8.3.2.3. TIMER DRIVER

Configuration pour une période de 1ms et interruption priorité 3.



8.3.2.4. I2C DRIVER

Configuration en master, utilisation du module 2. Clock à 400 kHz.

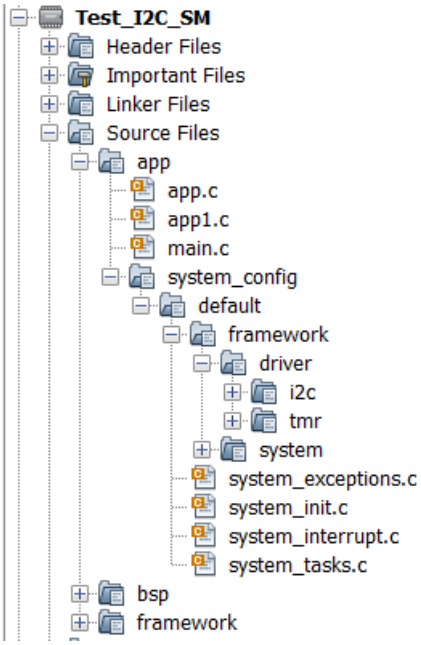


Ne pas cocher "slew rate control" afin d'éviter des problèmes d'incompatibilité avec le capteur de température LM92.

8.3.3. OBSERVATION DES RÉSULTATS

8.3.3.1. ORGANISATION

Après la génération du code, on constate au niveau du system_config, sous framework une section driver dans laquelle on trouve i2c et tmr.



8.4. RÉALISATION DE FONCTIONS I2C À MACHINE D'ÉTAT

Pour permettre une utilisation concurrente des fonctions I2C, par exemple pour gérer plusieurs composants et ceci au sein d'un cycle rapide, il est nécessaire de réaliser des fonctions comportant des machines d'état. Ce qui implique que la fonction est appelée cycliquement et que son traitement interne évolue.

Les fonctions à machine d'état sont des fonctions comportant un switch avec une variable d'état. Ces fonctions doivent être appelées cycliquement.

Une attente sera réalisée selon le principe suivant :

```
switch (MonEtat) {
    case 0 :
        // Attente Machin Ready
        if  (IsMachinReady() ) {
            // Passe à l'étape suivante
            MonEtat = 1
    break;
    case 1 :
    ...
```

Nous allons suivre par étape la réalisation d'un tel système pour obtenir la température du LM92.

8.4.1. ACTION AU NIVEAU DE L'APPLICATION

Au niveau de l'application, il faut appeler la fonction d'initialisation dans le case APP_STATE_INIT. Dans le case APP_STATE_SERVICE_TASKS, on utilise la fonction d'exploitation des résultats et on effectue l'affichage.

Il faut inclure dans **app.h** :

```
#include "Mc32gestI2cLM92_SM.h"
```

8.4.1.1. AJOUT DONNÉE APPLICATION POUR LM92 DANS STRUCT APP_DATA

L'ajout de champ dans la struct APP_DATA permet de disposer des variables chaque fois que app.h est inclus.

```
typedef struct
{
    /* The application's current state */
    APP_STATES state;
    /*  TODO:  Define  any  additional  data  used  by  the
application. */
    int16_t RawTemp;    // valeur brute registre température
    int32_t TempMilli;  // température en millième de degré
    float Lm92Temp;     // température en degré
} APP_DATA;
```

8.4.1.2. TRAITEMENT DANS CASE APP_STATE_INIT

Appel d'une fonction qui initialise le système à machine d'état ainsi que le bus I2C.

```
Lm92SmInit(true);  // en fast
```

8.4.1.3. TRAITEMENT DANS CASE APP_STATE_ SERVICE_TASKS

Dans la boucle du programme principal comportant un délai d'attente de 1000 ms, on teste si la température est disponible.

```
case APP_STATE_SERVICE_TASKS:
    // Activé toute les 1000 ms
    BSP_LEDToggle(BSP_LED_2);
    // Test si LM92 ready
    if (I2C_LM92_SM_IsReady(&DescrLm92)) {
        lcd_gotoxy(1,2);
        appData.RawTemp =
                    I2C_LM92_SM_GetRawTemp(&DescrLm92);
        printf_lcd( "LM92 raw: %04X   ", appData.RawTemp );
        lcd_gotoxy(1,3);
        appData.TempMilli =
                    I2C_LM92_SM_GetTempMilli(&DescrLm92);
        printf_lcd("LM92 Tmilli: %06d", appData.TempMilli);
        lcd_gotoxy(1,4);
        appData.Lm92Temp = I2C_LM92_SM_GetTemp(&DescrLm92);
        printf_lcd( "LM92 temp: %6.2f  ",
                                        appData.Lm92Temp);
      // Quittance la lecture
        I2C_LM92_SM_Restart(&DescrLm92);
    }
    appData.state = APP_STATE_WAIT;
break;
```

Les informations sont placées dans le descripteur, ce qui permet de les exploiter lorsqu'on a le Ready. Le descripteur est fourni dans Mc32gestI2cLM92_SM.h.

Pour relancer une conversion, on utilise la fonction **I2C_LM92_SM_Restart** qui réarme le système.

8.4.2. ACTION AU NIVEAU DE L'INTERRUPTION DU TIMER1

Dans la réponse à l'interruption du Timer1, prévue pour un cycle de 1 ms, on effectue l'appel de la fonction d'exécution **I2C_LM92_SM_Execute**. Le traitement évolue à l'intérieur de cette fonction. En plus on gère le déclenchement de l'application toutes les 1000 ms.

```
// Interruption timer1, Cycle 1 ms
void __ISR(_TIMER_1_VECTOR, ipl3AUTO)
                            _IntHandlerDrvTmrInstance0(void)
{
    static int count = 0;

    BSP_LEDOn(BSP_LED_0);

    // Appel cyclique traitement LM92
    I2C_LM92_SM_Execute(&DescrLm92);
    PLIB_INT_SourceFlagClear(INT_ID_0,INT_SOURCE_TIMER_1);
    count++;
    if (count >= 1000 ) {
        count = 0;
        APP_UpdateState(APP_STATE_SERVICE_TASKS);
    }
    BSP_LEDOff(BSP_LED_0);
}
```

8.4.3. INITIALISATION DANS LE FICHIER SYSTEM_INIT.C

Dans la fonction SYS_Initialize qui se trouve dans le fichier system_init.c, on met en commentaire l'appel de la fonction **DRV_I2C0_Initialize()** afin de l'appeler plus tard dans la fonction I2C_SM_init.

Voici une partie de la fonction SYS_Initialize :

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

    /* Initialize Drivers */
    /*Initialize TMR0 */
    DRV_TMR0_Initialize();

    // DRV_I2C0_Initialize();  // CHR utilisé plus tard
                               // dans I2C_SM_init

    /* Initialize System Services */
    SYS_INT_Initialize();

    /* Initialize Middleware */
    /* Enable Global Interrupts */
    SYS_INT_Enable();

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

8.4.4. CONTENU DU FICHIER MC32GESTI2CLM92_SM.H

Ce fichier contient les types énumérés définissant les valeurs des états et des séquences. Il contient aussi le typedef de la structure correspondant au descripteur, ainsi que le prototype des fonctions.

```
#include <stdint.h>
#include "Mc32_I2cUtil_SM.h"

// enumeration Etat principal
typedef enum { LM92_SM_Idle, LM92_SM_Busy, LM92_SM_ready}
E_LM92_state;

// enumeration Etapes de la séquence
typedef enum { LM92_I2CSEQ_Idle,
               LM92_I2CSEQ_Start,
               LM92_I2CSEQ_WriteAddrW,
               LM92_I2CSEQ_WritePtrTemp,
               LM92_I2CSEQ_ReStart,
               LM92_I2CSEQ_WriteAddrR,
               LM92_I2CSEQ_ReadMsb,
               LM92_I2CSEQ_ReadLsb,
               LM92_I2CSEQ_Stop,
                                } E_LM92_Sequence;
```

☝ Avec l'introduction du driver I2C, il n'est plus possible de modifier l'ID du module I2C d'où la mise en commentaire.

```
// Descripteur LM92 pour traitement par machine d'état
typedef struct {
    // I2C_MODULE i2cModuleId;        // Id du Module I2C
    E_LM92_state Lm92state;         // Etat principal
    E_LM92_Sequence Lm92Sequence;  // Etapes de la séquence
    S_Descr_I2C_SM I2cSmInfo;  // Descr. pour fonct. I2C_SM
    uint8 Lsb;
    uint8 Msb;
    sint16 RawTemp;     // valeur brute registre température
    sint32 TempMilli;   // température en millième de degré
    float Temperature;  // température en degré
} S_Descr_LM92_SM;

// Descripteur gestion LM92 par State Machine
extern S_Descr_LM92_SM DescrLm92;     // descripteur LM92

// prototypes des fonctions
void Lm92SmInit(bool Fast);  // pour appel externe

void I2C_LM92_SM_Init(S_Descr_LM92_SM *pDescr, bool Fast);
void I2C_LM92_SM_Execute(S_Descr_LM92_SM *pDescr);
void I2C_LM92_SM_Restart(S_Descr_LM92_SM *pDescr);
bool I2C_LM92_SM_IsReady(S_Descr_LM92_SM *pDescr);
```

```
float I2C_LM92_SM_GetTemp(S_Descr_LM92_SM *pDescr);
int32_t I2C_LM92_SM_GetTempMilli(S_Descr_LM92_SM *pDescr);
int16_t I2C_LM92_SM_GetRawTemp(S_Descr_LM92_SM *pDescr);
```

8.4.5. CONTENU DU FICHIER MC32GESTI2CLM92_SM.C

Ce fichier contient l'implémentation des fonctions.

```
#include "Mc32gestI2cLM92_SM.h"
#include "Mc32_I2cUtil_SM.h"
#include "system_config.h"    // pour bsp

// Compilation conditionelle (Mettre en commentaire
// pour ne pas utiliser les leds)
#define USE_LED_MEASURE true

// Définition pour LM92
#define lm92_rd    0x91        // lm92 address for read
#define lm92_wr    0x90        // lm92 address for write
#define lm92_temp_ptr  0x00    // adr. pointeur température

// Définitions du bus (pour mesures)
// #define I2C-SCK  SCL2/RA2      PORTAbits.RA2   pin 58
// #define I2C-SDA  SDa2/RA3      PORTAbits.RA3   pin 59

// Descripteur gestion LM92 par State Machine
S_Descr_LM92_SM DescrLm92;     // descripteur LM92
```

8.4.5.1. LA FONCTION LM92SMINIT

Cette fonction est une fonction interface qui permet un appel sans devoir fournir un descripteur.

```
void Lm92SmInit(bool Fast)
{
    I2C_LM92_SM_Init(&DescrLm92, true);
}
```

8.4.5.2. LA FONCTION I2C_LM92_SM_INIT

La fonction I2C_LM92_SM_Init effectue l'initialisation du mécanisme SM du LM92 et de la communication I2C.

```
void I2C_LM92_SM_Init(S_Descr_LM92_SM *pDescr, bool Fast)
{
   pDescr->Lm92state = LM92_SM_Idle;
   pDescr->Lm92Sequence =  LM92_I2CSEQ_Idle;
   pDescr->RawTemp = 0;
   pDescr->TempMilli = 0;
   pDescr->Temperature = 0.0;

   I2C_SM_init( Fast, &pDescr->I2cSmInfo );
}
```

8.4.5.3. LA FONCTION I2C_LM92_SM_RESTART

Cette fonction sort de l'état Ready et passe en Idle, ce qui fait recommencer la séquence au début dans la fonction **I2C_LM92_SM_Execute**.

```
void I2C_LM92_SM_Restart(S_Descr_LM92_SM *pDescr) {
   pDescr->Lm92state = LM92_SM_Idle;
   pDescr->Lm92Sequence =  LM92_I2CSEQ_Idle;
}
```

8.4.5.4. LA FONCTION I2C_LM92_SM_ISREADY

Cette fonction retourne true si la fonction d'exécution est dans l'état ready.

```
bool I2C_LM92_SM_IsReady(S_Descr_LM92_SM *pDescr) {
    bool answer = false;
    if (pDescr->Lm92state == LM92_SM_ready ) {
        answer = true;
    }
    return answer;
}
```

8.4.5.5. LA FONCTION I2C_LM92_SM_EXECUTE

Cette fonction doit être appelée cycliquement (interruption rapide ou boucle de traitement cyclique assez rapide).

Cette fonction effectue la lecture du registre de température du LM92 en la décomposant en étapes sans attente.

☝ Utilisation pour mesure de BSP_LED_6 et BSP_LED_5.

```
void I2C_LM92_SM_Execute(S_Descr_LM92_SM *pDescr)
{
    //Déclaration des variables
    int16_t RawTemp;
    bool AckBit;

    switch ( pDescr->Lm92state )  {
        case  LM92_SM_Idle :
            // Passe à Busy
            pDescr->Lm92state = LM92_SM_Busy;
            pDescr->Lm92Sequence = LM92_I2CSEQ_Start;
            #ifdef USE_LED_MEASURE
               BSP_LEDOn(BSP_LED_6);  // Début activité
            #endif
        break;

        case  LM92_SM_Busy :
            // Effectue la lecture par étapes
            switch (pDescr->Lm92Sequence)  {
                case LM92_I2CSEQ_Start :
                    // i2c_start();
                    I2C_SM_start(&pDescr->I2cSmInfo);
                    if (I2C_SM_isReady
```

```
                    (&pDescr->I2cSmInfo)){
            pDescr->Lm92Sequence =
                    LM92_I2CSEQ_WriteAddrW;
            I2C_SM_begin(&pDescr->I2cSmInfo);
            #ifdef USE_LED_MEASURE
                BSP_LEDToggle(BSP_LED_5);
            #endif
        }
    break;

    case LM92_I2CSEQ_WriteAddrW :
        //  i2c_write(lm92_wr);  addr. + Write
        I2C_SM_write(&pDescr->I2cSmInfo,
                    lm92_wr, &AckBit);
        if (I2C_SM_isReady
                        (&pDescr->I2cSmInfo)){
            pDescr->Lm92Sequence =
                     LM92_I2CSEQ_WritePtrTemp;
            I2C_SM_begin(&pDescr->I2cSmInfo);
            #ifdef USE_LED_MEASURE
                BSP_LEDToggle(BSP_LED_5);
            #endif
        }
    break;

    case LM92_I2CSEQ_WritePtrTemp :
        // i2c_write(lm92_temp_ptr);
        I2C_SM_write(&pDescr->I2cSmInfo,
                    lm92_temp_ptr, &AckBit);
        if (I2C_SM_isReady
                     (&pDescr->I2cSmInfo)){
            pDescr->Lm92Sequence =
                        LM92_I2CSEQ_ReStart;
            I2C_SM_begin(&pDescr->I2cSmInfo);
            #ifdef USE_LED_MEASURE
                BSP_LEDToggle(BSP_LED_5);
            #endif
        }
    break;

    case LM92_I2CSEQ_ReStart :
        // i2c_reStart();
        I2C_SM_reStart(&pDescr->I2cSmInfo);
        if (I2C_SM_isReady
                       (&pDescr->I2cSmInfo)){
            pDescr->Lm92Sequence =
                     LM92_I2CSEQ_WriteAddrR;
            I2C_SM_begin(&pDescr->I2cSmInfo);
            #ifdef USE_LED_MEASURE
                BSP_LEDToggle(BSP_LED_5);
            #endif
        }
```

```
            break;
            case LM92_I2CSEQ_WriteAddrR :
                // i2c_write(lm92_rd);     addr.+ Read
                I2C_SM_write(&pDescr->I2cSmInfo,
                             lm92_rd, &AckBit);
                if (I2C_SM_isReady
                                (&pDescr->I2cSmInfo)){
                    pDescr->Lm92Sequence =
                                LM92_I2CSEQ_ReadMsb;
                    I2C_SM_begin(&pDescr->I2cSmInfo);
                    #ifdef USE_LED_MEASURE
                        BSP_LEDToggle(BSP_LED_5);
                    #endif
                }
            break;

            case LM92_I2CSEQ_ReadMsb :
                //  msb = i2c_read(1);    // ack
                I2C_SM_read(&pDescr->I2cSmInfo, true,
                            &pDescr->Msb);
                if (I2C_SM_isReady
                                (&pDescr->I2cSmInfo)){
                    pDescr->Lm92Sequence =
                                LM92_I2CSEQ_ReadLsb;
                    I2C_SM_begin(&pDescr->I2cSmInfo);
                    #ifdef USE_LED_MEASURE
                        BSP_LEDToggle(BSP_LED_5);
                    #endif
                }
            break;

            case LM92_I2CSEQ_ReadLsb :
               // lsb = i2c_read(0);      // no ack
                I2C_SM_read(&pDescr->I2cSmInfo, false,
                            &pDescr->Lsb);
                if (I2C_SM_isReady
                                (&pDescr->I2cSmInfo)){
                    pDescr->Lm92Sequence =
                                LM92_I2CSEQ_Stop;
                    I2C_SM_begin(&pDescr->I2cSmInfo);
                    #ifdef USE_LED_MEASURE
                        BSP_LEDToggle(BSP_LED_5);
                    #endif
                }
            break;
```

```
                    case LM92_I2CSEQ_Stop :
                        // i2c_stop();
                        I2C_SM_stop(&pDescr->I2cSmInfo);
                        if (I2C_SM_isReady
                                        (&pDescr->I2cSmInfo)){
                            // Effectue les calculs
                            // des températures
                            RawTemp = pDescr->Msb;
                            RawTemp = RawTemp << 8;
                            RawTemp = RawTemp | pDescr->Lsb;
                            pDescr->RawTemp = RawTemp;
                            RawTemp = RawTemp / 8;
                            // bit poid faible = 0.0625 degré
                            pDescr->TempMilli = RawTemp * 62.5;
                            pDescr->Temperature =
                                            RawTemp * 0.0625;
                            pDescr->Lm92Sequence =
                                            LM92_I2CSEQ_Idle;
                            I2C_SM_begin(&pDescr->I2cSmInfo);
                            pDescr->Lm92state = LM92_SM_ready;
                            // marque fin séquence
                            #ifdef USE_LED_MEASURE
                                BSP_LEDToggle(BSP_LED_6);
                            #endif
                         }
                    break;

                    case LM92_I2CSEQ_Idle :
                        // ajout pour éviter Warning
                    break;
                }
            break;

            case  LM92_SM_ready :
                // Les résultats sont disponibles
                // Attente du Restart de l'utilisateur
                // après lecture des résultats
            break;
        }
    } // end I2C_LM92_SM_Execute
```

8.4.5.6. LA FONCTION I2C_LM92_SM_GETTEMP

Retourne la valeur du champ Temperature.

```
float I2C_LM92_SM_GetTemp(S_Descr_LM92_SM *pDescr)
{
    return pDescr->Temperature;
}
```

8.4.5.7. LA FONCTION I2C_LM92_SM_GETTEMPMILLI

Retourne la valeur du champ TempMilli (température au millième de degré).

```
int32_t I2C_LM92_SM_GetTempMilli(S_Descr_LM92_SM *pDescr) {
    return pDescr->TempMilli;
}
```

8.4.5.8. LA FONCTION I2C_LM92_SM_GETRAWTEMP

Retourne la valeur du champ RawTemp qui correspond au contenu du registre de température du LM92.

```
int16_t I2C_LM92_SM_GetRawTemp(S_Descr_LM92_SM *pDescr) {
    return pDescr->RawTemp;
}
```

8.4.6. CONTENU DU FICHIER MC32_I2CUTIL_SM.H

Ce fichier contient les définitions nécessaires à la gestion de la machine d'état interne aux fonctions, ainsi que le prototype des fonctions.

Les fonctions reprennent le principe de décomposition d'une transaction I2C en actions de base (start, restart, read, write et stop), tout en utilisant les fonctions du driver I2C.

☝ Cela a pour effet que l'ID du module I2C n'est plus en paramètres pour les fonctions.

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

// KIT 32MX795F512L Constants
#define KIT_I2C_BUS   I2C_ID_2   // n'est plus utilisée

typedef enum { I2C_XSM_Idle, I2C_XSM_Busy, I2C_XSM_Ready
             } E_I2C_XSM;

typedef struct {
    E_I2C_XSM I2cMainSM;
    int I2cSeqSM;
    int DebugCode;
} S_Descr_I2C_SM;

bool I2C_SM_isReady( S_Descr_I2C_SM *pDSM );
void I2C_SM_begin( S_Descr_I2C_SM *pDSM );

void I2C_SM_init( bool Fast, S_Descr_I2C_SM *pDSM );
void I2C_SM_start(S_Descr_I2C_SM *pDSM);
void I2C_SM_reStart(S_Descr_I2C_SM *pDSM);
void I2C_SM_write(S_Descr_I2C_SM *pDSM, uint8 data,
                  bool *pAck );
void I2C_SM_read(S_Descr_I2C_SM *pDSM,
                 bool ackTodo, uint8 *pData );
void I2C_SM_stop(S_Descr_I2C_SM *pDSM);
```

8.4.7. CONTENU DU FICHIER MC32_I2CUTIL_SM.C

Ce fichier contient l'implémentation des fonctions I2C, mais avec une machine d'état interne permettant de remplacer les boucles d'attentes par des passages successifs à travers un test.

Ce set de fonctions doit permettre la réalisation du traitement d'autres composants I2C en s'inspirant du principe adopté pour le LM92.

Les fonctions sont réalisées en utilisant les fonctions du driver I2C obtenues du MHC de Harmony 1_06

```
#include "Mc32_I2cUtil_SM.h"
#include "system_config/default/framework/driver/i2c/
                                                drv_i2c_static.h"

#define I2C_CLOCK_FAST 400000
#define I2C_CLOCK_SLOW 100000
```

8.4.7.1. LA FONCTION I2C_SM_INIT

Cette fonction initialise la machine d'état interne, et surtout configure et active le bus I2C. Cette fonction est prévue pour un appel unique. Elle appelle la fonction d'initialisation du driver I2C (version modifiée).

```
void I2C_SM_init( bool Fast, S_Descr_I2C_SM *pDSM )
{
    I2C_SM_begin(pDSM);

    // Appel la version modifiée de la fonction du driver
    DRV_I2C0_Initialize();
}
```

8.4.7.2. LA FONCTION I2C_SM_BEGIN

Cette fonction permet le réarmement de la machine d'état interne aux fonctions.

```
void I2C_SM_begin( S_Descr_I2C_SM *pDSM )
{
    pDSM->I2cMainSM = I2C_XSM_Idle;
    pDSM->I2cSeqSM  = 0;
    pDSM->DebugCode = 0;
}
```

8.4.7.3. LA FONCTION I2C_SM_ISREADY

Cette fonction permet de savoir si la fonction a achevé sa séquence interne, c'est-à-dire si le traitement est effectué.

```
bool I2C_SM_isReady( S_Descr_I2C_SM *pDSM )
{
    bool answer = false;

    if (pDSM->I2cMainSM == I2C_XSM_Ready ) {
        answer = true;
    }
    return answer;
}
```

8.4.7.4. LA FONCTION I2C_SM_START

Effectue le start I2C en séquence. Doit être appelée cycliquement.

```
void I2C_SM_start(S_Descr_I2C_SM *pDSM)
{
    int DebugCode = 0;

    switch (pDSM->I2cMainSM ) {

        case I2C_XSM_Idle:
            // démarre le traitement
            pDSM->I2cMainSM = I2C_XSM_Busy;
            pDSM->I2cSeqSM = 0;
        break;

        case I2C_XSM_Busy :
            switch ( pDSM->I2cSeqSM) {
                case 0 :
                    // Wait for the bus to be idle,
                    // then start the transfer
                    if ( DRV_I2C0_MasterBusIdle() ) {
                       // OK alors Start
                       pDSM->I2cSeqSM = 1;
                       if( DRV_I2C0_MasterStart() == false)
                       {
                           pDSM->DebugCode = 1;
                       }
                    }
                break;

                case 1 :
                     // Wait for the signal to complete
                     if (DRV_I2C0_WaitForStartComplete()) {
                        pDSM->I2cSeqSM = 0;
                        pDSM->I2cMainSM = I2C_XSM_Ready;
                    }
                break;
            } // end switch SeqSM
        break;

        case I2C_XSM_Ready :
              // Attente relancement
        break;
    } // end switch
} // end I2C_SM_start
```

8.4.7.5. LA FONCTION I2C_SM_RESTART

Effectue le restart I2C en séquence. Doit être appelée cycliquement.

```
void I2C_SM_reStart( S_Descr_I2C_SM *pDSM)
{
     switch (pDSM->I2cMainSM ) {

        case I2C_XSM_Idle:
            // démarre le traitement
            pDSM->I2cMainSM = I2C_XSM_Busy;
            pDSM->I2cSeqSM = 0;
        break;

        case I2C_XSM_Busy :
            switch ( pDSM->I2cSeqSM) {
                case 0 :
                    // Restart the transfer
                    if(DRV_I2C0_MasterRestart() == false)
                    {
                         pDSM->DebugCode = 2;
                    }
                    pDSM->I2cSeqSM = 1;
                break;

                case 1 :
                    // Wait for the signal to complete
                    if ( DRV_I2C0_WaitForStartComplete())
                    {
                        pDSM->I2cSeqSM = 0;
                        pDSM->I2cMainSM = I2C_XSM_Ready;
                    }
                break;
            } // end switch SeqSM
        break;

        case I2C_XSM_Ready :
              // Attente relancement
        break;
    } // end switch
} // end I2C_SM_reStart
```

8.4.7.6. LA FONCTION I2C_SM_WRITE

Effectue la transmission d'un octet sur le bus I2C en séquence. Doit être appelée cycliquement.

Cette fonction reçoit en paramètre par valeur l'octet à transmettre et par référence un booléen pour indiquer si l'octet transmis a été quittancé (acknowledge).

```
void I2C_SM_write(S_Descr_I2C_SM *pDSM,
                  uint8_t data, bool *pAck )
{
    switch (pDSM->I2cMainSM ) {

        case I2C_XSM_Idle:
            // demarre le traitement
            pDSM->I2cMainSM = I2C_XSM_Busy;
            pDSM->I2cSeqSM = 0;
        break;

        case I2C_XSM_Busy :
            switch ( pDSM->I2cSeqSM) {
                case 0 :
                    // Wait for the bus to be idle
                    // (nécessaire aprés un reStart)
                    if (DRV_I2C0_MasterBusIdle() ) {
                         pDSM->I2cSeqSM = 1;
                    }
                break;

                case 1 :
                    // Wait for the transmitter to be ready
                       // CHR ?! cette étape n'existe pas
                       // dans le driver,
                       // mais ne semble pas nécessaire

                    // Transmit the byte
                    if (DRV_I2C0_ByteWrite(data) == false)
                    {
                              pDSM->DebugCode = 3;
                    }
                    pDSM->I2cSeqSM = 2;
                break;

                case 2 :
                    // Wait for byte write completion
                    if
                     (DRV_I2C0_WaitForByteWriteToComplete() ) {
                          // on peut obtenir Ack
                          *pAck =
                            DRV_I2C0_WriteByteAcknowledged();
                          pDSM->I2cSeqSM = 0;
                          pDSM->I2cMainSM = I2C_XSM_Ready;
                    }
                break;
            } // end switch SeqSM
```

```
        break;
        case I2C_XSM_Ready :
              // Attente relancement
        break;
    } // end switch

} // end I2C_SM_write
```

8.4.7.7. LA FONCTION I2C_SM_READ

Effectue la réception d'un octet sur le bus I2C en séquence. Doit être appelée cycliquement.

Cette fonction a en paramètre par valeur, un booléen pour indiquer s'il faut effectuer ou non l'acquittement. Le retour de la valeur est réalisé par référence.

```
void I2C_SM_read(S_Descr_I2C_SM *pDSM,
                 bool ackTodo, uint8_t *pData )
{
    switch (pDSM->I2cMainSM ) {
        case I2C_XSM_Idle:
            // démarre le traitement
            pDSM->I2cMainSM = I2C_XSM_Busy;
            pDSM->I2cSeqSM = 0;
        break;
        case I2C_XSM_Busy :
            switch ( pDSM->I2cSeqSM) {
                case 0 :
                    // Check for receive overflow
                    // If OK Initiate clock to receive
                    if(DRV_I2C0_SetUpByteRead() == false)
                    {
                         pDSM->DebugCode = 4;  // debug
                    }
                    pDSM->I2cSeqSM = 1;
                break;
                case 1 :
                    // Wait until data availlable
                    if (DRV_I2C0_WaitForReadByteAvailable()
                                                        ) {
                        *pData = DRV_I2C0_ByteRead();
                        pDSM->I2cSeqSM = 2;
                    }
                break;
                case 2 :
                    // Wait until ready to ack
                    if (ackTodo) {
                        DRV_I2C0_MasterACKSend();
                    } else {
                        DRV_I2C0_MasterNACKSend();
                    }
```

```
                    // ??? attente dans boucle
                    DRV_I2C0_WaitForACKOrNACKComplete();
                    pDSM->I2cSeqSM = 0;
                    pDSM->I2cMainSM = I2C_XSM_Ready;
                break;
            } // end switch SeqSM
        break;

        case I2C_XSM_Ready :
              // Attente relancement
        break;
    } // end switch
} // end I2C_SM_read
```

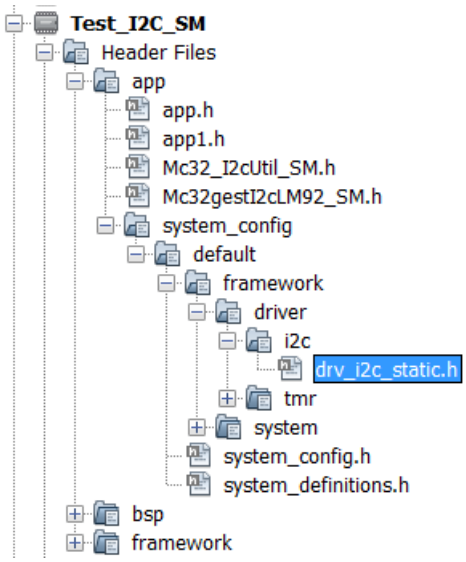
8.4.7.8. LA FONCTION I2C_SM_STOP

Effectue la transaction stop sur le bus I2C en séquence, pour terminer la transaction I2C master. Doit être appelée cycliquement.

```
void I2C_SM_stop(S_Descr_I2C_SM *pDSM)
{
    switch (pDSM->I2cMainSM ) {

        case I2C_XSM_Idle:
            // demarre le traitement
            pDSM->I2cMainSM = I2C_XSM_Busy;
            pDSM->I2cSeqSM = 0;
        break;

        case I2C_XSM_Busy :
            switch ( pDSM->I2cSeqSM) {
                case 0 :
                    // Disable receiver and stop I2C
                    if ( DRV_I2C0_MasterStop() ) {
                         pDSM->I2cSeqSM = 1;
                    }
                    // Si pas OK on répète le stop
                break;

                case 1 :
                    // Wait for the signal to complete
                    // boucle d'attente dans la fonction
                   DRV_I2C0_WaitForStopComplete();
                   pDSM->I2cSeqSM = 0;
                   pDSM->I2cMainSM = I2C_XSM_Ready;
                break;
            } // end switch SeqSM
        break;

        case I2C_XSM_Ready :
              // Attente relancement
        break;
    } // end switch
} // end I2C_SM_stop
```

8.4.8. CONTENU DU FICHIER DRV_I2C_STATIC.H

Voici le contenu du fichier drv_i2c_static.h (sans l'entête Microchip) que l'on trouve sous :

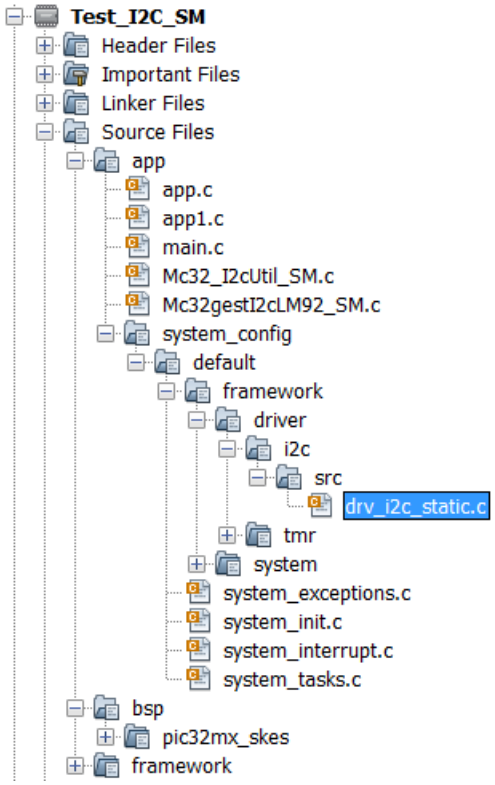



```
#ifndef _DRV_I2C_STATIC_H
#define _DRV_I2C_STATIC_H

#include <stdbool.h>
#include "system_config.h"
#include "peripheral/i2c/plib_i2c.h"
#include "system/clk/sys_clk.h"
#include "peripheral/ports/plib_ports.h"

// *****************************************************
// Section: Interface Headers for Instance 0 for the
                                          static driver
// *****************************************************
void DRV_I2C0_Initialize(void);
void DRV_I2C0_DeInitialize(void);

// *****************************************************
// Section: Instance 0 Byte transfer functions
                                        (Master/Slave)
// *****************************************************

bool DRV_I2C0_SetUpByteRead(void);
bool DRV_I2C0_WaitForReadByteAvailable(void);
uint8_t DRV_I2C0_ByteRead(void);
bool DRV_I2C0_ByteWrite(const uint8_t byte);
bool DRV_I2C0_WaitForByteWriteToComplete(void);
bool DRV_I2C0_WriteByteAcknowledged(void);
bool DRV_I2C0_ReceiverBufferIsEmpty(void);
```

```
// *****************************************************************
// Section: Instance 0 I2C Master functions
// *****************************************************************
void DRV_I2C0_BaudRateSet(I2C_BAUD_RATE baudRate);
bool DRV_I2C0_MasterBusIdle(void);
bool DRV_I2C0_MasterStart(void);
bool DRV_I2C0_MasterRestart(void);
bool DRV_I2C0_WaitForStartComplete(void);
bool DRV_I2C0_MasterStop(void);
bool DRV_I2C0_WaitForStopComplete(void);
void DRV_I2C0_MasterACKSend(void);
void DRV_I2C0_MasterNACKSend(void);
bool DRV_I2C0_WaitForACKOrNACKComplete(void);

#endif // #ifndef _DRV_I2C_STATIC_H
```

8.4.9. CONTENU DU FICHIER DRV_I2C_STATIC.C

Voici le contenu du fichier drv_i2c_static.c (sans l'entête Microchip) que l'on trouve sous :



☺ De par le fait que le driver est généré dans la structure de notre projet, il est possible d'apporter des modifications au driver sans toucher à des sections système communes.

```
#include "framework/driver/i2c/drv_i2c_static.h"
```

8.4.9.1. LA FONCTION DRV_I2C0_INITIALIZE CRÉÉE

La fonction DRV_I2C0_Initialize ci-dessous correspond à la version générée par Harmony :

```
void DRV_I2C0_Initialize(void)
{
    /* Initialize I2C0 */
    PLIB_I2C_BaudRateSet(I2C_ID_2,
    SYS_CLK_PeripheralFrequencyGet(CLK_BUS_PERIPHERAL_1),
    400000);
    PLIB_I2C_StopInIdleDisable(I2C_ID_2);

    /* Low frequency is enabled (**NOTE** PLIB function
    logic inverted) */
    PLIB_I2C_HighFrequencyEnable(I2C_ID_2);

    /* Enable I2C0 */
    PLIB_I2C_Enable(I2C_ID_2);
}
```

8.4.9.2. LA FONCTION DRV_I2C0_INITIALIZE MODIFIÉE

La fonction DRV_I2C0_Initialize modifiée ci-dessous reprend le principe déjà utilisé dans la fonction d'initialisation de l2C en version classique. Ce code est plus robuste et sera donc utilisé.

```
void DRV_I2C0_Initialize(void)
{
    /* Initialize I2C0 */
    PLIB_I2C_Disable(I2C_ID_2);   // Ajout CHR
    PLIB_I2C_HighFrequencyEnable(I2C_ID_2);
    // Toujours en Fast
    PLIB_I2C_BaudRateSet(I2C_ID_2,
              SYS_CLK_PeripheralFrequencyGet
                        (CLK_BUS_PERIPHERAL_1), 400000);
    /* Enable I2C0 */
    PLIB_I2C_Enable(I2C_ID_2);
}
```

8.4.9.3. LA FONCTION DRV_I2C0_DEINITIALIZE

Voici la fonction DRV_I2C0_DeInitialize :

```
void DRV_I2C0_DeInitialize(void)
{
    /* Disable I2C0 */
    PLIB_I2C_Disable(I2C_ID_2);
}
```

8.4.9.4. LA FONCTION DRV_I2C0_SETUPBYTEREAD

Voici la fonction DRV_I2C0_SetUpByteRead. Cette fonction teste la situation d'overflow et génère les coups d'horloge pour la lecture.

```
bool DRV_I2C0_SetUpByteRead(void)
{
    /* Check for receive overflow */
    if ( PLIB_I2C_ReceiverOverflowHasOccurred(I2C_ID_2))
    {
        PLIB_I2C_ReceiverOverflowClear(I2C_ID_2);
        return false;
    }
    /* Initiate clock to receive */
    PLIB_I2C_MasterReceiverClock1Byte(I2C_ID_2);
    return true;
}
```

8.4.9.5. LA FONCTION DRV_I2C0_WAITFORREADBYTEAVAILABLE

Voici la fonction DRV_I2C0_WaitForReadByteAvailable. Cette fonction teste la disponibilité d'un octet.

```
bool DRV_I2C0_WaitForReadByteAvailable(void)
{
    /* Wait for Receive Buffer Full */
    if(PLIB_I2C_ReceivedByteIsAvailable(I2C_ID_2))
       return true;

    return false;
}
```

8.4.9.6. LA FONCTION DRV_I2C0_BYTEREAD

Voici la fonction DRV_I2C0_ByteRead. Cette fonction effectue la lecture d'un octet dans le tampon de réception.

```
uint8_t DRV_I2C0_ByteRead(void)
{
    /* Return received value */
    return (PLIB_I2C_ReceivedByteGet(I2C_ID_2));
}
```

8.4.9.7. LA FONCTION DRV_I2C0_BYTEWRITE

Voici la fonction DRV_I2C0_ByteWrite. Cette fonction effectue des tests et envoie un octet si tout est OK.

```
bool DRV_I2C0_ByteWrite(const uint8_t byte)
{
    /* if no IWCOL errors exist, then transmit byte */
    if ( (!(PLIB_I2C_TransmitterIsBusy(I2C_ID_2))) &&
         (PLIB_I2C_TransmitterByteHasCompleted(I2C_ID_2)) )
    {
        PLIB_I2C_TransmitterByteSend(I2C_ID_2, byte);
    }

    /* check if writing to I2CxTRN caused a transmitter
        overflow */
    if (PLIB_I2C_TransmitterOverflowHasOccurred(I2C_ID_2))
        return false;

    return true;
}
```

8.4.9.8. LA FONCTION DRV_I2C0_WAITFORBYTEWRITETOCOMPLETE

Voici la fonction DRV_I2C0_WaitForByteWriteToComplete. Elle indique si une écriture est terminée.

```
bool DRV_I2C0_WaitForByteWriteToComplete(void)
{
    /* if TBF == 0 and TRSTAT == 0 then write complete */

    if ( (!(PLIB_I2C_TransmitterIsBusy(I2C_ID_2))) &&
         (PLIB_I2C_TransmitterByteHasCompleted(I2C_ID_2)) )
         return true;

    return false;
}
```

8.4.9.9. LA FONCTION DRV_I2C0_WRITEBYTEACKNOWLEDGED

Voici la fonction DRV_I2C0_WriteByteAcknowledged. Cette fonction teste si l'acknowledge a été effectué.

```
bool DRV_I2C0_WriteByteAcknowledged(void)
{
    /* Check to see if transmit ACKed = true or NACKed =
    false */
    if (PLIB_I2C_TransmitterByteWasAcknowledged(I2C_ID_2))
       return true;

    return false;
}
```

8.4.9.10. LA FONCTION DRV_I2C0_BAUDRATESET

Voici la fonction DRV_I2C0_BaudRateSet :

```
void DRV_I2C0_BaudRateSet(I2C_BAUD_RATE baudRate)
{
    /* Disable I2C0 */
    PLIB_I2C_Disable(I2C_ID_2);

    /* Change baud rate */
    PLIB_I2C_BaudRateSet(I2C_ID_2,
    SYS_CLK_PeripheralFrequencyGet(CLK_BUS_PERIPHERAL_1),
    baudRate);

    /* Low frequency is enabled (**NOTE** PLIB function
    inverted) */
    PLIB_I2C_HighFrequencyEnable(I2C_ID_2);

    /* Enable I2C0 */
    PLIB_I2C_Enable(I2C_ID_2);
}
```

8.4.9.11. LA FONCTION DRV_I2C0_MASTERBUSIDLE

Voici la fonction DRV_I2C0_MasterBusIdle qui permet de tester si le bus I2C est au repos :

```
bool DRV_I2C0_MasterBusIdle(void)
{
    if (PLIB_I2C_BusIsIdle(I2C_ID_2))
       return true;
    else
       return false;
}
```

8.4.9.12. LA FONCTION DRV_I2C0_MASTERSTART

Voici la fonction DRV_I2C0_MasterStart qui effectue une série de tests avant d'exécuter le Start :

```
bool DRV_I2C0_MasterStart(void)
{
    /* if bus is not idle return with false */
    if (!(PLIB_I2C_BusIsIdle(I2C_ID_2)))
       return false;

    /* return false is Bus Collision exisits */
    if (PLIB_I2C_ArbitrationLossHasOccurred(I2C_ID_2))
    {
        return false;
    }

    /* Issue start */
    PLIB_I2C_MasterStart(I2C_ID_2);

    return true;
}
```

8.4.9.13. LA FONCTION DRV_I2C0_WAITFORSTARTCOMPLETE

Voici la fonction DRV_I2C0_WaitForStartComplete qui indique si le start est terminé :

```
bool DRV_I2C0_WaitForStartComplete(void)
{
    /* Wait for start/restart sequence to finish  (hardware
    clear) */

    if ( (PLIB_I2C_BusIsIdle(I2C_ID_2)) &&
         (PLIB_I2C_StartWasDetected(I2C_ID_2)) )
        return true;

    return false;
}
```

8.4.9.14. LA FONCTION DRV_I2C0_MASTERRESTART

Voici la fonction DRV_I2C0_MasterRestart qui effectue une série de tests avant d'exécuter le MasterStartRepeat.

```
bool DRV_I2C0_MasterRestart (void)
{
    /* if bus is not idle return with false */
    if (!(PLIB_I2C_BusIsIdle(I2C_ID_2)))
       return false;

    /* return false is Bus Collision exisits */
    if (PLIB_I2C_ArbitrationLossHasOccurred(I2C_ID_2))
    {
        return false;
```

```
    }

    /* Issue restart */
    PLIB_I2C_MasterStartRepeat(I2C_ID_2);

    return true;
}
```

8.4.9.15. LA FONCTION DRV_I2C0_MASTERSTOP

Voici la fonction DRV_I2C0_MasterStop, qui effectue une série de tests avant d'exécuter le MasterStop :

```
bool DRV_I2C0_MasterStop(void)
{
    /* if bus is not idle return with false */
    if (!(PLIB_I2C_BusIsIdle(I2C_ID_2)))
       return false;

    /* Issue stop */
    PLIB_I2C_MasterStop(I2C_ID_2);

    return true;
}
```

8.4.9.16. LA FONCTION DRV_I2C0_WAITFORSTOPCOMPLETE

Voici la fonction DRV_I2C0_WaitForStopComplete qui indique si le stop est terminé :

```
bool DRV_I2C0_WaitForStopComplete(void)
{
    if ( (PLIB_I2C_BusIsIdle(I2C_ID_2)) &&
        (PLIB_I2C_StopWasDetected(I2C_ID_2)) )
        return true;

        return false;
}
```

8.4.9.17. LA FONCTION DRV_I2C0_MASTERACKSEND

Voici la fonction DRV_I2C0_MasterACKSend qui effectue l'acknowledge si le master est prêt.

```
void DRV_I2C0_MasterACKSend(void)
{
    /* Check if receive is ready to ack */
    if ( PLIB_I2C_MasterReceiverReadyToAcknowledge
                                          (I2C_ID_2) )
    {
        PLIB_I2C_ReceivedByteAcknowledge (I2C_ID_2, true);
    }
}
```

8.4.9.18. LA FONCTION DRV_I2C0_MASTERNACKSEND

Voici la fonction DRV_I2C0_MasterNACKSend qui effectue le NACK si le master est prêt.

```
void DRV_I2C0_MasterNACKSend(void)
{
    /* Check if receive is ready to nack */
    if ( PLIB_I2C_MasterReceiverReadyToAcknowledge
                                                (I2C_ID_2) )
    {
        PLIB_I2C_ReceivedByteAcknowledge (I2C_ID_2, false);
    }
}
```

8.4.9.19. LA FONCTION DRV_I2C0_WAITFORACKORNACKCOMPLETE

Voici la fonction DRV_I2C0_WaitForACKOrNACKComplete qui indique si l'ACK, respectivement le NACK, et terminé :

```
bool DRV_I2C0_WaitForACKOrNACKComplete(void)
{
    /* Check for ACK/NACK to complete */

    if(PLIB_I2C_ReceiverByteAcknowledgeHasCompleted(I2C_ID_
    2))
        return true;

    return false;
}
```

8.4.10. CONTRÔLE DE FONCTIONNEMENT DES FONCTIONS SM

8.4.10.1. VUE D'ENSEMBLE CYCLE 1 MS

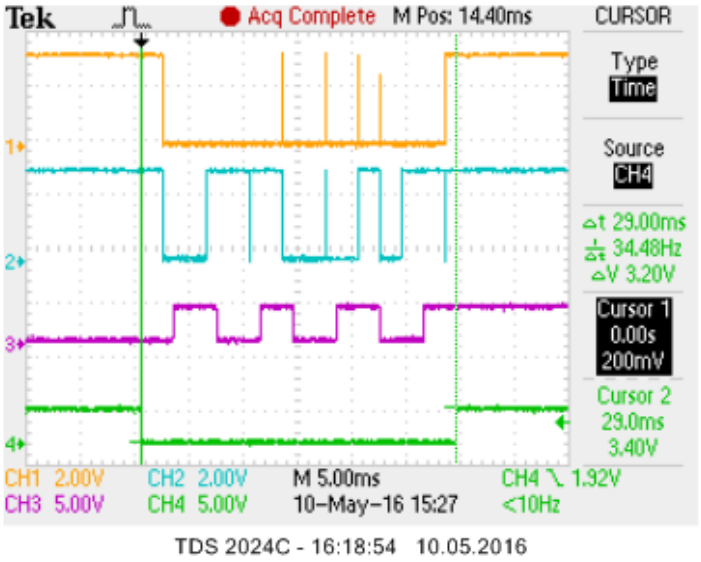
Voici une mesure de l'ensemble de la transaction avec un cycle de 1 ms :

Canal 1 : SCK
I2C-SCK / SCL2/RA2
PORTAbits.RA2 / pin 58

Canal 2 : SDA
I2C-SDA / SDa2/RA3
PORTAbits.RA3 / pin 59

Canal 3 : LED_5
toggle à chaque fin d'étape

Canal 4 : LED_6
marqueur enveloppe de la partie active



On obtient un temps d'exécution de 29 ms, ce qui est important et rend difficile l'observation des signaux I2C.

8.4.10.2. VUE D'ENSEMBLE CYCLE 100 US

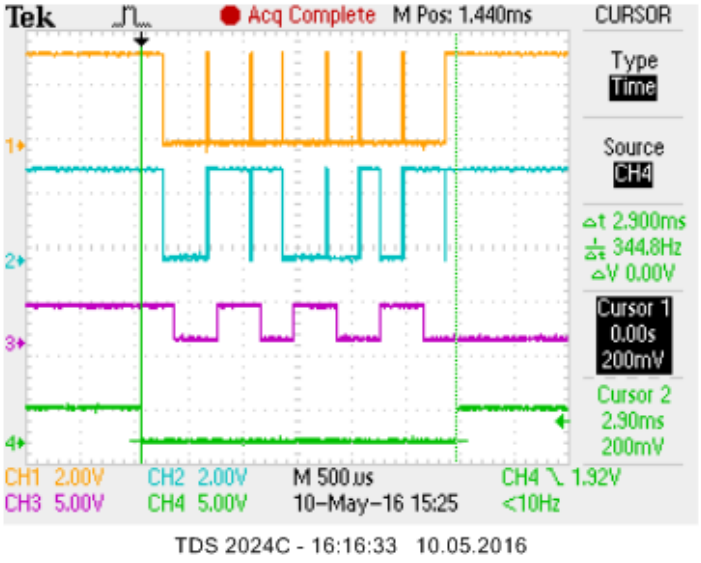
Voici une mesure de l'ensemble de la transaction avec un cycle de 100 us après adaptation du Timer et de la réponse à l'interruption.

Canal 1 : SCK
I2C-SCK / SCL2/RA2
PORTAbits.RA2 / pin 58

Canal 2 : SDA
I2C-SDA / SDa2/RA3
PORTAbits.RA3 / pin 59

Canal 3 : LED_5
toggle à chaque fin d'étape

Canal 4 : LED_6
marqueur enveloppe de la partie active



On parvient à mieux observer les actions I2C avec un cycle de 100 us. La durée d'exécution est de 2.9 ms.

8.4.10.3. DÉTAIL DÉBUT TRANSACTION CYCLE 100 US

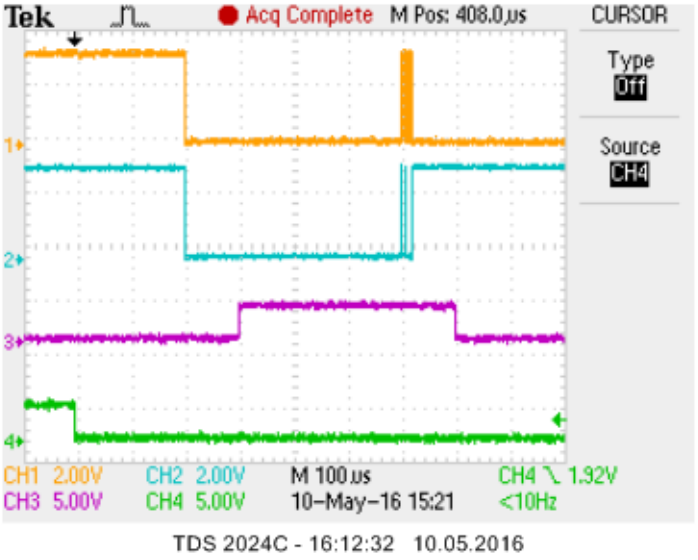
Voici une mesure de détail du début de la transaction :

Canal 1 : SCK
I2C-SCK / SCL2/RA2
PORTAbits.RA2 / pin 58

Canal 2 : SDA
I2C-SDA / SDa2/RA3
PORTAbits.RA3 / pin 59

Canal 3 : LED_5
toggle à chaque fin d'étape

Canal 4 : LED_6
marqueur enveloppe de la partie active



Les transitions de la LED_5 (canal 3) montrent qu'il y a quittance du i2c_start (flanc montant de LED_5 et quittance du 1er i2c_write, lors du flanc descendant.

8.4.11. CONCLUSION SUR LES FONCTIONS SM

Comme on peut le constater, l'introduction de machine d'état à deux niveaux rend un peu plus lourde l'écriture des fonctions mais permet d'utiliser la même systématique.

La découpe en fonction I2C de base comme start, stop, write et read permet d'avoir une correspondance avec les séquences décrites dans les datasheets des composants I2C. Cela évite aussi une séquence entièrement dédiée et comportant de 15 à 20 états.

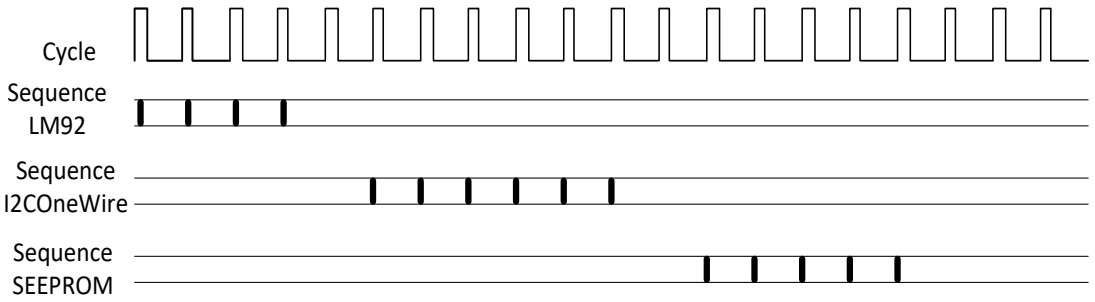
Avec l'utilisation du driver IC2 fourni par le MHC, on obtient l'adaptation au hardware lors des choix au niveau du configurateur.

8.5. UTILISATION EN MACHINE D'ÉTAT DE PLUSIEURS COMPOSANTS

Si on veut gérer sur le même bus I2C plusieurs composants, il est nécessaire d'effectuer les séquences l'une après l'autre.

8.5.1. ILLUSTRATION DU SÉQUENCEMENT DANS LE TEMPS

Pour illustrer cela, supposons que nous disposons du LM92, du I2C-OneWire et de la SEEPROM. Dans le temps, le traitement doit être réalisé de la manière suivante :



8.5.2. PRINCIPE RÉALISATION DU SÉQUENCEMENT DANS LE TEMPS

Si on appelle l'une après l'autre les fonctions de réalisation de séquence, on obtiendra un mélange des actions élémentaires I2C, ce qui ne fonctionnera pas du tout.

Pour réaliser le séquencement dans le temps il faut à nouveau introduire un switch avec une variable qui représente le capteur à exécuter.

Par exemple :

```
typedef enum { IC_LM92, IC_ DS2482, IC_EEPROM
             } E_Composants_I2C;

// Variable globale
E_Composants_I2C ComposantEnCours = IC_LM92;
```

Dans le traitement cyclique :

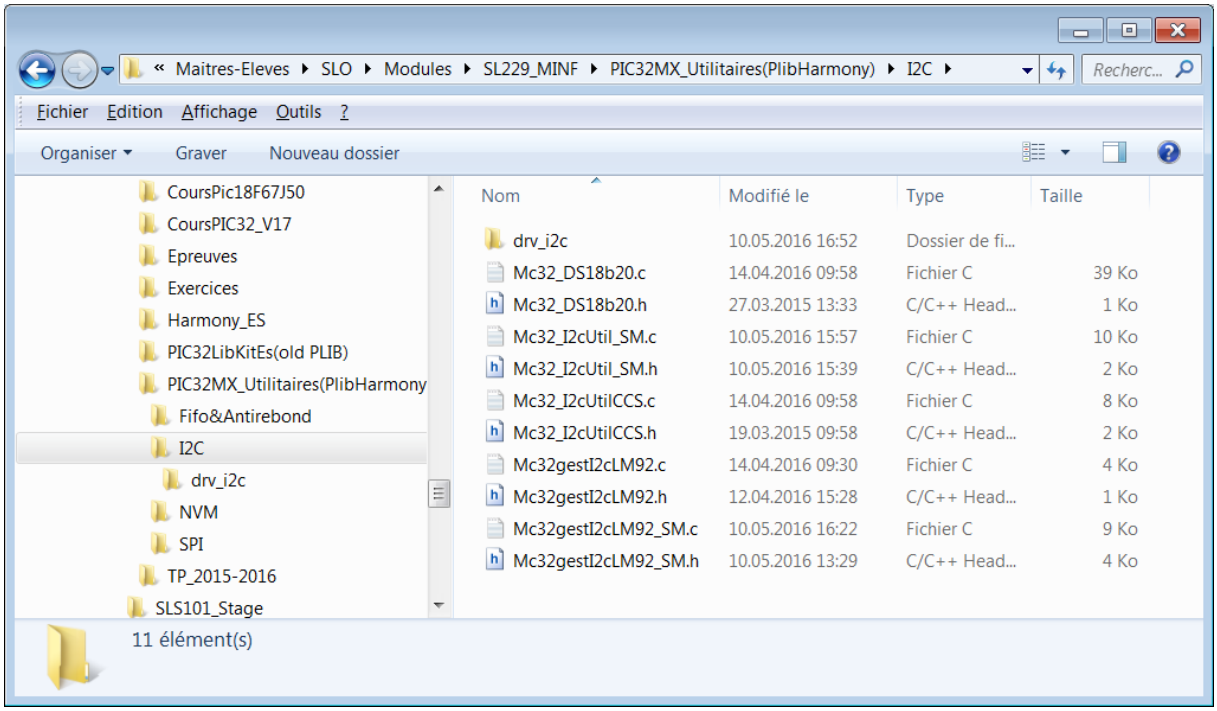
```
Switch (ComposantEnCours) {
    case IC_LM92 :
        // Appel cyclique traitement LM92
        I2C_LM92_SM_Execute(&DescrLm92);
        if (I2C_LM92_SM_IsReady(&DescrLm92)) {
            ComposantEnCours = IC_DS2482;
        }
    break;

    case IC_DS2482:
        // Appel cyclique traitement DS2482
        I2C_DS2482_SM_Execute(&DescrDs2482);
        if (I2C_DS2482_SM_IsReady(&DescrDs2482)) {
            ComposantEnCours = IC_EEPROM;
        }
    break;
```

```
case IC_EEPROM:
        // Appel cyclique traitement SEEPROM
        I2C_EEPROM_SM_Execute(&DescrEEprom);
        if (I2C_EEPROM_SM_IsReady(&DescrEEprom)) {
            ComposantEnCours = IC_LM92;
        }
    break;
} // end switch
```

Utilisation de la situation IsReady indiquant la fin de séquence pour passer au traitement suivant.

8.6. LOCALISATION DES FICHIERS I2C

Pour permettre un accès plus facile aux différentes librairies, vous trouverez sous : ...\Maitres-Eleves\SLO\Modules\SL229_MINF\PIC32MX_Utilitaires(PlibHarmony)\I2C les fichiers suivants :



On y trouve les deux familles d'utilitaires, l'une avec le traitement standard et l'autre par machine d'état. Le traitement du DS18B20 (OneWire) par machine d'état n'est pas fourni.

Le répertoire drv_i2c contient une copie du driver I2C avec les modifications.

8.7. CONCLUSION

Ce document devrait permettre, en s'inspirant des principes utilisés pour la gestion par machine d'état du LM92, de s'adapter à la gestion en machine d'état d'autres composants I2C.

8.8. HISTORIQUE DES VERSIONS

8.8.1. VERSION 1.5 MAI 2015

Reprise et transformation en cours T.P. de la partie SM du chapitre 9 de théorie version de mars 2014. Utilisation du driver I2C de Harmony 1.03.

8.8.2. VERSION 1.6 MAI 2016

Adaptation à Harmony 1.06 (très peu de changements par rapport à 1.03). Mesures refaites avec le projet Harmony 1.06.

8.8.3. VERSION 1.7 AVRIL 2017

Relecture générale par SCA. Test avec Harmony 1.08 et mise à jour.

8.8.1. VERSION 1.71 FÉVRIER 2022

Enlevé références à CCS et PIC18.