les écoles supérieures

MINF
Programmation des PIC32MX

Chapitre 9

Gestion du bus I2C

Théorie PIC32MX

Christian HUBER (CHR)
Serge CASTOLDI (SCA)
Version 1.91 février 2022

CONTENU DU CHAPITRE 9

9. GESTION DU BUS I2C AVEC LES PIC32MX

Ce chapitre traite du bus I2C et de sa gestion avec le microcontrôleur PIC32MX795F512L en utilisant le compilateur XC32 et le framework Harmony. Il présente la mise en œuvre des composants LM92 et DS2482-100. Ce dernier étant un convertisseur I2C-OneWire, ce chapitre contiendra une introduction à ce bus.

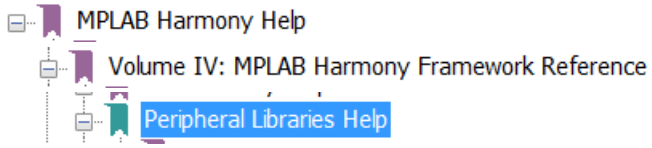
La librairie de gestion du bus I2C proposée dans ce document reprend le principe de décomposition d'une transaction I2C en actions de base, qui sont : start, restart, read, write et stop. Ce principe est issu de l'utilisation anciennement d'un autre compilateur (CCS), et a l'avantage de simplifier la gestion du bus I2C. Ainsi, il suffira au programmeur de décomposer la transaction I2C voulue en une série d'actions de base et d'appeler les fonctions adéquates dans l'ordre.

☝ Un désavantage de cette approche est le fait que ces fonctions sont bloquantes (elles incluent des boucles d'attente), ce qui est peu compatible avec un traitement cyclique rapide.

Le chapitre T.P. correspondant traite lui des fonctions I2C réalisées sous forme de machine d'état. Ce fonctionnement est donc non bloquant.

Les documents de référence sont :

- La documentation "PIC32 Family Reference Manual" :
  Section 24 : Inter-Integrated Circuit
- Pour les détails spécifiques au PIC32MX795F512L, il faut se référer au document "PIC32MX5XX/6XX/7XX Family Data Sheet" :
  Section 19 : Inter-Integrated Circuit (I2C)
- La documentation d'Harmony, qui se trouve dans <Répertoire Harmony>\v<n>\doc :
  Section MPLAB Harmony Framework Reference > Peripheral Libraries Help,
  sous-section I2C Peripheral Library



Ce document a été établi sur la base de Harmony v1.06.

9.1. BUS I2C

Avant d'étudier comment gérer le bus I2C avec les microcontrôleurs PIC32MX, il est nécessaire de connaître le bus I2C.

Le bus I2C, dont le sigle signifie *Inter Integrated Circuit*, ce qui donne IIC et par contraction I2C, a été proposé initialement par Philips mais est adopté de nos jours par de très nombreux fabricants. C'est un bus de communication de type série présent sur divers circuits d'interface spécialisés (convertisseurs A/D ou D/A, horloges temps réel, etc.) ainsi que sur de nombreuses mémoires EEPROM à accès série.

9.1.1. GENERALITES

Le bus I2C n'utilise que deux lignes de signal (et la masse correspondante bien sûr) et permet d'échanger des informations sous forme série. Ses points forts sont les suivants :

- C'est un bus série bifilaire utilisant une ligne de données appelée SDA (Serial DAta) et une ligne d'horloge appelée SCL (Serial CLock).
- Les données peuvent être échangées dans les deux sens. Etant donné qu'il n'y a que 2 lignes (data et clock), le bus reste half-duplex.
- Le bus est multi-maître.
- Dans la variante de base, chaque esclave dispose d'une adresse codée sur 7 bits. On peut donc en connecter simultanément 128 sur le même bus, avec des adresses différentes (sous réserve de ne pas le surcharger électriquement bien sûr).
  Une possibilité d'adressage 10 bits, cohabitant avec l'adressage 7 bits, existe.
- Une quittance (acknowledge) est générée pour chaque octet de donnée transféré.
- Selon les périphériques connectés, le bus peut travailler à une vitesse maximum de (état de la norme I2C version 3, 2007) 100 kb/s (standard mode), 400 kb/s (fast mode), 1 Mb/s (fast mode plus) ou encore 3,4 Mb/s (high speed mode).
  Un procédé automatique permet de ralentir l'équipement le plus rapide pour s'adapter à la vitesse de l'élément le plus lent lors d'un transfert.
- Le nombre maximum d'esclaves n'est limité que par la charge capacitive maximale du bus qui peut être de 400 pF. Ce nombre ne dépend donc que de la technologie des circuits et du mode de câblage employés.
- Les niveaux électriques permettent l'utilisation de circuits en technologies CMOS, NMOS ou TTL.

9.1.2. LES LIGNES DU BUS I2C

La Figure 9-1 montre le principe adopté au niveau des étages d'entrée/sortie des circuits d'interface au bus 12C. La partie entrée est représentée par un simple tampon, tandis que la partie sortie fait appel à une configuration à drain ouvert (l'équivalent en MOS du classique collecteur ouvert bipolaire). Ceci permet de réaliser des ET câblés par simple connexion des sorties SDA et SCL de tous les circuits.

Aucune charge n'étant prévue dans ces derniers, une résistance de rappel à une tension positive doit être mise en place. Le niveau électrique n'est pas précisé pour l'instant car il dépend de cette tension. Nous parlerons donc de niveaux logiques hauts ou "1" ou encore de niveaux logiques bas ou "0" étant entendu que l'on travaille en logique positive c'est-à-dire qu'un niveau haut correspond à une tension plus élevée qu'un niveau bas.

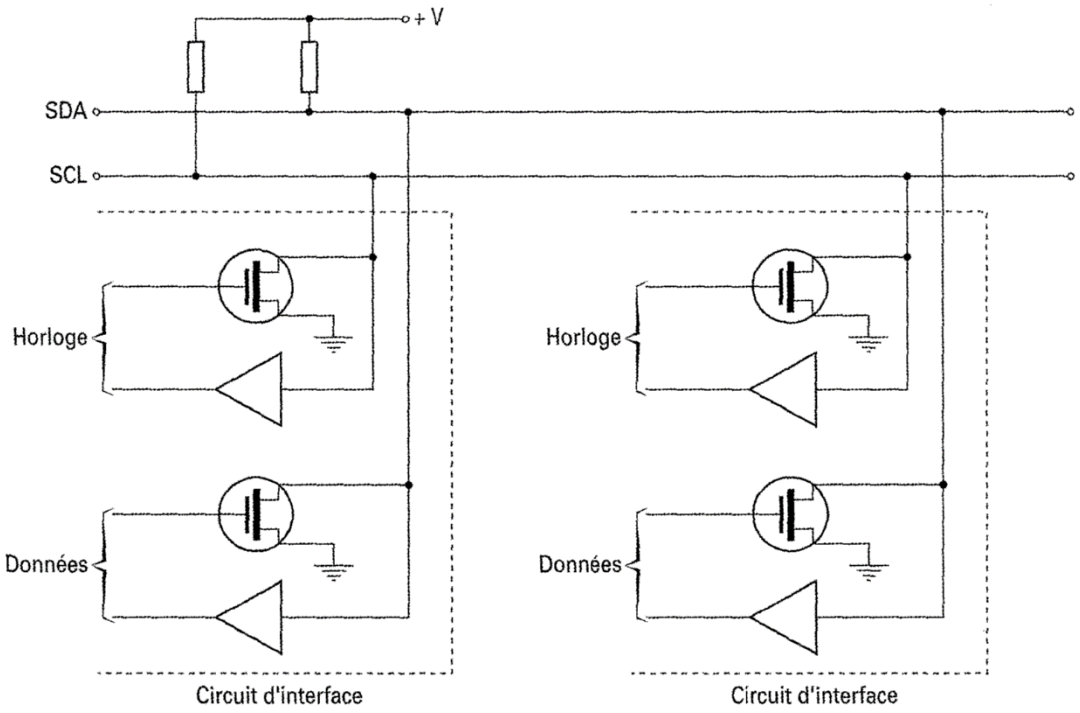


Figure 9-1

Compte tenu de ce mode de connexion en ET câblé, lorsque aucun périphérique I2C n'émet sur le bus, les lignes SDA et SCL sont au niveau haut qui est leur état de repos.

9.1.2.1. DIMENSIONNEMENT DES RESISTANCES DE PULL-UP

La Figure 9-2 montre la possibilité d'introduire des résistances série **Rs** pour protéger les composants des pics de tension.

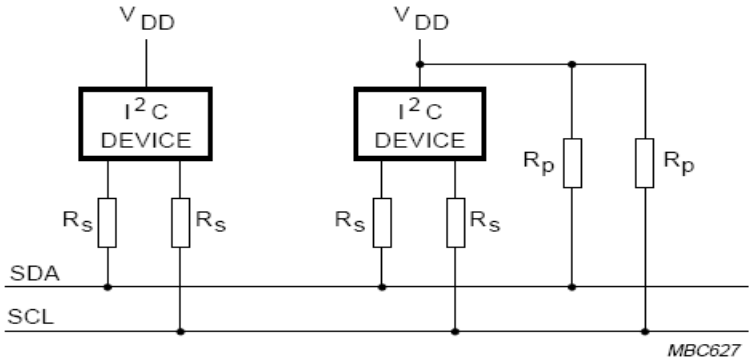
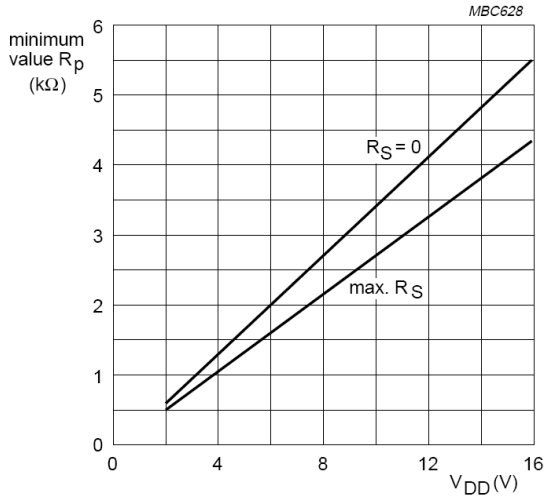
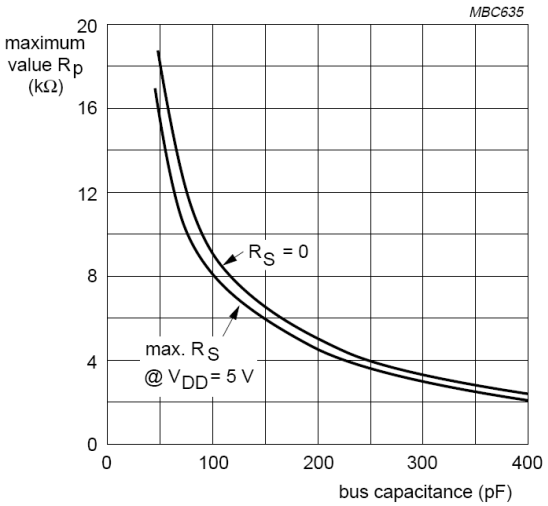


Figure 9-2

Les diagrammes ci-dessous permettent de déterminer la valeur de Rp en fonction de la tension d'alimentation et de la capacité du bus.



Valeur minimum de Rp en fonction de la tension d'alimentation et de Rs.



Valeur maximum de Rp en fonction de la capacité du bus de Rs

Remarque : au niveau du kit PIC32MX, la valeur de résistance utilisée est de 2.2 kΩ, ce qui permet une capacité de bus maximum d'environ 400 pF. La tension utilisée est de 3,3 V, donc il serait possible de mettre des résistances de plus faible valeur (min 1 kΩ).

9.1.3. PRINCIPE DE LA COMMUNICATION I2C

Pour illustrer le principe de la communication I2C, supposons que le maître veuille transmettre 2 octets de données à un esclave. La séquence de communication sera en principe la suivante :

Start	Adresse	W	ACK	Data 1	ACK	Data 2	ACK	Stop

La transmission commence par la condition de départ (Start), il s'agit d'un changement d'état des deux lignes SDA et SCL (activation).

Au début de toute communication figure l'adresse du composant destinataire. L'adresse est sur 7 bits, le 8ème bit servant à indiquer s'il s'agit d'une lecture ou d'une écriture. Dans notre exemple il s'agit d'une écriture. Cet octet est appelé octet d'en-tête.

La transmission des données se réalise par tranche de 8 bits. Dans notre exemple : 2 octets. A la suite de chaque octet il y a un bit d'acquittement provenant du destinataire.

La fin de transmission est signalée par la condition d'arrêt (Stop). Il s'agit d'un changement d'état des deux lignes SDA et SCL (retour à l'état de repos).

9.1.3.1. LA CONDITION DE DEPART

Une condition de départ est réalisée lorsque la ligne SDA passe du niveau haut au niveau bas alors que SCL est au niveau haut.

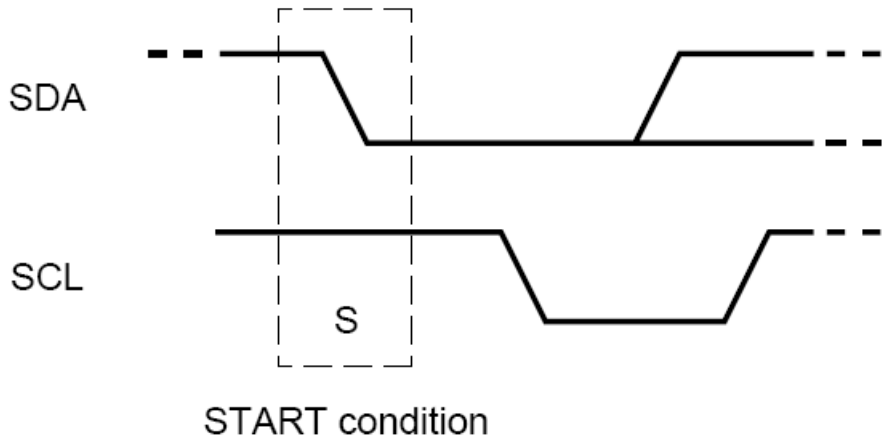
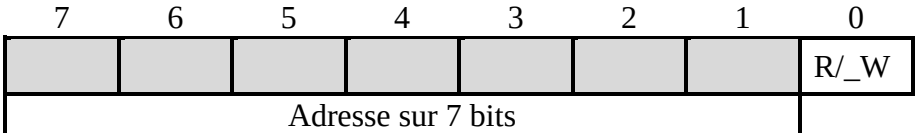


Figure 9-3 : condition de départ

9.1.3.2. L'OCTET D'EN-TETE

L'octet d'en-tête se compose de l'adresse (7 bits) et du bit de direction.



Valeur du bit R/_W :

- 0 = écriture, transfert maître → esclave
- 1 = lecture, transfert esclave → maître

9.1.3.3. PRINCIPE DE LA TRANSMISSION DES DONNEES

La Figure 9-4 illustre le principe de la transmission des données. Après la condition de départ, un octet est transmis en commençant par son bit de poids fort. Cet octet est suivi du bit d'acquittement. Il en est ainsi pour chaque octet.

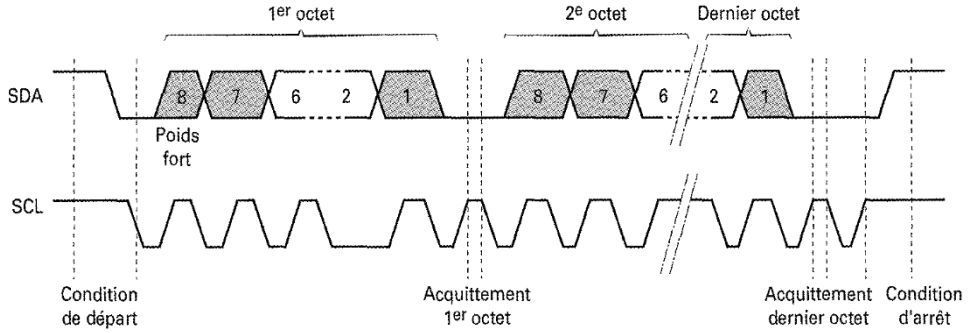


Figure 9-4

9.1.3.4. NOTION D'ACQUITTEMENT

La Figure 9-5 montre le détail de l'acquittement par le récepteur. A noter la nécessité du 9ème coup d'horloge.

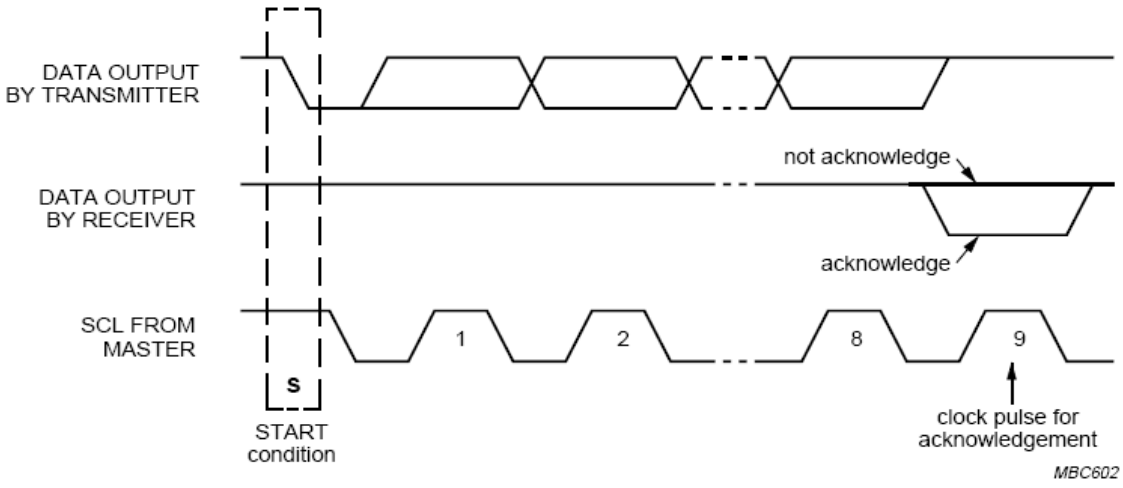


Figure 9-5

9.1.3.5. LA CONDITION D'ARRET

Une condition d'arrêt est réalisée lorsque SDA passe du niveau bas au niveau haut alors que SCL est au niveau haut.

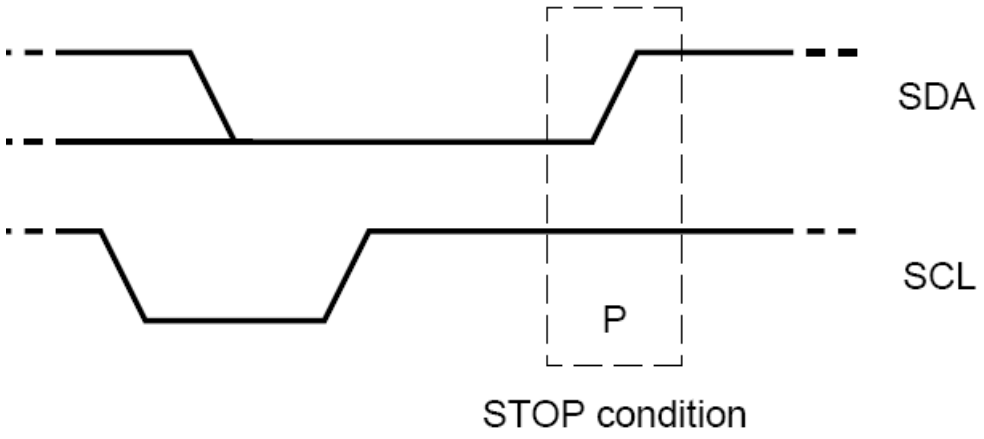


Figure 9-6 : condition d'arrêt

9.1.3.6. DETAIL DE LA TRANSMISSION D'UN BIT

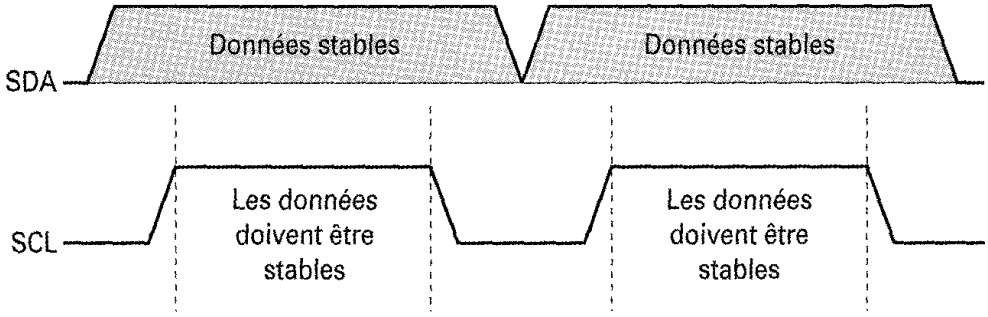


Figure 9-7

Comme le montre la Figure 9-7, une donnée n'est considérée comme valide sur le bus que lorsque le signal SCL est à l'état haut. L'émetteur doit donc positionner la donnée à émettre lorsque SCL est à l'état bas et la maintenir tant que SCL reste à l'état haut.

9.1.4. NOTION DE MAITRE ET D'ESCLAVES

Le maître est toujours celui qui initialise le transfert, fournit le signal d'horloge et termine le transfert. L'esclave est le composant adressé par le maître.

Dans une situation simple, le microcontrôleur est le maître et les différents composants des esclaves.

9.1.5. SITUATION MULTI-MASTER

Le bus I2C permet la présence de plusieurs maîtres. Cependant, un procédé d'arbitrage empêche la communication simultanée des maîtres.

La Figure 9-8 illustre la procédure d'arbitrage, qui est basée sur le niveau de la ligne SDA pendant que la ligne SCL est à 1. Si un des maîtres fournit un SDA à 1 et l'autre à 0, de par la situation de collecteur ouvert c'est la valeur 0 qui l'emporte. Le maître dont la valeur de donnée ne correspond pas à l'état de la ligne SDA perd la main et laissera la priorité à l'autre.

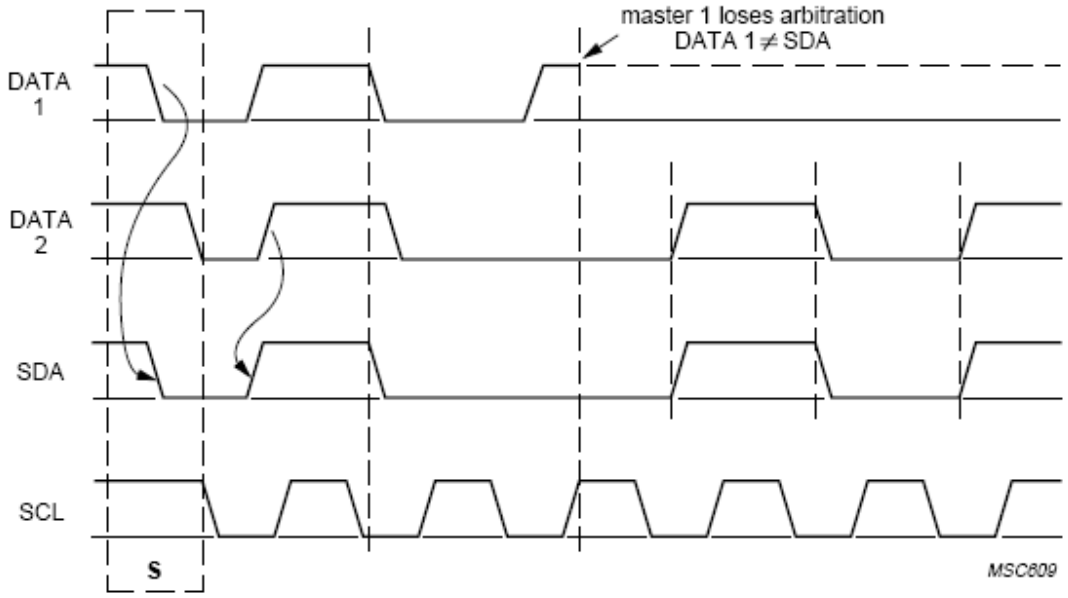
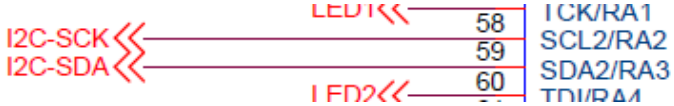


Figure 9-8 : arbitrage entre 2 maîtres

9.2. REALISATION DU BUS I2C AVEC LE PIC32MX795F512L

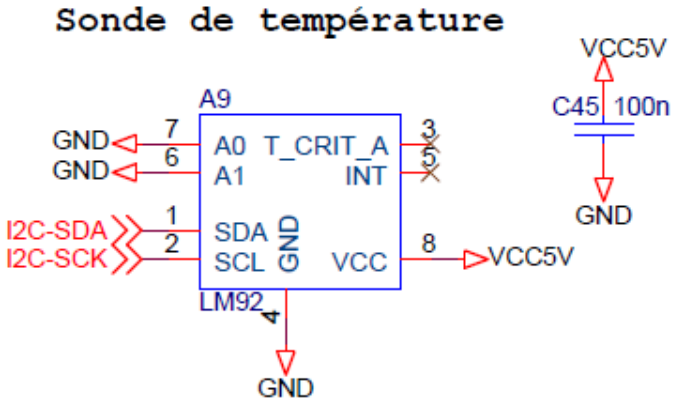
Le PIC32MX dispose de modules spécifiques pour la gestion du bus I2C. Dans le cadre du kit PIC32MX795F512L, c'est le module I2C no 2 qui est utilisé.



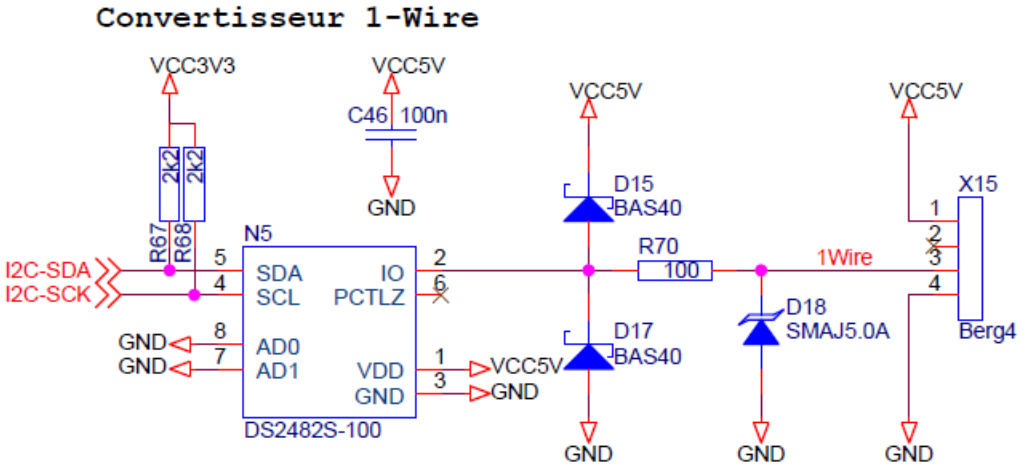
9.2.1. COMPOSANT I2C DU KIT PIC32MX795F512L

Il y a trois composants I2C sur le kit PIC32MX795F512L.

9.2.1.1. SONDE DE TEMPERATURE LM92



9.2.1.2. CONVERTISSEUR 1-WIRE DS2482S-100

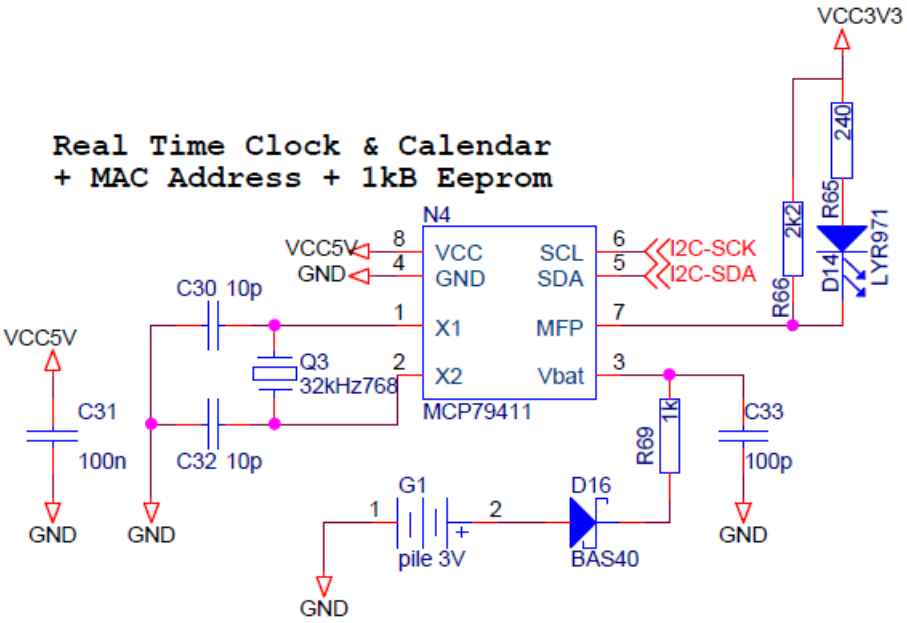


On remarque la paire de résistances de pull up pour le bus I2C.

9.2.1.3. RTC, MAC ADRESSE ET SEEPROM

Le MCP79411 intègre :

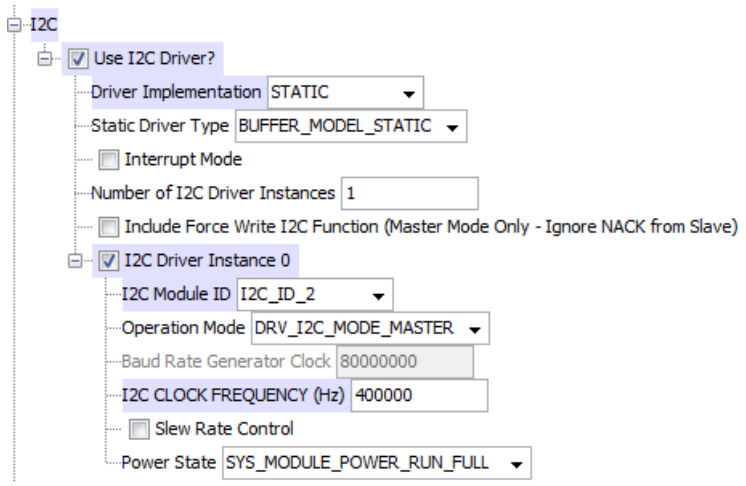
- un Real Time Clock/Calendar,
- une adresse MAC unique préprogrammée d'usine,
- une EEPROM de 1 Kbits, soit 128 octets.



9.3. DRIVER I2C FOURNI PAR HARMONY

Créons un driver I2C automatiquement via Harmony et observons le code généré.
Cet exemple a été effectué avec Harmony 1.08.

9.3.1. CONFIG DU DRIVER I2C



9.3.2. FONCTION D'INITIALISATION OBTENUE

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

    /* Low frequency is enabled (**NOTE** PLIB function logic reverted) */
    PLIB_I2C_HighFrequencyEnable(I2C_ID_2);   (2)

    i2c0Obj.i2cMode         = DRV_I2C_MODE_MASTER;
    i2c0Obj.transfersize    = 0;
    i2c0State               = DRV_I2C_TASK_SEND_DEVICE_ADDRESS;   (1)

    QueueInitialize_0();

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

1) Comme nous ne nous intéressons ici qu'à l'initialisation du périphérique I2C et non au fonctionnement du driver, la partie d'initialisation des variables peut être ignorée.

2) Cette partie correspond à la coche "Slew Rate Control" du MHC. La coche n'est pas mise, donc low frequency.
   - Comme l'indique le commentaire, Harmony a une incohérence et c'est la fonction PLIB_I2C_HighFrequencyEnable (au lieu de Disable) qui est appelée. Ce comportement pourrait être appelé à évoluer avec les versions de Harmony.
   - D'après le datasheet I2C du PIC32, le "slope control" devrait être activé pour des fréquences de clock de 100 kHz et 400 kHz. Toutefois, il est désactivé ici dû

à des problèmes d'incompatibilité du capteur de température LM92 avec des flancs trop lents.

☝ Ce code est fonctionnel. Toutefois, un code d'initialisation plus robuste sera présenté plus loin dans ce chapitre (cf. §9.4.4.3)

9.4. FONCTIONS I2C SOUHAITEES

Dans le but de pouvoir facilement mettre en œuvre un nouveau composant I2C ou de migrer la gestion d'un composant I2C réalisée à l'aide du compilateur CCS, vous trouverez dans les fichiers Mc32_I2cUtilCCS.h et Mc32_I2cUtilCCS.c les fonctions compatibles.

9.4.1. LES FONCTIONS DU COMPILATEUR CCS

Voici la liste des fonctions les plus courantes nécessaires à la gestion de l'I2C (documentation issue du compilateur CCS) :

i2c_start()	Issues a start command when in the I2C master mode.
i2c_write(data)	Sends a single byte over the I2C interface.
i2c_read()	Reads a byte over the I2C interface.
i2c_stop()	Issues a stop command when in the I2C master mode.

- Ajout également d'une fonction i2c_reStart() pour la répétition du start.

9.4.2. CONFIGURATION DE L'I2C

L'initialisation du bus I2C est faite par un appel à **i2c_init**() (anciennement réalisée via une directive).

9.4.3. CONTENU DU FICHIER MC32_I2CUTILCCS.H

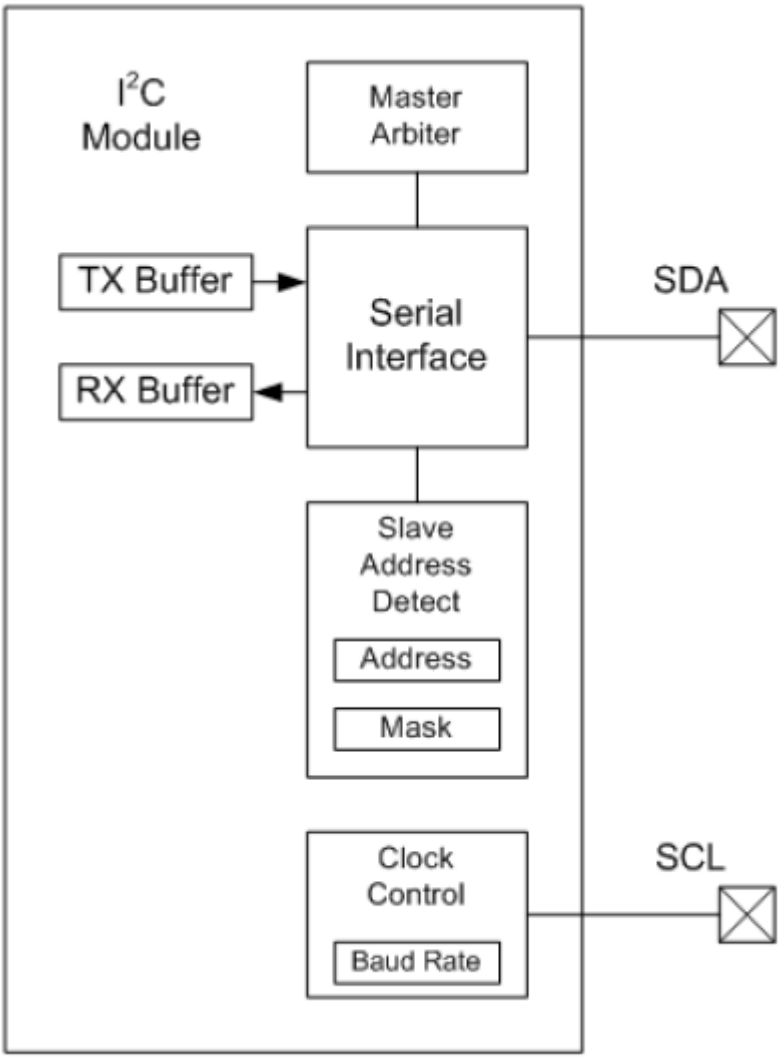
Voici les prototypes des fonctions I2C :

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


void i2c_init( bool Fast );

void i2c_start(void);
void i2c_reStart(void);
bool i2c_write( uint8_t data );
uint8_t i2c_read(bool ackTodo);
void i2c_stop( void );
```

9.4.4. REALISATION DES FONCTIONS

Ces fonctions utilisent des fonctions élémentaires de gestion du bus I2C. Elles ont été réalisées en utilisant les fonctions I2C de la PLIB.

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



9.4.4.1. VUE D'ENSEMBLE DES FONCTIONS DE LA PLIB_I2C

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

Baud Rate Generator Control Functions

	Name	Description
	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.

9.4.4.2. #INCLUDE NECESSAIRE ET DEFINITIONS

Nous avons besoin du fichier plib_i2c.h pour la librairie I2C. L'autre fichier est nécessaire pour la fonction SYS_CLK_PeripheralFrequencyGet.

```
#include "app.h"
#include "Mc32_I2cUtilCCS.h"
#include "peripheral\i2c\plib_i2c.h"
#include "peripheral\osc\plib_osc.h"

// KIT PIC32MX795F512L Constants
#define KIT_I2C_BUS   I2C_ID_2
#define I2C_CLOCK_FAST 400000
#define I2C_CLOCK_SLOW 100000
```

☝ Sur le kit, les différents périphériques I2C sont connectés au module I2C no 2.

9.4.4.3. LA FONCTION I2C_INIT

Cette fonction permet de définir la fréquence de travail du module I2C, de configurer le comportement et finalement d'enclencher le module. Sa réalisation s'inspire du driver i2c généré par Harmony (cf. §9.3), mais avec une petite adaptation.

```
// Initialisation de l'I2C
//      si BOOL Fast = FALSE   LOW speed  100 kHz
//      si BOOL Fast = TRUE   HIGH speed  400 kHz
void i2c_init( bool Fast )
{
    PLIB_I2C_Disable(KIT_I2C_BUS);      // Ajout CHR

    // LOW frequency is enabled (**NOTE** PLIB function logic reverted)
    // A 100k et 400kHz, on devrait activer le "slope control"
    // (cf. § I2C datasheet PIC32). Toutefois, le LM92 a des problèmes
    // d'incompatibilité avec les flancs trop lents => désactivé
    // Voir application note
    // "AN-2113 Applying I2C Compatible Temperature Sensors in Systems with Slow
        Clock Edges"
    PLIB_I2C_HighFrequencyEnable(KIT_I2C_BUS);
    if (Fast)  {
       PLIB_I2C_BaudRateSet(KIT_I2C_BUS,
               SYS_CLK_PeripheralFrequencyGet(CLK_BUS_PERIPHERAL_1),
                     I2C_CLOCK_FAST);
    } else {
        PLIB_I2C_BaudRateSet(KIT_I2C_BUS,
               SYS_CLK_PeripheralFrequencyGet(CLK_BUS_PERIPHERAL_1),
                     I2C_CLOCK_SLOW);
    }

    PLIB_I2C_SlaveClockStretchingEnable(KIT_I2C_BUS);  // ajout CHR

    PLIB_I2C_Enable(KIT_I2C_BUS);
}
```

Il est nécessaire de combiner les fonctions `PLIB_I2C_HighFrequencyEnable` et `PLIB_I2C_BaudRateSet` pour obtenir une fréquence correcte.

Cette solution a été testée avec Harmony 1.06 et 1.08 pour des fréquences d'horloges de 100 kHz et 400 kHz.

9.4.4.4. LA FONCTION I2C_START

Cette fonction génère la condition de départ en mode master. Elle a été réalisée en adaptant la fonction StartTransfer de l'ancien exemple Microchip.

```
void i2c_start(void)
{
    // Wait for the bus to be idle, then start the transfer
    while(PLIB_I2C_BusIsIdle(KIT_I2C_BUS) == 0);

    /* Check for recieve overflow */
    if ( PLIB_I2C_ReceiverOverflowHasOccurred(KIT_I2C_BUS))
    {
        PLIB_I2C_ReceiverOverflowClear(KIT_I2C_BUS);
    }

    /* Check for transmit overflow */
    if (PLIB_I2C_TransmitterOverflowHasOccurred(KIT_I2C_BUS))
    {
        PLIB_I2C_TransmitterOverflowClear(KIT_I2C_BUS);
    }


    PLIB_I2C_MasterStart(KIT_I2C_BUS);

    if (PLIB_I2C_ArbitrationLossHasOccurred(KIT_I2C_BUS));
    {
        // Handel bus collision
        PLIB_I2C_ArbitrationLossClear(KIT_I2C_BUS);
    }

    // Wait for the signal to complete
    while (PLIB_I2C_StartWasDetected(KIT_I2C_BUS) == false);
} // end i2c_start
```

9.4.4.1. LA FONCTION I2C_RESTART

Cette fonction permet de répéter le start. La différence est que l'on ne contrôle pas si le bus est en idle et que l'on utilise la fonction `PLIB_I2C_MasterStartRepeat`.

```
void i2c_reStart(void)
{
    // Pas d'attente bus en Idle
    /* Check for recieve overflow */
    if ( PLIB_I2C_ReceiverOverflowHasOccurred(KIT_I2C_BUS))
    {
        PLIB_I2C_ReceiverOverflowClear(KIT_I2C_BUS);
    }
    /* Check for transmit overflow */
    if (PLIB_I2C_TransmitterOverflowHasOccurred(KIT_I2C_BUS))
    {
        PLIB_I2C_TransmitterOverflowClear(KIT_I2C_BUS);
    }
   PLIB_I2C_MasterStartRepeat(KIT_I2C_BUS);

   if (PLIB_I2C_ArbitrationLossHasOccurred(KIT_I2C_BUS));
    {
        // Handel bus collision
        PLIB_I2C_ArbitrationLossClear(KIT_I2C_BUS);
    }
```

```
   // Wait for the signal to complete
   while (PLIB_I2C_StartWasDetected(KIT_I2C_BUS) == false);
} // end i2c_reStart
```

9.4.4.2. LA FONCTION I2C_WRITE

Cette fonction effectue l'envoi d'un octet sur le bus I2C.

Syntaxe : **i2c_write** (data);

Le paramètre data est la valeur 8 bits à transmettre.

En mode maître, la fonction génère le signal d'horloge. En outre, cette fonction retourne le bit ACK (0 = ACK, 1 = NO_ACK). Réalisation sur la base de la fonction TransmitOneByte de l'ancien exemple Microchip.

```
bool i2c_write( uint8_t data )
{
    bool  AckBit;

    // Wait for the bus to be idle (nécessaire aprés un reStart)
    while(PLIB_I2C_BusIsIdle(KIT_I2C_BUS) == false);

    // Wait for the transmitter to be ready
    while( PLIB_I2C_TransmitterIsBusy(KIT_I2C_BUS) == true);

    // Transmit the byte
    PLIB_I2C_TransmitterByteSend(KIT_I2C_BUS, data);

    // Wait as long as TBF = 1
    while(PLIB_I2C_TransmitterIsBusy(KIT_I2C_BUS));

    // Wait as long as TRSTAT == 1
    while(!PLIB_I2C_TransmitterByteHasCompleted(KIT_I2C_BUS));

    // Gestion du bit d'acknowledge
    AckBit = PLIB_I2C_TransmitterByteWasAcknowledged(KIT_I2C_BUS);

    return AckBit;
} // end i2c_write
```

9.4.4.3. LA FONCTION I2C_READ

Cette fonction effectue la lecture d'un octet sur le bus I2C.

Syntaxe : data = i2c_read (**ack**);

La fonction retourne l'octet lu.

Paramètre **ack** :

- 1 (true) signifie qu'il faut effectuer l'acquittement.
- 0 (false) signifie qu'il ne faut pas effectuer l'acquittement.

En mode maître, la fonction génère le signal d'horloge. Cette fonction est obtenue en partie à partir de l'ancien et du nouvel exemple Microchip.

```
uint8_t i2c_read(bool ackTodo)
{
    uint8_t i2cByte;

    BSP_LEDOn(BSP_LED_5); // provisoire : pour observation

// ajout idem driver statique I2C de Harmony 1_03
    if ( PLIB_I2C_ReceiverOverflowHasOccurred(KIT_I2C_BUS))
    {
        i2cByte = PLIB_I2C_ReceivedByteGet(KIT_I2C_BUS);
        PLIB_I2C_ReceiverOverflowClear(KIT_I2C_BUS);
    }

    // en relation avec stetching
    PLIB_I2C_SlaveClockRelease(KIT_I2C_BUS);

    //Set Rx enable in MSTR which causes SLAVE to send data
    PLIB_I2C_MasterReceiverClock1Byte(KIT_I2C_BUS);

    // Wait till RBF = 1; Which means data is available in I2C2RCV reg
    while(!PLIB_I2C_ReceivedByteIsAvailable(KIT_I2C_BUS));

    //Read from I2CxRCV
    i2cByte = PLIB_I2C_ReceivedByteGet(KIT_I2C_BUS);

    while ( PLIB_I2C_MasterReceiverReadyToAcknowledge
                                        ( KIT_I2C_BUS ) == false );

     if (ackTodo) {
          PLIB_I2C_ReceivedByteAcknowledge ( KIT_I2C_BUS, true );
     } else {
          PLIB_I2C_ReceivedByteAcknowledge ( KIT_I2C_BUS, false );
     }

    // wait till ACK/NACK sequence is complete i.e ACKEN = 0
    while( PLIB_I2C_MasterReceiverReadyToAcknowledge
                                            (KIT_I2C_BUS) == false);

    BSP_LEDOff(BSP_LED_5); // provisoire : pour observation

    return i2cByte;
} // end i2c_read
```

9.4.4.4. LA FONCTION I2C_STOP

Cette fonction génère la condition d'arrêt en mode master.

Syntaxe : **i2c_stop** ();

Réalisation sur la base du nouvel exemple Microchip.

```
void i2c_stop( void )
{
    // Attente bus en idle
    while(PLIB_I2C_BusIsIdle(KIT_I2C_BUS) == false);

    PLIB_I2C_MasterStop(KIT_I2C_BUS);

    // Wait for the signal to complete
    while (PLIB_I2C_StopWasDetected(KIT_I2C_BUS) == false);
} // end i2c_stop
```

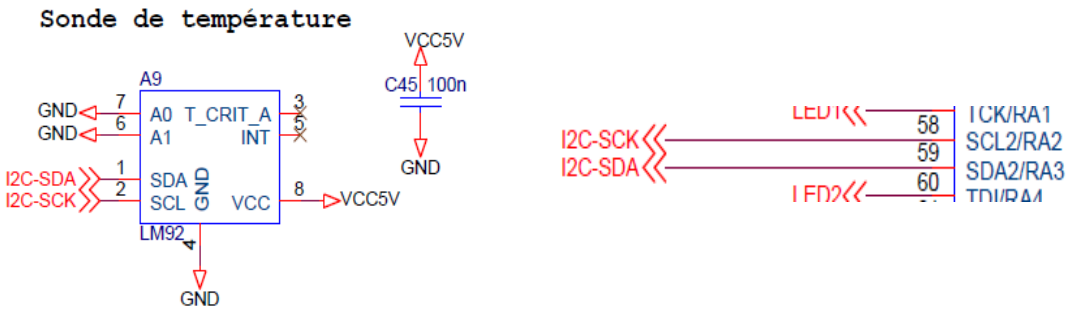
9.5. COMMUNICATION AVEC LE COMPOSANT LM92

Nous allons illustrer l'utilisation des fonctions I2C décrites précédemment en prenant comme exemple la communication avec le composant LM92, qui est un capteur de température avec bus I2C. Nous allons présenter deux façons de lire la température :

- Lecture simple (pointeur = registre température). Soit par défaut, soit suite à configuration.
- Etablissement du pointeur, puis lecture de la température. Cette méthode peut être adaptée pour lire une autre info que la température.

9.5.1. CONNEXION ENTRE LE PIC32MX ET LE LM92

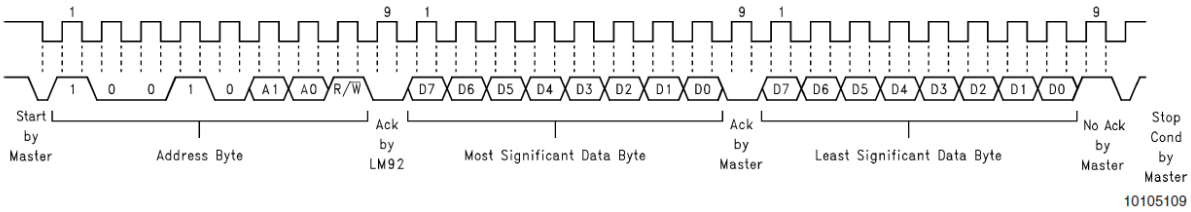
Utilisation du module I2C no 2, RA2 pour le signal SCL et RA3 pour le signal SDA.



Remarque : les résistances de pull-up de 2k2 sont présentes à un autre endroit du schéma.

9.5.2. LECTURE REGISTRE TEMPERATURE DU LM92

Pour lire le registre de température du LM92, il faut respecter la séquence indiquée par la Figure 9-9.



Typical 2-Byte Read From Preset Pointer Location Such as Temp or Comparison Registers

Figure 9-9

On remarque qu'il y a envoi de l'adresse avec le bit de poids faible à 1 pour indiquer la lecture. Il faut aussi noter que lors de la lecture du msb il y a acquittement par le maître, alors que pour la lecture du lsb, il n'y a pas d'acquittement.

9.5.2.1. REALISATION DE LA LECTURE

La fonction **I2C_ReadRawTempLM92** sert à lire les données brutes représentant la température. Cette fonction est fournie dans les fichiers Mc32gestI2cLm92.h et Mc32gestI2cLm92.c.

```
int16_t I2C_ReadRawTempLM92(void)
{
    // Déclaration des variables
    uint8_t msb = 1;
    uint8_t lsb = 1;
    int16_t RawTemp;

    BSP_LEDToggle(BSP_LED_6);  // provisoire : pour observation
    i2c_start();
    i2c_write(lm92_rd);     // adresse + lecture
    msb = i2c_read(1);      // ack
    lsb = i2c_read(0);      // no ack
    i2c_stop();

    BSP_LEDToggle(BSP_LED_6);  // provisoire : pour observation
    RawTemp = msb;
    RawTemp = RawTemp << 8;
    RawTemp = RawTemp | lsb;
    return RawTemp;
} // end I2C_ReadRawTempLM92
```

9.5.2.2. OBSERVATION DE LA SEQUENCE

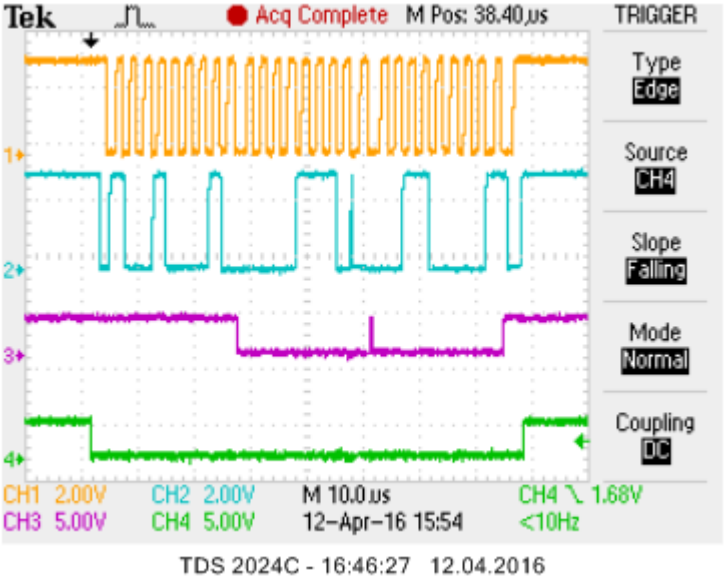
L'action sur la LED_6 marque le début et la fin de la fonction **I2C_ReadRawTempLM92.** La LED_5 est activée dans la fonction **i2c_read.**

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

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

Canal 3 : LED_5
marqueur de la fonction
i2c_read

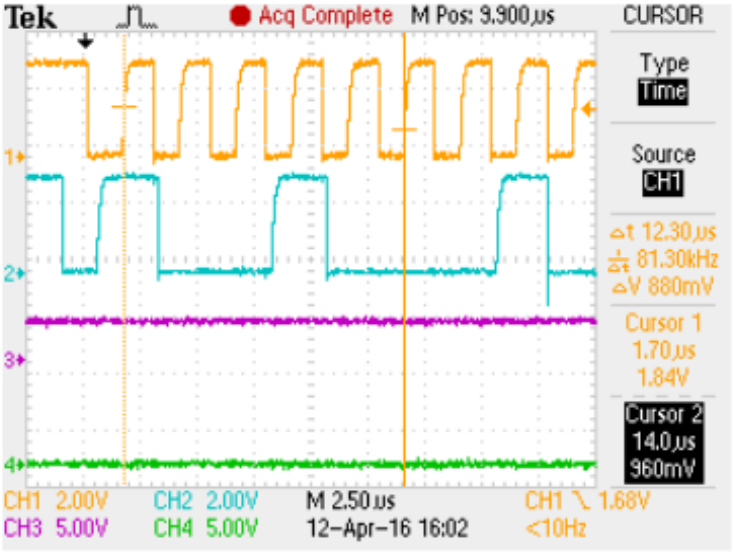
Canal 4 : LED_6
marqueur de la fonction
I2C_ReadRawTempLM92



On obtient bien 3 séries de 9 coups d'horloge. Et si on convertit la température brute en degré, on obtient un résultat cohérent.

9.5.2.3. CONTROLE DE LA PERIODE DE SCL EN FAST

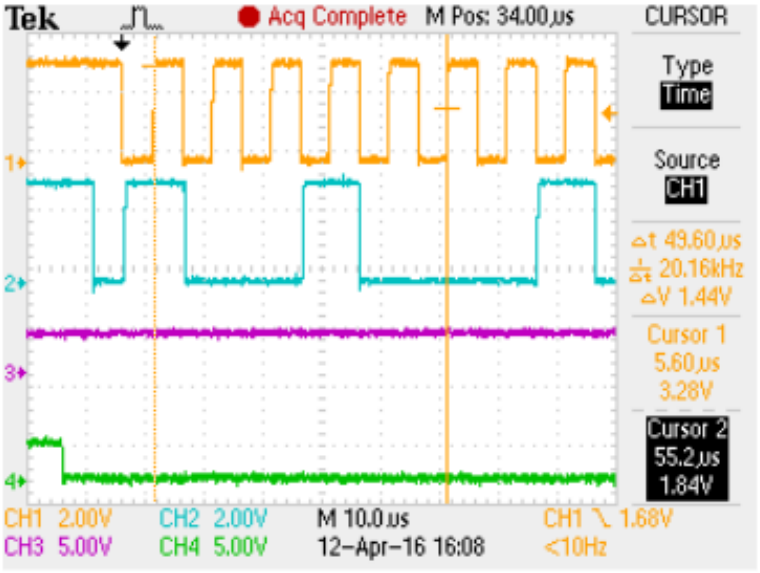
En modifiant la base de temps de la mesure précédente, il est possible avec les curseurs de mesurer la période du signal SCL.



La mesure de 5 périodes nous donne 12,3 us d'où 2,46 us pour une période, ce qui est assez proche des 2.5 us qui correspondent à 400 kHz. Cela nous donne une fréquence de 406,5 kHz que le LM92 semble supporter sans problème.

9.5.2.4. CONTROLE DE LA PERIODE DE SCL EN SLOW

En modifiant l'initialisation et en sélectionnant slow, on obtient :



La mesure de 5 période nous donne 49,6 us d'où 9,92 us pour une période, ce qui est assez proche des 10 us qui correspondent à 100 kHz. Cela nous donne une fréquence de 100,8 kHz.

9.5.3. SET POINTER ET LECTURE TEMPERATURE DU LM92

Pour lire la température, l'autre solution consiste à sélectionner le registre de température, puis à effectuer la lecture de la température. Voici la séquence à respecter pour cela.

La Figure 9-10 nous montre le début de la séquence (sélection du pointeur), tandis que la Figure 9-11 nous montre la partie lecture.

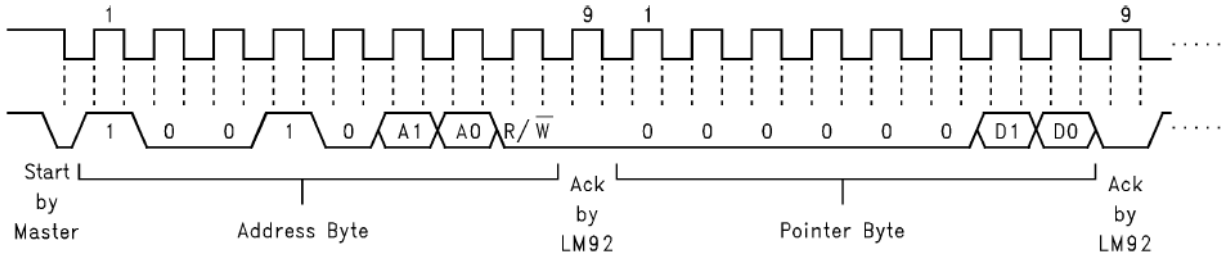


Figure 9-10

On observe que le bit R/_W est à 0 pour indiquer une action d'écriture. Le deuxième octet contient l'adresse du pointeur.

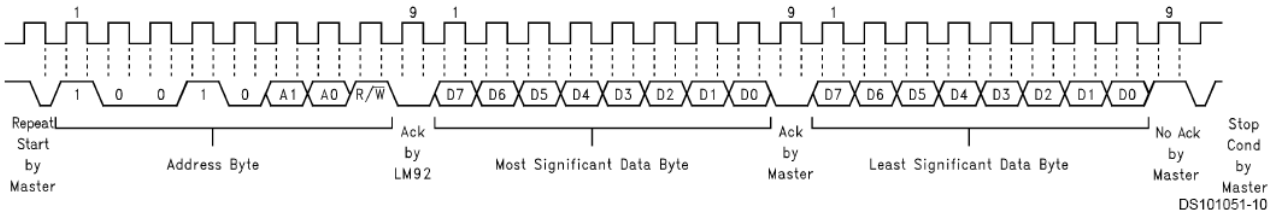


Figure 9-11

On remarque qu'il y a à nouveau envoi de l'adresse, mais cette fois-ci avec le bit de poids faible à 1 pour indiquer la lecture. Il faut aussi noter que lors de la lecture du msb, il y a acquittement par le maître, alors que pour la lecture du lsb il n'y a pas d'acquittement. Il y a répétition du start, raison de la réalisation d'une fonction i2c_reStart.

9.5.3.1. REALISATION DE LA LECTURE AVEC CONFIG

La fonction WriteCfgReadRawTempLM92 ci-dessous effectue la lecture en commençant par configurer le pointeur de lecture. On constate un 2ème start (réalisé par la fonction reStart) avec l'envoi à nouveau de l'adresse en indiquant la lecture. On obtient d'abord le poids fort et ensuite le poids faible. A noter la 1ère lecture avec acquittement, la 2ème sans.

```
int16_t I2C_WriteCfgReadRawTempLM92(void)
{
    //Déclaration des variables
    uint8_t msb = 1;
    uint8_t lsb = 1;
    int16_t RawTemp;

    BSP_LEDOn(BSP_LED_6);  // provisoire : pour observation

    i2c_start();
    i2c_write(lm92_wr);    // adresse + écriture
    i2c_write(lm92_temp_ptr);    // sélection ptr. temp.

    i2c_reStart();
    i2c_write(lm92_rd);    // adresse + lecture
    msb = i2c_read(1);     // ack
    lsb = i2c_read(0);     // no ack
    i2c_stop();

    BSP_LEDOff(BSP_LED_6);  // provisoire : pour observation
    RawTemp = msb;
    RawTemp = RawTemp << 8;
    RawTemp = RawTemp | lsb;
    return RawTemp;
} // end I2C_WriteCfgReadRawTempLM92
```

9.5.3.2. OBSERVATION DE LA SEQUENCE

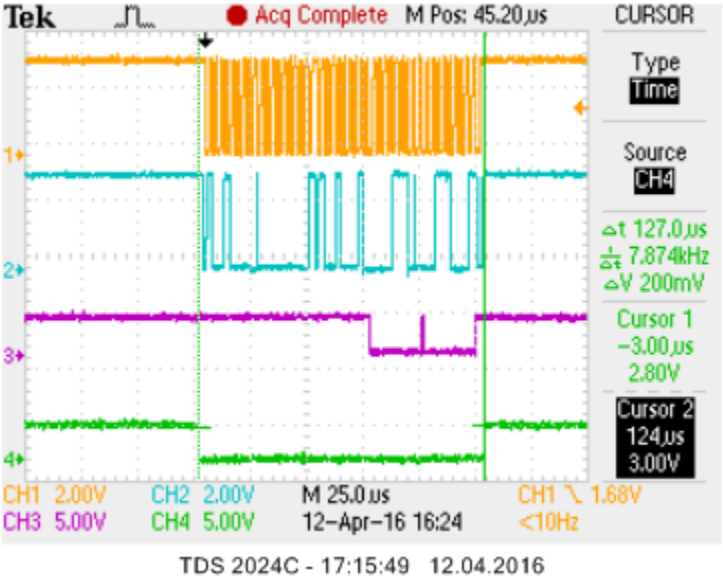
L'action sur la LED_6 marque le début et la fin de la fonction **I2C_WriteCfgReadRawTempLM92.** La LED_5 est activée dans la fonction **i2c_read.**

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

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

Canal 3 : LED_5
marqueur de la fonction
i2c_read

Canal 4 : LED_6
marqueur de la fonction
I2C_ReadRawTempLM92



On observe les 5 séries de 9 coups d'horloge. Le marqueur de l'action i2c_read permet de bien repérer l'action de lecture.

La mesure au curseur du signal marquant l'exécution de la fonction nous indique 127 us.

9.5.4. CONFIGURATION DU I2C POUR LE LM92

Avant de pouvoir utiliser les fonctions I2C, il est nécessaire d'initialiser le module I2C.

Ceci est réalisé par la fonction I2C_InitLM92, qui utilise la fonction i2c_init ainsi que la fonction I2C_WriteConfigLM92.

```
void I2C_InitLM92(void)  {

   bool Fast = true;
   i2c_init( Fast );
   I2C_WriteConfigLM92();
}
```

9.5.4.1. LA FONCTION I2C_WRITECONFIGLM92

Cette fonction établit le pointeur sur le registre de température et effectue la lecture de la température. En s'inspirant de cette fonction, il est possible de configurer les alarmes du LM92.

```
void I2C_WriteConfigLM92(void)
{
   //Déclaration des variables
    uint8_t tmp;

    i2c_start();
    i2c_write(lm92_wr);          // adresse + écriture
    i2c_write(lm92_temp_ptr);    // sélection ptr. temp.
    i2c_reStart();
    i2c_write(lm92_rd);    // adresse + lecture
    tmp = i2c_read(1);     // ack
    tmp = i2c_read(0);     // no ack
    i2c_stop();
}
```

9.5.4.2. DEFINITIONS DE L'ADRESSE DU LM92

Le fabricant indique que l'adresse est sur 7 bits avec 5 bits fixes (propre au LM92) et 2 configurables par les lignes A0 et A1. Ces 2 lignes étant connectées à la masse on obtient la situation suivante pour la valeur de l'adresse :

7	6	5	4	3	2	1	0
1	0	0	1	0	A1=0	A0=0	R/_W

Adresse sur 7 bits

D'où les deux définitions :

```
// Définition pour LM92
#define lm92_rd    0x91  // lm92 address for read
#define lm92_wr    0x90  // lm92 address for write
```

9.5.4.3. DEFINITIONS DU POINTEUR DE TEMPERATURE

L'organisation de la valeur 8 bits pour la sélection du pointeur est la suivante:

P7	P6	P5	P4	P3	P2	P1	P0
0	0	0	0	0	Register Select		

P2	P1	P0	Register
0	0	0	Temperature (Read only) (Power-up default)
0	0	1	Configuration (Read/Write)
0	1	0	T_{HYST} (Read/Write)
0	1	1	T_CRIT (Read/Write)
1	0	0	T_{LOW} (Read/Write)
1	0	1	T_{HIGH} (Read/Write)
1	1	1	Manufacturer's ID

Ce qui conduit à la définition suivante :

```
#define lm92_temp_ptr  0x00   // adr. pointeur température
```

9.5.4.4. ORGANISATION DU REGISTRE DE TEMPERATURE

L'exemple de code fournit 2 octets correspondant au registre de température. Pour afficher la température il est nécessaire d'effectuer un regroupement sur 16 bits.
Puis d'effectuer un décalage de 3 bits à droite ou de diviser par 8.

1.10 TEMPERATURE REGISTER

(Read Only):

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Sign	MSB	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	CRIT	HIGH	LOW
													Status Bits		

D0–D2: Status Bits

D3–D15: Temperature Data. One LSB = 0.0625°C. Two's complement format.

Le bit de poids faible représente 0.0625 degré soit 1/16, si on souhaite exprimer la température en degré, il faut multiplier par 0.0625.

9.5.4.5. LA FONCTION CONVRAWTODEG

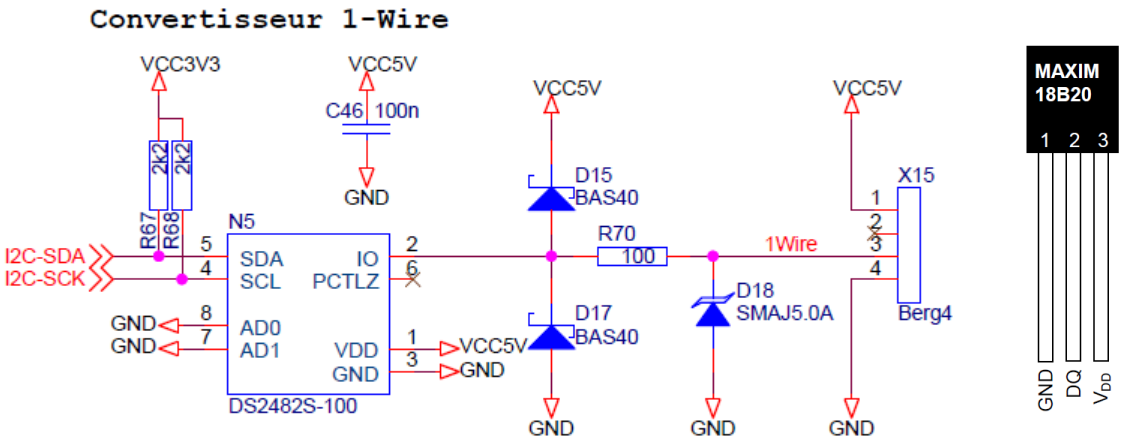
La fonction LM92_ConvRawToDeg effectue la conversion de la température brute en une valeur en degré.

Utilisation d'un passage par référence suite à des problèmes rencontrés lors de l'utilisation dans une réponse à l'interruption.

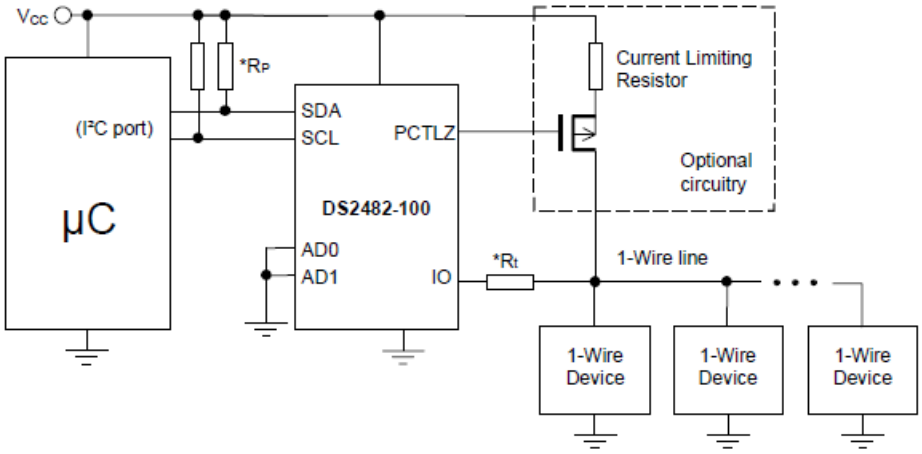
```
void LM92_ConvRawToDeg( int16_t RowTemp, float *pTemp)
{
    float TempLoc;

    RowTemp = RowTemp / 8;
    // bit poids faible = 0.0625 degré
    TempLoc = RowTemp * 0.0625;
    *pTemp = TempLoc;
}
```

9.6. MISE EN ŒUVRE DU DS2482-100

Le DS2482-100 étant un convertisseur I2C en OneWire, nous allons donc connecter un composant OneWire comme un capteur de température DS18B20 afin de pouvoir réaliser un test complet.



9.6.1. VUE D'ENSEMBLE DE PRINCIPE



9.6.2. FONCTIONNEMENT DU DS2482-100

Le DS2482-100 gère le signal OneWire. Il exécute 8 commandes différentes nécessaires à la communication. Elles sont réparties en 4 catégories, device control, I²C communication, 1-Wire set-up and 1-Wire communication.

Voici sans trop de détails ni traduction, ces commandes. Avec tout d'abord le détail du registre de statut du DS2482-100.

9.6.2.1. REGISTRE DE STATUT DU DS2482-100

Status Register Bit Assignment

bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
DIR	TSB	SBR	RST	LL	SD	PPD	1WB

9.6.2.1.1. Détail des bits du registre

1-Wire Busy (1WB)
The 1WB bit reports to the host processor whether the 1-Wire line is busy. During 1-Wire communication 1WB is 1; once the command is completed, 1WB returns to its default 0. Details on when 1WB changes state and for how long it remains at 1 are found in the *Function Commands* section.

Presence-Pulse Detect (PPD)
The PPD bit is updated with every 1-Wire Reset command. If the DS2482 detects a presence pulse from a 1-Wire device at t_{MSP} during the Presence Detect cycle, the PPD bit will be set to 1. This bit returns to its default 0 if there is no presence pulse or if the 1-Wire line is shorted during a subsequent 1-Wire Reset command.

Short Detected (SD)
The SD bit is updated with every 1-Wire Reset command. If the DS2482 detects a logic 0 on the 1-Wire line at t_{SI} during the Presence Detect cycle, the SD bit is set to 1. This bit returns to its default 0 with a subsequent 1-Wire Reset command provided that the short has been removed. If SD is 1, PPD is 0. The DS2482 cannot distinguish between a short and a DS1994 or DS2404 signaling a 1-Wire interrupt. For this reason, if a DS2404/DS1994 is used in the application, the interrupt function must be disabled. The interrupt signaling is explained in the respective device data sheets.

Logic Level (LL)
The LL bit reports the logic state of the active 1-Wire line without initiating any 1-Wire communication. The 1-Wire line is sampled for this purpose every time the Status register is read. The sampling and updating of the LL bit takes place when the host processor has addressed the DS2482 in read mode (during the acknowledge cycle), provided that the Read Pointer is positioned at the Status register.

Device Reset (RST)
If the RST bit is 1, the DS2482 has performed an internal reset cycle, either caused by a power-on reset or from executing the Device Reset command. The RST bit is cleared automatically when the DS2482 executes a Write Configuration command to restore the selection of the desired 1-Wire features.

Single Bit Result (SBR)
The SBR bit reports the logic state of the active 1-Wire line sampled at t_{MSR} of a 1-Wire Single Bit command or the first bit of a 1-Wire Triplet command. The power-on default of SBR is 0. If the 1-Wire Single Bit command sends a 0-bit, SBR should be 0. With a 1-Wire Triplet command, SBR could be 0 as well as 1, depending on the response of the 1-Wire devices connected. The same result applies to a 1-Wire Single Bit command that sends a 1-bit.

Triplet Second Bit (TSB)
The TSB bit reports the logic state of the active 1-Wire line sampled at t_{MSR} of the second bit of a 1-Wire Triplet command. The power-on default of TSB is 0. This bit is updated only with a 1-Wire Triplet command and has no function with other commands.

Branch Direction taken (DIR)
Whenever a 1-Write Triplet command is executed, this bit reports to the host processor the search direction that was chosen by the 3rd bit of the triplet. The power-on default of DIR is 0. This bit is updated only with a 1-Wire Triplet command and has no function with other commands. For additional information see the description of the 1-Wire Triplet command and the Dallas Application Note 187, "1-Wire Search Algorithm".

9.6.2.2. COMMAND, DEVICE RESET

Device Reset

Command Code	F0h
Command Parameter	None
Description	Performs a global reset of device state machine logic. Terminates any ongoing 1-Wire communication.
Typical Use	Device initialization after power-up; re-initialization (reset) as desired.
Restriction	None (can be executed at any time)
Error Response	None
Command Duration	Maximum 525ns, counted from falling SCL edge of the command code acknowledge bit.
1-Wire Activity	Ends maximum 262.5ns after the falling SCL edge of the command code acknowledge bit.
Read Pointer Position	Status register (for busy polling)
Status Bits Affected	RST set to 1, 1WB, PPD, SD, SBR, TSB, DIR set to 0
Configuration Bits Affected	1WS, APU, PPM, SPU set to 0

9.6.2.3. COMMAND, SET READ POINTER

Set Read Pointer

Command Code	E1h
Command Parameter	Pointer Code
Description	Sets the Read Pointer to the specified register. Overwrites the read pointer position of any 1-Wire communication command in progress.
Typical Use	To prepare reading the result from a 1-Wire Byte command; random read access of registers.
Restriction	None (can be executed at any time)
Error Response	If the pointer code is not valid, the pointer code is not acknowledged and the command is ignored.
Command Duration	None; the read pointer is updated on the rising SCL edge of the pointer code acknowledge bit.
1-Wire Activity	Not affected
Read Pointer Position	As specified by the pointer code
Status Bits Affected	None
Configuration Bits Affected	None

Valid Pointer Codes

Register Selection	Code
Status Register	F0h
Read Data Register	E1h
Configuration Register	C3h

9.6.2.4. COMMAND, WRITE CONFIGURATION

Write Configuration

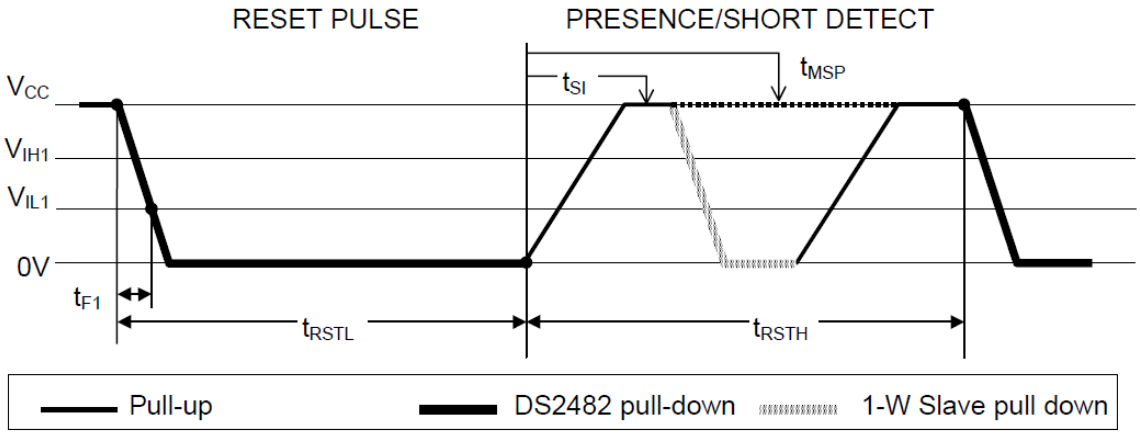
Command Code	D2h
Command Parameter	Configuration Byte
Description	Writes a new configuration byte. The new settings take effect immediately. **NOTE:** When writing to the Configuration register, the new data is accepted only if the upper nibble (bits 7 to 4) is the one's complement of the lower nibble (bits 3 to 0). When read, the upper nibble is always 0h.
Typical Use	Defining the features for subsequent 1-Wire communication.
Restriction	1-Wire activity must have ended before the DS2482 can process this command.
Error Response	Command code and parameter are not acknowledged if 1WB = 1 at the time the command code is received and the command is ignored.
Command Duration	None; the Configuration register is updated on the rising SCL edge of the configuration byte acknowledge bit.
1-Wire Activity	None
Read Pointer Position	Configuration register (to verify write)
Status Bits Affected	RST set to 0
Configuration Bits Affected	1WS, SPU, PPM, APU updated

9.6.2.5. COMMAND, 1-WIRE RESET

1-Wire Reset

Command Code	B4h
Command Parameter	None
Description	Generates a 1-Wire Reset/Presence Detect cycle (Figure 5) at the 1-Wire line. The state of the 1-Wire line is sampled at t_{SI} and t_{MSP} and the result is reported to the host processor through the Status register, bits PPD and SD.
Typical Use	To initiate or end any 1-Wire communication sequence.
Restriction	1-Wire activity must have ended before the DS2482 can process this command.
Error Response	Command code is not acknowledged if 1WB = 1 at the time the command code is received and the command is ignored.
Command Duration	t_{RSTL} + t_{RSTH} + maximum 262.5ns, counted from the falling SCL edge of the command code acknowledge bit.
1-Wire Activity	Begins maximum 262.5ns after the falling SCL edge of the command code acknowledge bit.
Read Pointer Position	Status register (for busy polling)
Status Bits Affected	1WB (set to 1 for t_{RSTL} + t_{RSTH}), PPD is updated at t_{RSTL} + t_{MSP}, SD is updated at t_{RSTL} + t_{SI}
Configuration Bits Affected	1WS, PPM, APU apply

9.6.2.6. SIGNAL ISSU DE LA COMMAND 1-WIRE RESET



For presence pulse masking and pull-up details see Figure 3.

9.6.2.7. COMMAND, 1-WIRE SINGLE BIT

1-Wire Single Bit

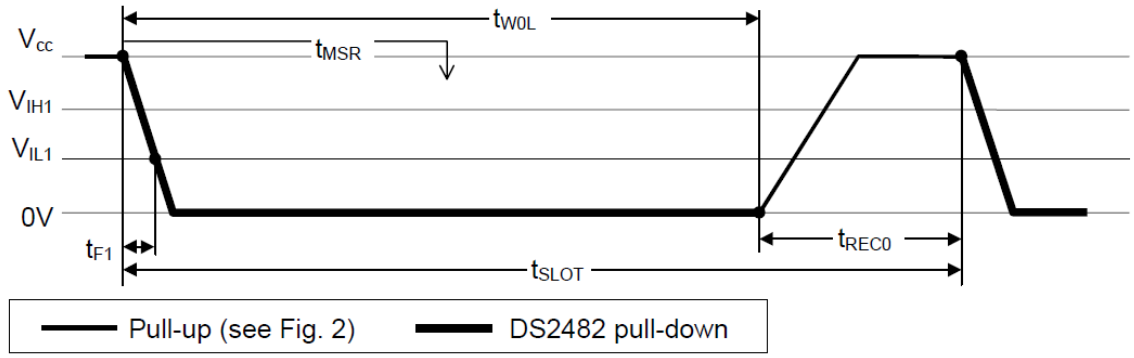
Command Code	87h
Command Parameter	Bit Byte
Description	Generates a single 1-Wire time slot with a bit value 'V' as specified by the bit byte at the 1-Wire line. A 'V' value of 0b generates a write-zero time slot (Figure 6), a value of 1b generates a write-one slot, which also functions as a read-data time slot (Figure 7). In either case the logic level at the 1-Wire line is tested at t_{MSR} and SBR is updated.
Typical Use	To perform single bit writes or reads at the 1-Wire line when single bit communication is necessary (the exception).
Restriction	1-Wire activity must have ended before the DS2482 can process this command.
Error Response	Command code and bit byte are not acknowledged if 1WB = 1 at the time the command code is received and the command is ignored.
Command Duration	t_{SLOT} + maximum 262.5ns, counted from the falling SCL edge of the first bit (MS bit) of the bit byte.
1-Wire Activity	Begins maximum 262.5ns after the falling SCL edge of the MS bit of the bit byte.
Read Pointer Position	Status register (for busy polling and data reading)
Status Bits Affected	1WB (set to 1 for t_{SLOT}) SBR is updated at t_{MSR} DIR (may change its state)
Configuration Bits Affected	1WS, APU, SPU apply

Bit Allocation in the Bit Byte

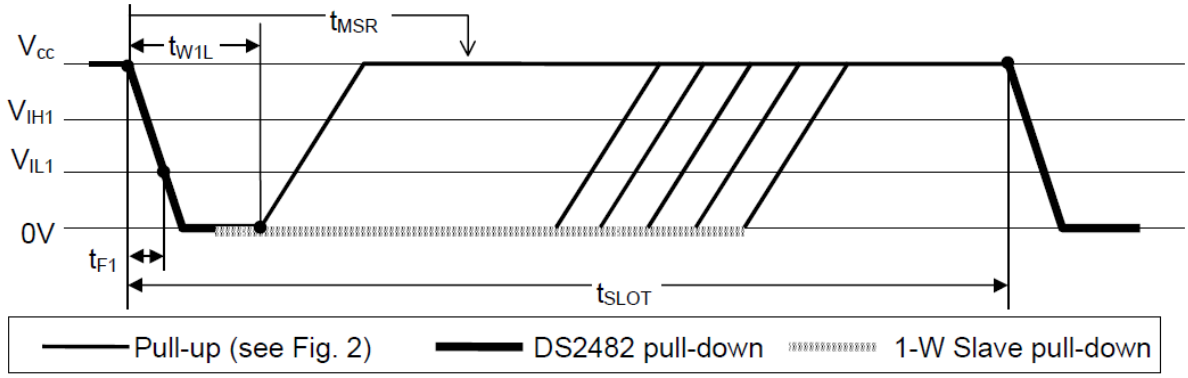
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
V	x	x	x	x	x	x	x

x = don't care

9.6.2.7.1. Timing pour écriture bit à 0



9.6.2.7.2. Timing écriture bit à 1 et lecture data



9.6.2.8. COMMAND, 1-WIRE WRITE BYTE

1-Wire Write Byte

Command Code	A5h
Command Parameter	Data Byte
Description	Writes single data byte to the 1-Wire line.
Typical Use	To write commands or data to the 1-Wire line; equivalent to executing eight 1-Wire Single Bit commands, but faster due to less I²C traffic.
Restriction	1-Wire activity must have ended before the DS2482 can process this command.
Error Response	Command code and data byte are not acknowledged if 1WB = 1 at the time the command code is received and the command will be ignored.
Command Duration	8 × t_{SLOT} + maximum 262.5ns, counted from falling edge of the last bit (LS bit) of the data byte.
1-Wire Activity	Begins maximum 262.5ns after falling SCL edge of the LS bit of the data byte (i.e., before the data byte acknowledge). **NOTE**: The bit order on the I²C bus and the 1-Wire line is different. (1-Wire: LS-bit first; I²C: MS-bit first) Therefore, 1-Wire activity cannot begin before the DS2482 has received the full data byte.
Read Pointer Position	Status register (for busy polling)
Status Bits Affected	1WB (set to 1 for 8 × t_{SLOT})
Configuration Bits Affected	1WS, SPU, APU apply

9.6.2.9. COMMAND, 1-WIRE READ BYTE

1-Wire Read Byte

Command Code	96h
Command Parameter	None
Description	Generates eight read-data time slots on the 1-Wire line and stores result in the Read Data register.
Typical Use	To read data from the 1-Wire line; equivalent to executing eight 1-Wire Single Bit commands with V = 1 (write-1 time slot), but faster due to less I²C traffic.
Restriction	1-Wire activity must have ended before the DS2482 can process this command.
Error Response	Command code is not acknowledged if 1WB = 1 at the time the command code is received and the command is ignored.
Command Duration	8 × t_{SLOT} + maximum 262.5ns, counted from the falling SCL edge of the command code acknowledge bit.
1-Wire Activity	Begins maximum 262.5ns after the falling SCL edge of the command code acknowledge bit.
Read Pointer Position	Status register (for busy polling) **NOTE**: To read the data byte received from the 1-Wire line, issue the Set Read Pointer command and select the Read Data register. Then access the DS2482 in read mode.
Status Bits Affected	1WB (set to 1 for 8 × t_{SLOT})
Configuration Bits Affected	1WS, APU apply

9.6.2.10. COMMAND, 1-WIRE TRIPLET

1-Wire Triplet

Command Code	78h
Command Parameter	Direction Byte
Description	Generates three times slots, two read time slots and one write time slot at the 1-Wire line. The type of write time slot depends on the result of the read time slots and the direction byte. The direction byte determines the type of write time slot if both read time slots are 0 (a typical case). In this case the DS2482 generates a write 1-time slot if V = 1 and a write-0 time slot if V = 0. If the read time slots are 0 and 1, there follows a write-0 time slot. If the read time slots are 1 and 0, there follows a write-1 time slot. If the read time slots are both 1 (error case), the subsequent write time slot is a write 1.
Typical Use	To perform a 1-Wire Search ROM sequence; a full sequence requires this command to be executed 64 times to identify and address one device.
Restriction	1-Wire activity must have ended before the DS2482 can process this command.
Error Response	Command code and direction byte is not acknowledged if 1WB = 1 at the time the command code is received and the command will be ignored.
Command Duration	$3 \times t_{SLOT}$ + maximum 262.5ns, counted from the falling SCL edge of the first bit (MS bit) of the direction byte.
1-Wire Activity	Begins maximum 262.5ns after the falling SCL edge of the MS bit of the direction byte.
Read Pointer Position	Status register (for busy polling)
Status Bits Affected	1WB (set to 1 for $3 \times t_{SLOT}$) SBR is updated at the first t_{MSR} TSB and DIR are updated at the second t_{MSR} (i. e., at $t_{SLOT} + t_{MSR}$)
Configuration Bits Affected	1WS, APU apply

9.6.3. UTILISATION DU DS2482 EN RELATION AVEC DS18B20

Pour arriver à communiquer avec le DS18B20, il faut déterminer le principe de communication et utiliser les commandes correspondant aux besoins.

9.6.3.1. PRINCIPE DE LA COMMUNICATION AVEC LE DS18B20

Voici un extrait du document "Le bus 1-wire" écrit par D. Menesplier de l'ENAC :

Les DS18B20 possèdent un code unique sur 64 bits comme tous les circuits 1-Wire. Le code famille est h"28", suivi de 6 octets propres au circuit et d'un octet de CRC.

La détection de présence de ce circuit se fait en envoyant le pulse de Reset, qui est un état bas pendant au moins 480 µs. Quand un circuit DS 18B20 est présent sur le bus, il le signale en maintenant le bus à l'état bas pendant 60 à 240 µs.

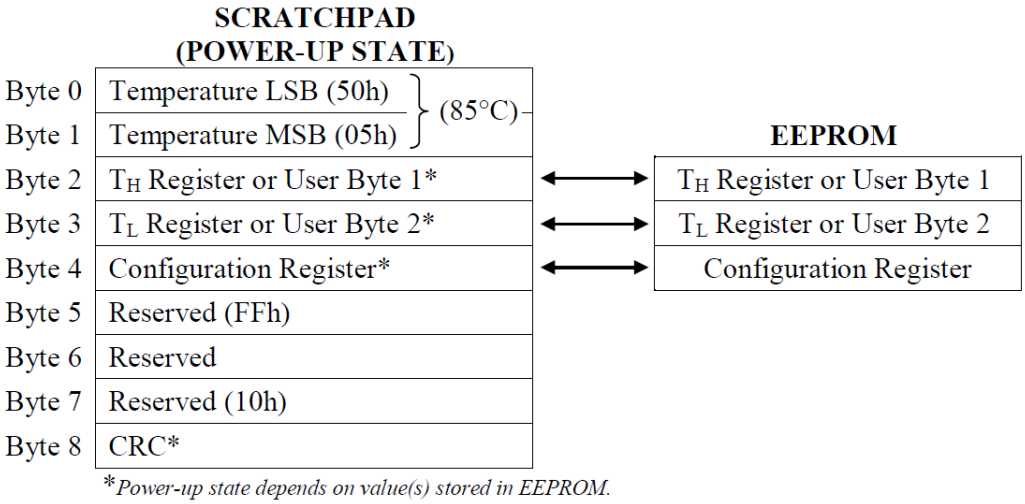
Toute transaction avec un tel circuit doit démarrer par un pulse de Reset suivie de l'envoi d'une commande ROM. On pourra après envoyer une commande de fonction propre à ce type de circuit.

Si le circuit est seul sur le bus 1-Wire, la commande ROM peut être l'appel général SKIP ROM = 0xCC. Si ce n'est pas le cas, il faudra connaître les 64 bits propres au circuit que l'on veut atteindre et utiliser la commande MATCH ROM = 0x55 suivie des 8 octets du code.

Une recherche préalable des 8 octets de code sera faite par la commande READ ROM = 0x33 si le circuit est seul ou bien ou bien par SEARCH ROM = 0xF0 s'il y a plusieurs circuits sur le bus.

9.6.3.1.1. Organisation de la mémoire interne

Elle est constituée d'une zone RAM de 9 octets et d'une zone EEPROM non volatile de 3 octets.



*Power-up state depends on value(s) stored in EEPROM.

9.6.3.2. COMMANDES DU DS18B20

Après avoir envoyé une commande ROM pour adresser un DS18B20 esclave, le maître doit envoyer un code de commande. Voici quelque commandes qui devraient suffire pour lire la température.

9.6.3.3. DS18B20, CONVERT

Code 0x44. Cette commande lance la conversion de température. Le résultat est rangé dans les 2 octets LSB et MSB . Le temps de conversion dépend de la résolution choisie. Le maître doit interroger le DS18B20 qui répond par un bit à "0" tant que la conversion n'est pas terminée. Quand l'opération est terminée, l'esclave répond par un bit à "1".

9.6.3.4. DS18B20, READ SCRATCHPAD

Code 0xBE. Les 9 octets de la RAM sont envoyés vers le maître. L'esclave commence par le bit 0 du premier octet et transmet ainsi les 9 octets de sa RAM.
Le maître peut interrompre à tout moment la lecture en faisant un Reset.

9.6.3.5. DS18B20, RÉSUMÉ DES COMMANDES

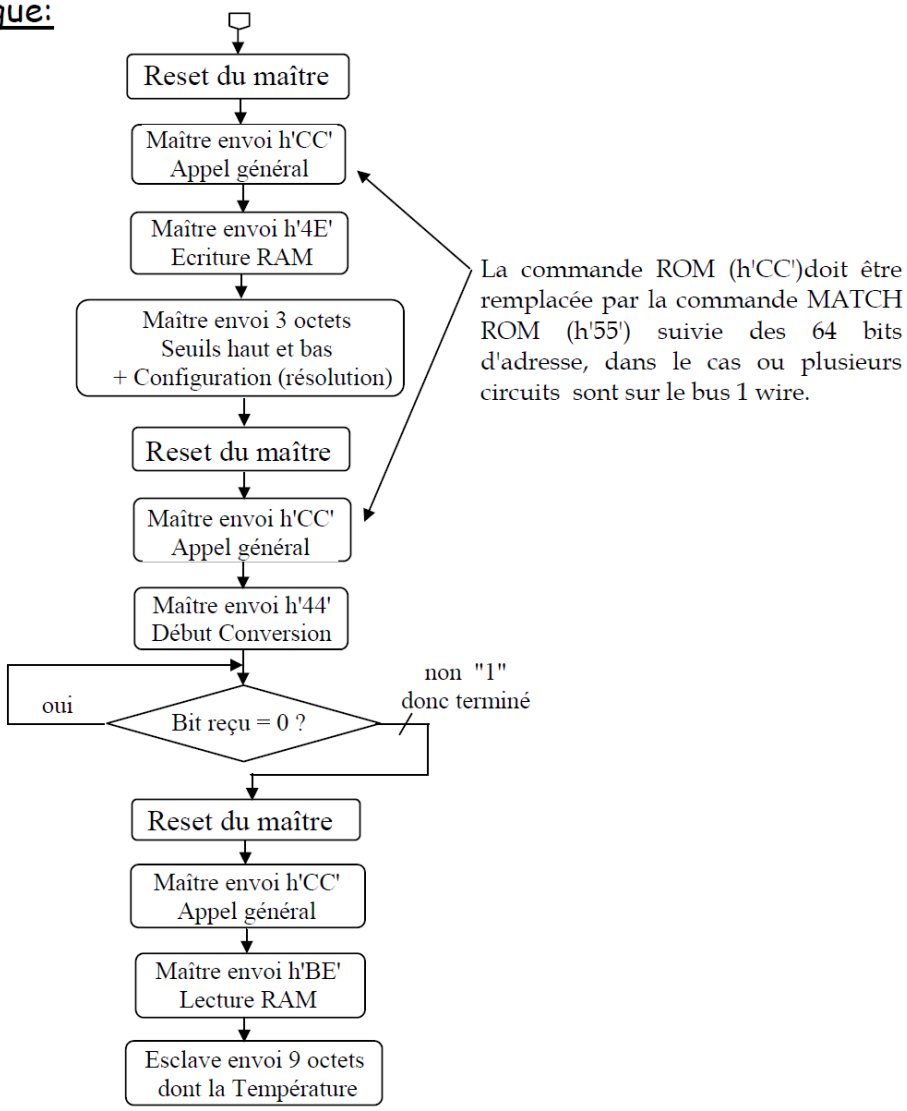
Voici le résumé des commandes tiré directement du datasheet du DS18B20 :

COMMAND	DESCRIPTION	PROTOCOL	1-Wire BUS ACTIVITYAFTER COMMAND IS ISSUED	NOTES
TEMPERATURE CONVERSION COMMANDS				
Convert T	Initiates temperature conversion.	44h	DS18B20 transmits conversion status to master (not applicable for parasite-powered DS18B20s).	1
MEMORY COMMANDS				
Read Scratchpad	Reads the entire scratchpad including the CRC byte.	BEh	DS18B20 transmits up to 9 data bytes to master.	2
Write Scratchpad	Writes data into scratchpad bytes 2, 3, and 4 (T_H, T_L, and configuration registers).	4Eh	Master transmits 3 data bytes to DS18B20.	3
Copy Scratchpad	Copies T_H, T_L, and configuration register data from the scratchpad to EEPROM.	48h	None	1
Recall E^2	Recalls T_H, T_L, and configuration register data from EEPROM to the scratchpad.	B8h	DS18B20 transmits recall status to master.	
Read Power Supply	Signals DS18B20 power supply mode to the master.	B4h	DS18B20 transmits supply status to master.	

Note 1: For parasite-powered DS18B20s, the master must enable a strong pullup on the 1-Wire bus during temperature conversions and copies from the scratchpad to EEPROM. No other bus activity may take place during this time.
Note 2: The master can interrupt the transmission of data at any time by issuing a reset.
Note 3: All three bytes must be written before a reset is issued.

9.6.3.6. EXEMPLE DE DIALOGUE

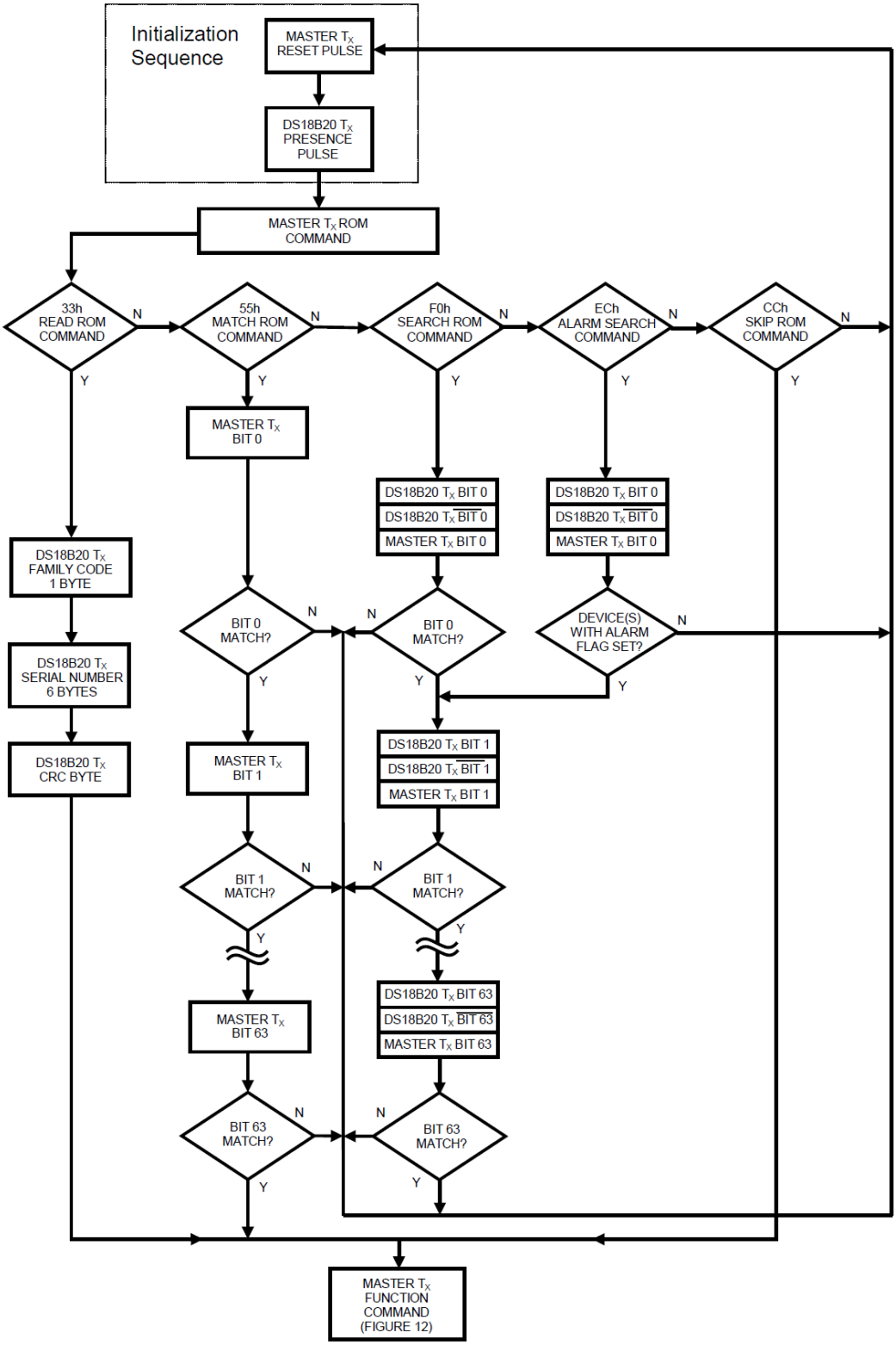
Exemple de dialogue:



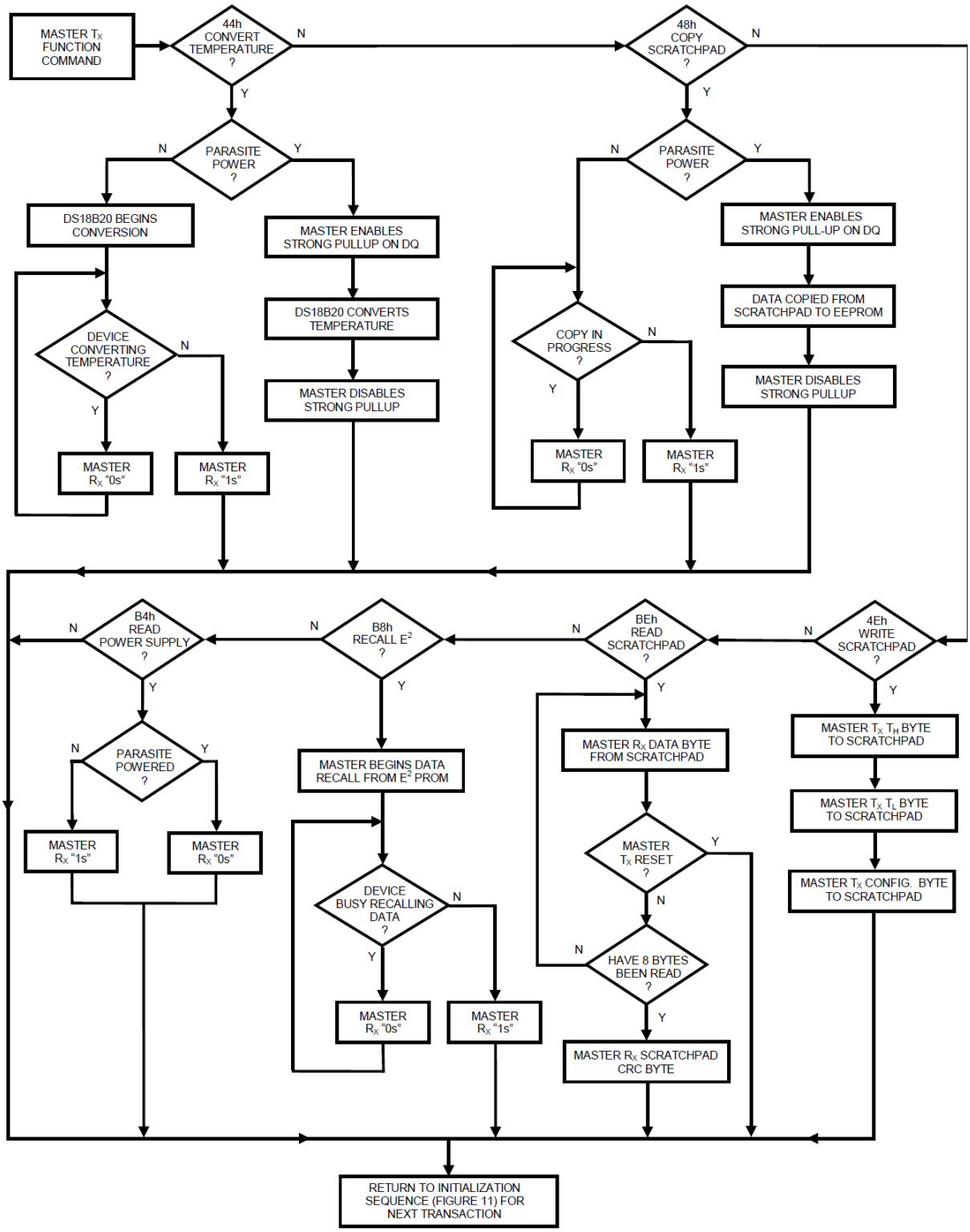
Une fois la configuration effectuée, il suffit de répéter l'envoi de la demande de conversion et la lecture de la RAM.

9.6.3.7. SÉQUENCE COMMANDE ROM

Flowchart issu du datasheet du DS18B20 :



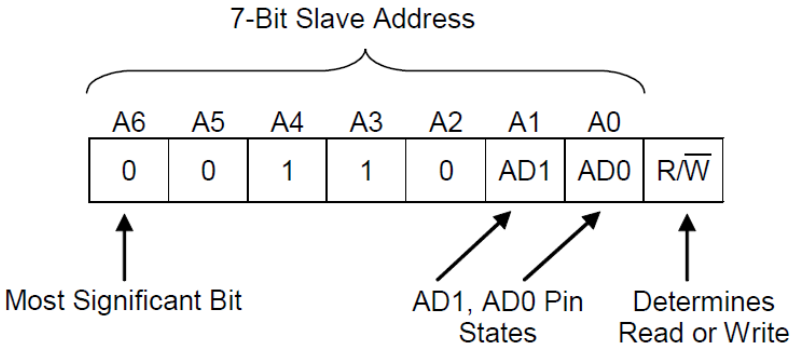
9.6.3.8. SÉQUENCE TRAITEMENT DES COMMANDES



Pour exécuter une 2ème commande, il est nécessaire de recommencer entièrement la séquence.

9.6.4. COMMUNICATION AVEC LE COMPOSANT DS18B20

9.6.4.1. DÉFINITIONS POUR LE DS2482




```
#define ds2482_100_write  0x30  // addr ds2482-100 R/_W = 0
#define ds2482_100_read   0x31  // addr ds2482-100 R/_W = 1

// Reset
// code de reset du ds2482-100
#define ds2482_100_reset_code 0xf0
// envoi d'un reset/presence sur la ligne one wire
#define ds2482_100_one_wire_device_reset 0xb4

// Configuration
// code de commande pour écrire le byte de configuration
#define ds2482_100_write_config_code 0xd2

#define ds2482_100_config_byte_spu 0b10100101
  // 1ws = 1 wire speed = standard (low) = 0
  // SPU = strong pullup = 1
  // PPM = presence pulse masking = 0
  // APU = active pullup = 1
#define ds2482_100_config_byte_nospu 0b11100001
  // 1ws = 1 wire speed = standard (low) = 0
  // SPU = strong pullup = 0
  // PPM = presence pulse masking = 0
  // APU = active pullup = 1

// set read pointer
// code de commande pour positionner le pointeur de lecture
#define ds2482_100_set_read_pointer_code 0xe1
// positionne le pointeur de lecture sur le status register
#define ds2482_100_set_read_pointer_to_status_register 0xf0

// Positionne pointeur lecture sur le read data register
#define ds2482_100_set_read_pointer_to_read_data_register 0xe1
// positionne pointeur lecture sur le channel selection register
#define ds2482_100_set_read_pointer_to_channel_selection_register 0xd2
// positionne le pointeur de lecture sur le configuration register
#define ds2482_100_set_read_pointer_to_configuration_register 0xf0
```

```
// write byte to 1 wire
// code de commande pour écrire un byte sur le ds2482-100
#define ds2482_100_1wire_write_byte_code 0xa5

// read byte to 1 wire
// code de commande pour lire un byte sur le ds2482-100
// résultat de la lecture dans le read data register
#define ds2482_100_1wire_read_byte_code 0x96

// write (and read) single bit
// code de commande pour écrire un bit slot sur la ligne
// on passe en paramètre la valeur du bit dans le msb
// et on retrouve la valeur du bit (soit écrite soit lue
// par le ds2482) dans le bit SBR du byte de status
#define ds2482_100_1wire_single_bit_code 0x87
// code de commande pour réaliser la séquence complète
#define ds2482_100_1wire_triplet_code 0x78
// valeur pour un single bit à 0
#define ds2482_100_1wire_single_bit_0 0x00
// valeur pour un single bit à 1
#define ds2482_100_1wire_single_bit_1 0x80
```

9.6.4.2. DÉFINITIONS 1-WIRE ET DS18B20

```
// Définitions one wire pour les commandes
#define one_wire_read_rom_command_code 0x33
#define one_wire_match_rom_command_code 0x55
#define one_wire_search_rom_command_code 0xf0
#define one_wire_skip_rom_command_code 0xcc
#define one_wire_no_command_code 0xff

// Définitions pour les sondes de température ds18b20
#define ds18b20_family_code 0x28
#define ds18b20_convert_T_command_code 0x44
#define ds18b20_copy_scratchpad_command_code 0x48
#define ds18b20_write_scratchpad_command_code 0x4e
#define ds18b20_read_scratchpad_command_code 0xbe
#define ds18b20_th_value 0x0
#define ds18b20_tl_value 0x0
// 12 bits résolution
#define ds18b20_config_byte 0b01111111
#define ds18b20_ok_status 0 // tout est ok
#define ds18b20_shorted 1 // ligne one wire court-circuitée
#define ds18b20_missing 2 // pas de capteur
// erreur de crc lors de la lecture de la température
#define ds18b20_temp_reading_crc_error 3
// condition de reset 85.0 °C
#define ds18b20_reset_condition 4
#define ds18b20_ident_code_reading_crc_error 5
// erreur de crc lors de la lecture du code d'identification

// définition de la température par défaut au power-up
// cette valeur n'est transmise qu'avant le premier accès à un canal
de sonde
#define power_up_temp_value (200) // 20°C
```

9.6.4.3. STRUCTURES ET VARIABLES

```
// pour ds2482-100
byte ds2482_100_status;

// Premièrement une structure pour le scratchpad du DS18b20
typedef struct
       {
         byte temp_lsb,temp_msb,r2,r3,r4,r5,r6,r7,crc;
       } t_ds18b20_scratchpad;

// Quatrièmement une union pour pouvoir accéder à 16bits
// soit en 2 bytes, soit en 16bits, soit en 16 bits signés
typedef union {
              struct
                 {
                   byte lsb,msb;
                 } octet;
                 uint16_t word;
                 int16_t signed_word;
              } t_16bits;

t_16bits new_temp;

// Structure pour la gestion d'un capteur
typedef struct
{
     byte Ds18b20Ident[8];
     t_ds18b20_scratchpad ds18b20_scratchpad;
     byte Ds18b20_status;
     t_16bits Temp16;
     t_16bits Last_temp16;
} t_sensor;
//
// Nous allons avoir besoin d'une de ces structure
t_sensor sensor;
```

9.6.4.4. LES FONCTIONS DE COMMUNICATIONS 1-WIRE

9.6.4.4.1. Write one byte

```
void ds2482_100_write_one_byte(uint8_t write_and_dest_chip,
                               uint8_t byte0 )
{
   i2c_start();
   i2c_write(write_and_dest_chip);
   i2c_write(byte0);
   i2c_stop();
}
```

9.6.4.4.2. Write two bytes

```
void ds2482_100_write_two_bytes(
                            uint8_t write_and_dest_chip,
                            uint8_t byte1, uint8_t byte2) {
   i2c_start();
   i2c_write(write_and_dest_chip);
   i2c_write(byte1);
   i2c_write(byte2);
   i2c_stop();
}
```

9.6.4.4.3. Write three bytes

```
void ds2482_100_write_three_bytes
                 (uint8_t write_and_dest_chip,
              uint8_t byte1, uint8_t byte2, uint8_t byte3)
{
   i2c_start();
   i2c_write(write_and_dest_chip);
   i2c_write(byte1);
   i2c_write(byte2);
   i2c_write(byte3);
   i2c_stop();
}
```

9.6.4.4.4. Read one byte

```
byte ds2482_100_read_one_wire_byte
                          (uint8_t read_and_source_chip){
   byte ret_byte;
   // read byte
   i2c_start();
   i2c_write(read_and_source_chip);
   ret_byte = i2c_read(0); // no ack
   i2c_stop();
   return ret_byte;
}
```

9.6.4.4.5. Read status

```
void one_wire_read_status(uint8_t read_and_source_chip) {
  // lecture du status de ds2482-800 ?
  i2c_start();
  i2c_write(read_and_source_chip);
  ds2482_100_status = i2c_read(0); // no ack
  i2c_stop();
}
```

9.6.4.4.6. Channel Busy

```
bool one_wire_channel_busy(uint8_t read_and_source_chip)
{
   // Transmission en cours sur la ligne one wire ?
   bool fin;

   one_wire_read_status(read_and_source_chip);
   fin =  ((ds2482_100_status & 0x01) > 0);
   // retourne un 1 si une transmission est en cours
   return fin;
}
```

9.6.4.4.7. End Xmit

Attend que la ligne 1-Wire se libère. Cette fonction est à utiliser pour attendre la fin d'exécution d'une commande 1-Wire.

```
void one_wire_end_xmit(uint8_t read_and_source_chip)
{
   do {
   } while (one_wire_channel_busy(read_and_source_chip));
}
```

9.6.4.5. LA FONCTION DE LECTURE DE LA TEMPÉRATURE

Cette fonction fournit la température et une indication sur la situation du composant OneWire.

☹ Il est nécessaire d'utiliser un passage par référence car lorsque l'on retourne une valeur float dans une variable globale, on obtient une valeur fantaisiste.

```
void ReadDS18b20(int8_t *Status, float *pTemp)
{
   float Temp = 21.5;
   uint8_t nb_bytes;

   // Reset du DS2482
   ds2482_100_write_one_byte(ds2482_100_write,
                                    ds2482_100_reset_code);
   one_wire_end_xmit(ds2482_100_read_address);

   // One-Wire reset/presence pulse
   ds2482_100_write_one_byte(ds2482_100_write,
                        ds2482_100_one_wire_device_reset);
   one_wire_end_xmit(ds2482_100_read_address);

   // Lecture et analyse du Status One-Wire
   one_wire_read_status(ds2482_100_read);
   switch (ds2482_100_status & 0x06) { // keep SD & PPD
       case 0:  // no sensor
           sensor.ds18b20_status = ds18b20_missing;
           // no sensor on line (ds18b20)
       break;
       case 2:  // normal operation,
                // not shorted and presence pulse ok
           sensor.ds18b20_status = ds18b20_ok_status;
           break;
       case 4:
       case 6:  // line shorted
            sensor.ds18b20_status = ds18b20_shorted;
                       // short circuit (ds18b20) on line
       break;

    } // end switch DS2482_100_status

      *status = ds2482_100_status & 0x06;

   if (*status != 2) {

      Goto ExitReadDs18b20;
   }

   // Envoi commande 0xCC SKIP ROM
   ds2482_100_write_two_bytes(ds2482_100_write_address,
                        ds2482_100_1wire_write_byte_code,
                        one_wire_skip_rom_command_code);
   one_wire_end_xmit(ds2482_100_read_address);
```

```
// Force strong Pullup
// (Nécessaire si Vcc non utilisé)
ds2482_100_write_two_bytes(ds2482_100_write_address,
                           ds2482_100_write_config_code,
                           ds2482_100_config_byte_spu);

// Envoi commande 0X44 début conversion T
ds2482_100_write_two_bytes(ds2482_100_write_address,
                      ds2482_100_1wire_write_byte_code,
                      ds18b20_convert_T_command_code);
one_wire_end_xmit(ds2482_100_read_address);

// Attente bit reçu = 0
// L'attente n'est possible que si circuit alimenté
// BSP_LEDOn(BSP_LED_7); // provisoire pour observation
delay_ms(750);  // par mesure 720 ms
// BSP_LEDOff(BSP_LED_7); // provisoire pour observation

// Supprime strong Pullup
ds2482_100_write_two_bytes(ds2482_100_write_address,
                         ds2482_100_write_config_code,
                         ds2482_100_config_byte_nospu);

// 2ème one wire reset/presence pulse
// ----------------------------------
// Remarque il n'est pas nécessaire de refaire
// le reset du DS2482-100
ds2482_100_write_one_byte(ds2482_100_write,
                      ds2482_100_one_wire_device_reset);
one_wire_end_xmit(ds2482_100_read_address);

// Envoi commande 0xCC SKIP ROM
ds2482_100_write_two_bytes(ds2482_100_write_address,
                      ds2482_100_1wire_write_byte_code,
                      one_wire_skip_rom_command_code);
one_wire_end_xmit(ds2482_100_read_address);

// Envoi commande 0xBE read scratchpad
// Lecture de 8 bytes de données et un crc, soit 9 bytes
ds2482_100_write_two_bytes(ds2482_100_write,
                     ds2482_100_1wire_write_byte_code,
                   ds18b20_read_scratchpad_command_code);
one_wire_end_xmit(ds2482_100_read_address);
ds18b20_scratchpad_ptr =
                    &sensor.ds18b20_scratchpad.temp_lsb;
```

```
   // principe séquencement des actions de lecture
        // 1) demander la lecture d'un byte au ds18b20
        // 2) lire le byte

   for (nb_bytes = 0 ; nb_bytes < 9 ; nb_bytes ++) {

      // Envoi commande de lecture d'un byte
      ds2482_100_write_one_byte(ds2482_100_write,
                    ds2482_100_1wire_read_byte_code);
      one_wire_end_xmit(ds2482_100_read_address);

      // Effectue la lecture d'un byte
         // Pointeur DS2482 sur Read data register
         // Lecture dans le DS2482
      ds2482_100_write_two_bytes(ds2482_100_write,
                      ds2482_100_set_read_pointer_code,
      ds2482_100_set_read_pointer_to_read_data_register);

      *ds18b20_scratchpad_ptr =
    ds2482_100_read_one_wire_byte(ds2482_100_read_address);
      // Remettre pointeur sur le Status
      // pour attendre fin action
      ds2482_100_write_two_bytes(ds2482_100_write,
                        ds2482_100_set_read_pointer_code,
           ds2482_100_set_read_pointer_to_status_register);
      one_wire_end_xmit(ds2482_100_read_address);
      // Pointe sur le byte suivant
      ds18b20_scratchpad_ptr++;
   } // end for
   // Expression de la température en degré
   new_temp.octet.msb = sensor.ds18b20_scratchpad.temp_msb;
   new_temp.octet.lsb = sensor.ds18b20_scratchpad.temp_lsb;
   temp = new_temp.signed_word * 0.0625;

ExitReadDs18b20:
   *pTemp = temp;
}
```

On constate que la séquence déjà relativement compliquée au niveau du DS18B20, est rendue plus compliquée par les actions nécessaires sur le DS2482-100. La lecture se complique par le besoin de changer le pointeur de lecture entre donnée et statut.

La séquence présentée est le scénario le plus simple. Dès que l'on veut vérifier la présence d'un composant avec identification, cela devient plus compliqué.

Le délai de 750 ms pose problème, il est nécessaire d'espacer les appels de la fonction sans descendre en dessous d'une seconde.

Dans une situation où il y a plusieurs composants OneWire sur la ligne, le traitement devient assez conséquent.

9.6.4.6. OBSERVATION DE LA TRANSACTION

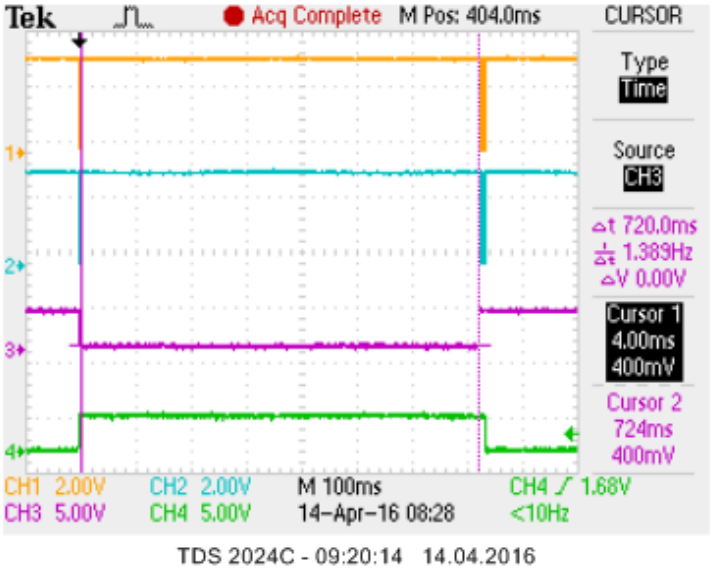
L'action sur la LED_6 marque le début et la fin de la fonction **ReadDs18b20.**

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

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

Canal 3 : LED_7
marqueur durée du délai dans la fonction ReadDS18B20

Canal 4 : LED_6
marqueur durée de la fonction ReadDS18B20.



On remarque de l'activité sur le bus I2C au début et à la fin du délai prévu de 750 ms qui fait en réalité 720 ms.

9.6.4.7. DÉTAILS DU DÉBUT LA TRANSACTION

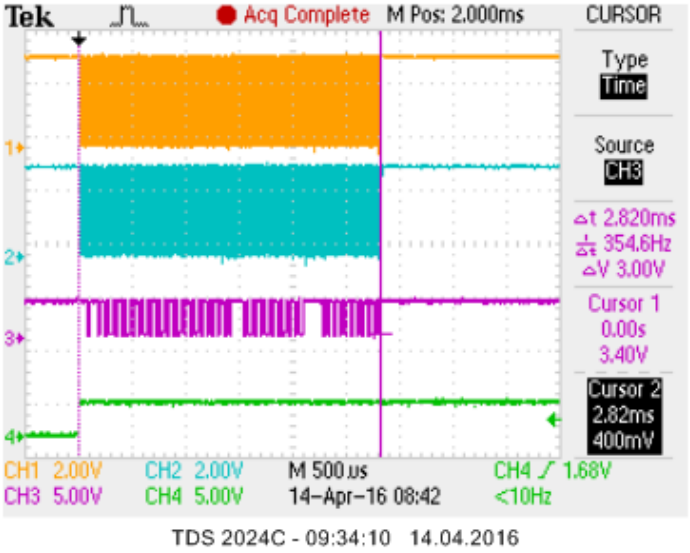
L'action sur la LED_6 marque le début et la fin de la fonction ReadDS18B20**.** La LED_5 est activée dans la fonction **i2c_read.**

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

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

Canal 3 : LED_5
marqueur dans la fonction i2c_read

Canal 4 : LED_6
marqueur durée de la fonction ReadDS18B20.



La durée du début de la transaction est de 2.8 ms.

9.6.4.8. DÉTAILS DE LA FIN DE LA TRANSACTION

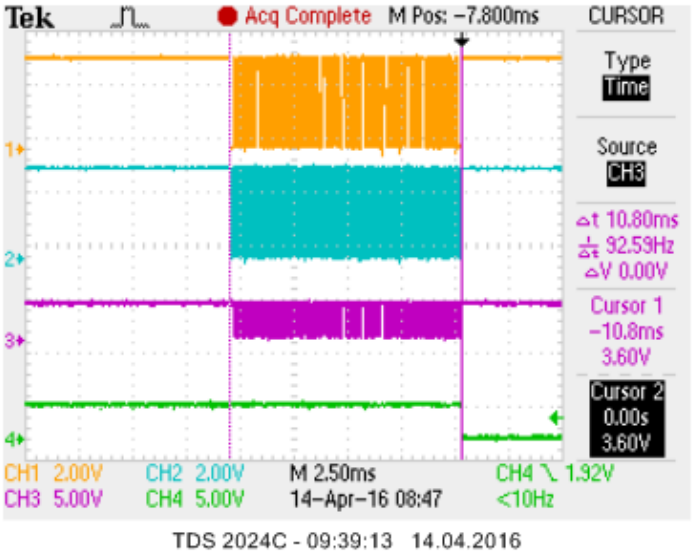
L'action sur la LED_6 marque le début et la fin de la fonction ReadDS18B20**.** La LED_5 est activée dans la fonction **i2c_read.**

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

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

Canal 3 : LED_5
marqueur dans la fonction i2c_read

Canal 4 : LED_6
marqueur autour de la fonction ReadDS18B20.



La durée de la fin de la transaction est de 10.8 ms.

9.6.4.9. CONTENU DU FICHIER APP.C DE L'APPLICATION UTILISÉE

Remarque : L'application est activée toutes les 1500 ms.

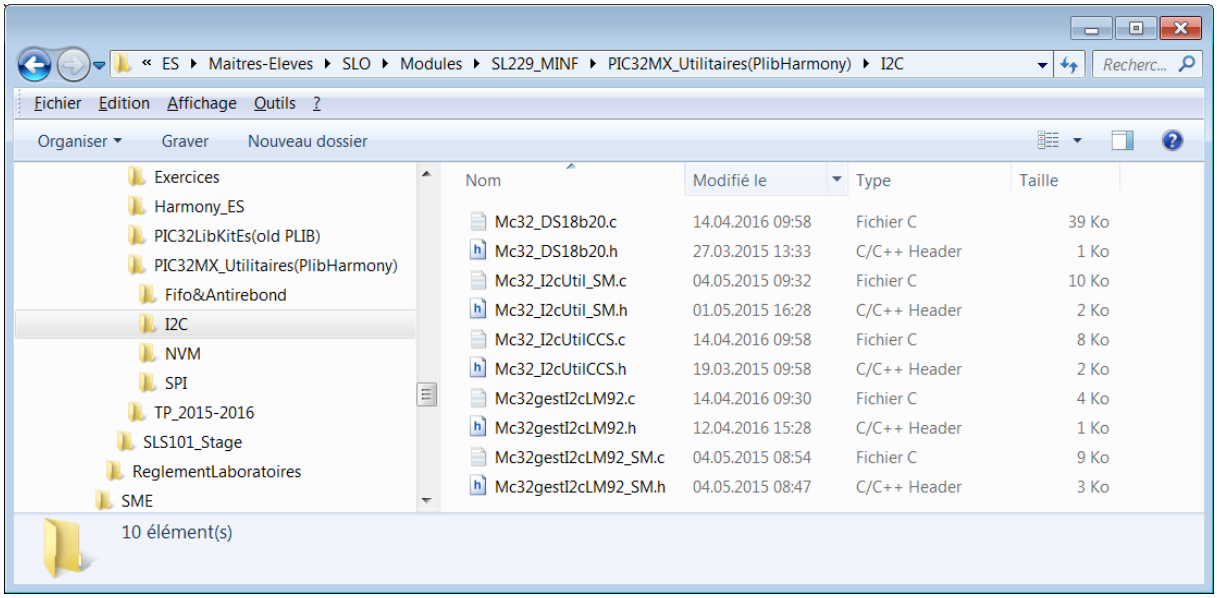
```
#include "app.h"
#include "Mc32DriverLcd.h"
#include "Mc32gestI2cLM92.h"
#include "Mc32_DS18b20.h"
#include "Mc32Delays.h"

// Variables
APP_DATA appData;
int16_t APP_RawTemp = 225;
float APP_TempLm92 = 21.7;
float APP_DS18B20Temp = 17.1;
uint8_t OneWireStatus = 2;
```

```
void APP_Tasks ( void )
{

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

            // Init
            I2C_InitLM92();
            init_oneWire();
            printf_lcd("TEST I2C Chap 9    ");
            lcd_gotoxy(1,2);
            printf_lcd("C. Huber 12.04.2016");
            DRV_TMR0_Start(); // depuis Harmony 1.06
            appData.state = APP_STATE_WAIT;
            break;
        }

        case APP_STATE_WAIT :
          // nothing to do
        break;

       case APP_STATE_SERVICE_TASKS:
            BSP_LEDToggle(BSP_LED_2);

            // APP_RawTemp = I2C_ReadRawTempLM92();
            APP_RawTemp = I2C_WriteCfgReadRawTempLM92();
            LM92_ConvRawToDeg( APP_RawTemp, &APP_TempLm92);

            // Affichage temperature du LM92
            lcd_gotoxy(1,3);
            printf_lcd("Temp = %6.1f", APP_TempLm92);
            delay_us(200) ;  //pour séparer les actions
            BSP_LEDOff(BSP_LED_6);
            ReadDS18B20(&OneWireStatus, &APP_DS18B20Temp);
            BSP_LEDOn(BSP_LED_6);
```

```
                lcd_gotoxy( 1, 4);
                switch (OneWireStatus) {
                    case 0 :
                       printf_lcd("Missing DS18B20");
                    break;
                    case 2 :
                       // Ok affiche la température
                       printf_lcd( "DS18B20 temp: %5.2f",
                                        APP_DS18B20Temp);
                    break;
                    case 4 :
                    case 6 :
                           printf_lcd("Line shorted   ");
                    break;
                } // end switch
                appData.state = APP_STATE_WAIT;
             break;
            /* TODO: implement your application state
               machine.*/

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

9.7. FICHIERS A DISPOSITION

Les 6 fichiers sont sous :
...\Maitres-Eleves\SLO\Modules\SL229_MINF\PIC32MX_Utilitaires(PlibHarmony)\I2C



Remarque : fichiers fournis avec les actions sur les LEDs d'observation mises en commentaire. Les fichiers xxx_SM seront traités dans le cadre des travaux pratiques.

9.8. CONCLUSION

On constate avec satisfaction que les fonctions I2C développées permettent de gérer simplement des composants I2C. Les librairies présentées d'accès aux capteurs de température LM92 ainsi qu'au DS18B20 (via une conversion I2C-OneWire) sont fonctionnelles.

Ce document devrait permettre, en s'inspirant des principes utilisés dans les deux exemples, de s'adapter à la gestion par le bus I2C de différents composants.

Il s'agira à chaque fois d'étudier avec soin la séquence à générer, en tenant compte de l'adresse de base du composant ainsi que de ses particularités. Une étude détaillée des datasheets est indispensable.

L'intégration d'un exemple de gestion d'un capteur OneWire fournit une base pour la gestion de ce type de composant par le biais du DS2482-100.

9.9. HISTORIQUE DES VERSIONS

9.9.1. VERSION 1.5 AVRIL 2015

Version de base pour PIC32MX avec compilateur XC32 et Harmony. La version 1.5 indique l'utilisation de la plib_i2c de Harmony.

9.9.2. VERSION 1.7 AVRIL 2016

La version 1.7 indique l'utilisation de la plib_i2c de Harmony V1.06 combiné avec le MPLABX 3.10. Contrôle et adaptation. Toutes les mesures ont été refaites.

9.9.3. VERSION 1.7.1 AVRIL 2016

En page 4 correction de SDL en SCL.

9.9.4. VERSION 1.8 AVRIL 2017

Relecture générale par SCA. Remise en forme. Eclaircissement des problèmes du code d'initialisation généré par Harmony.

9.9.5. VERSION 1.9 FEVRIER 2018

Ajouts documents de référence. Corrections mineures.

9.9.6. VERSION 1.91 FEVRIER 2022

Enlevé références à CCS et PIC18.