

MINF
Programmation des PIC32MX

Chapitre 8

Gestion du bus SPI



Théorie PIC32MX

Christian HUBER (CHR)
Serge CASTOLDI (SCA)
Version 1.91 mars 2019

CONTENU DU CHAPITRE 8

8. Gestion du bus SPI avec le PIC32MX	8-1
8.1. Bus SPI	8-2
8.1.1. Les lignes du bus SPI	8-2
8.1.2. Connexions entre master et slaves	8-2
8.1.3. Connexions multi-master	8-3
8.1.4. Les modes master et slave	8-3
8.2. Réalisation du bus SPI avec le PIC32MX	8-4
8.2.1. Câblage du SPI sur le KITPIC32MX	8-4
8.2.2. Schéma-bloc du module SPI	8-5
8.2.3. Particularité du SPIxSR	8-5
8.2.4. Les possibilités du module SPI	8-6
8.2.5. Timing en mode master (8 bits)	8-7
8.2.6. Timing en mode Slave	8-8
8.2.6.1. Slave Select pin disabled	8-8
8.2.6.2. Slave Select pin enabled	8-9
8.3. Les fonctions SPI de la PLIB_SPI	8-10
8.3.1. Les fonctions de configuration	8-10
8.3.1.1. La fonction PLIB_SPI_BaudRateClockSelect	8-10
8.3.1.2. La fonction PLIB_SPI_BaudRateSet	8-10
8.3.1.3. La fonction PLIB_SPI_CommunicationWidthSelect	8-11
8.3.1.4. La fonction PLIB_SPI_ClockPolaritySelect	8-11
8.3.1.5. La fonction PLIB_SPI_InputSamplePhaseSelect	8-11
8.3.1.6. La fonction PLIB_SPI_OutputDataPhaseSelect	8-12
8.3.1.7. La fonction PLIB_SPI_PinEnable	8-12
8.3.1.8. La fonction PLIB_SPI_IsBusy	8-12
8.3.1.9. La fonction PLIB_SPI_FIFOInterruptModeSelect	8-13
8.3.2. Les fonctions du transmetteur	8-13
8.3.3. Les fonctions du récepteur	8-13
8.3.4. Les fonctions de "Data Transfer"	8-14
8.3.4.1. La fonction PLIB_SPI_BufferClear	8-14
8.3.4.2. La fonction PLIB_SPI_BufferRead	8-14
8.3.4.3. La fonction PLIB_SPI_BufferWrite	8-14
8.4. Etude du driver SPI fourni par MHC	8-15
8.4.1. Configuration du driver	8-15
8.4.2. La fonction DRV_SPI0_Initialize	8-15
8.4.3. La fonction DRV_SPI0_ReceiverBufferIsFull	8-16
8.4.4. La fonction DRV_SPI0_TransmitterBufferIsFull	8-16
8.4.5. La fonction DRV_SPI0_BufferAddWriteRead	8-16
8.5. Réalisation de l'utilitaire SPI	8-18
8.5.1. Réalisation de la fonction spi_write1	8-18
8.5.2. Réalisation de la fonction spi_read1	8-18
8.5.2.1. Séquence des actions de lecture	8-19
8.5.2.2. Observation du moment de lecture du tampon	8-20

8.7. Périphériques SPI du Kit PIC32MX	8-21
8.8. Communication avec le Dac LTC2604	8-22
8.8.1. Chip Select du LTC2604	8-22
8.8.2. Configuration SPI nécessaire au LTC2604	8-22
8.8.2.1. Fonction de configuration du SPI pour le DAC	8-23
8.8.3. Détail du SPIxCON	8-25
8.8.4. Sélection de la fréquence de SCK	8-27
8.8.5. Initialisation du LTC2604	8-27
8.8.6. Ecriture d'une valeur sur le LTC2604	8-28
8.8.7. Observation des signaux du DAC	8-29
8.8.7.1. Détail du 3ème octet	8-29
8.8.7.2. Signal sur le DAC	8-30
8.10. Communication avec le LM70	8-31
8.10.1. Connexions entre le PIC32 et le LM70	8-31
8.10.2. Configuration SPI nécessaire au LM70	8-31
8.10.2.1. Fonction de configuration du SPI pour le LM70	8-32
8.10.2.2. Vérification de la configuration avec SPI1CON	8-32
8.10.3. Initialisation du LM70	8-33
8.10.4. Lecture du LM70	8-34
8.10.5. La fonction SPI_ReadRawTempLM70	8-34
8.10.6. Ecriture vers le LM70	8-35
8.10.6.1. Exemple d'écriture	8-35
8.10.7. Observation des signaux du LM70	8-36
8.10.7.1. Vue de détail	8-37
8.11. Utilisation combinée des 2 slaves SPI	8-38
8.11.1. Fonction lecture LM70 avec reconfiguration	8-38
8.11.2. Fonction écriture LTC2604 avec reconfiguration	8-39
8.11.3. Utilisation et obtention des résultats	8-39
8.11.3.1. Contenu de la réponse à l'interruption du Timer1	8-39
8.11.3.2. Affichage température dans l'application	8-40
8.11.3.3. Problème avec les variables float dans l'interruption	8-40
8.11.4. Remarque sur les reconfigurations	8-41
8.12. Fichiers à disposition	8-41
8.13. Conclusion	8-41
8.14. Historique des versions	8-42
8.14.1. Version 1.0 mai 2014	8-42
8.14.2. Version 1.1 mai 2014	8-42
8.14.3. Version 1.5 mars 2015	8-42
8.14.4. Version 1.7 mai 2016	8-42
8.14.5. Version 1.8 mars 2017	8-42
8.14.6. Version 1.9 février 2018	8-42
8.14.7. Version 1.91 mars 2019	8-42

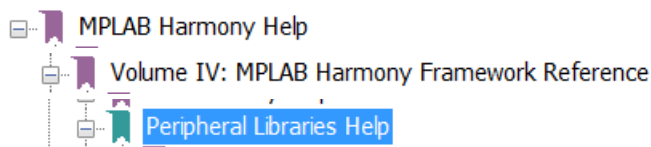
8. GESTION DU BUS SPI AVEC LE PIC32MX

Ce chapitre traite du bus SPI et de sa gestion avec le microcontrôleur PIC32MX795F512L ainsi que de la mise en pratique avec les composants LM70 et LTC2604.

Le PIC32MX dispose de modules de communication sérielle synchrone dédiés à la communication SPI (Serial Peripheral Interface).

Les documents de référence sont :

- La documentation "PIC32 Family Reference Manual" :
Section 23 : Serial Peripheral Interface
- Pour les détails spécifiques au PIC32MX795F512L, il faut se référer au document "PIC32MX5XX/6XX/7XX Family Data Sheet" :
Section 18 : Serial Peripheral Interface (SPI)
- La documentation d'Harmony, qui se trouve dans <Répertoire Harmony>\v<n>\doc :
Section MPLAB Harmony Framework Reference > Peripheral Libraries Help,
sous-section SPI Peripheral Library



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

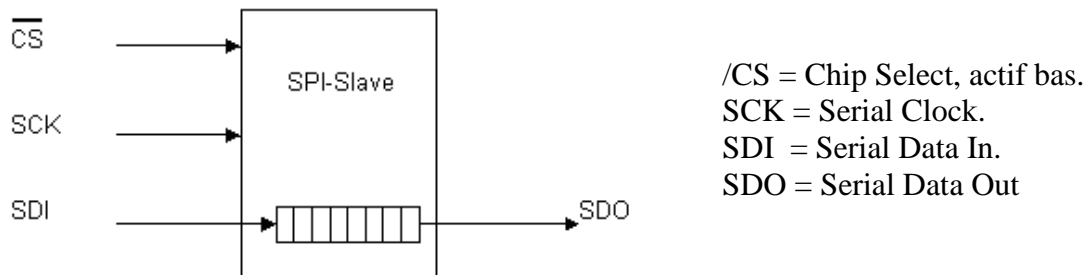
8.1. BUS SPI

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

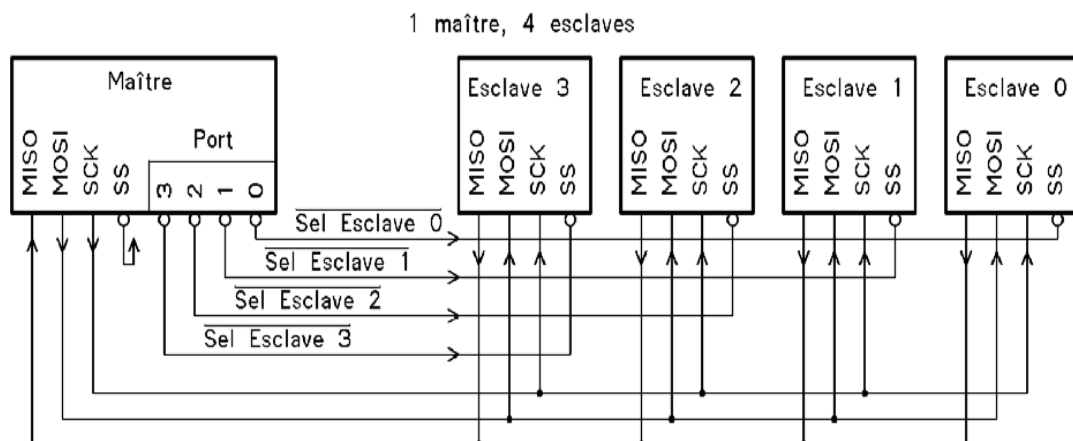
Nous allons présenter le rôle des lignes du bus, les modes et les signaux.

8.1.1. LES LIGNES DU BUS SPI

Voici les connexions standards sur un composant SPI, donc esclave.



8.1.2. CONNEXIONS ENTRE MASTER ET SLAVES

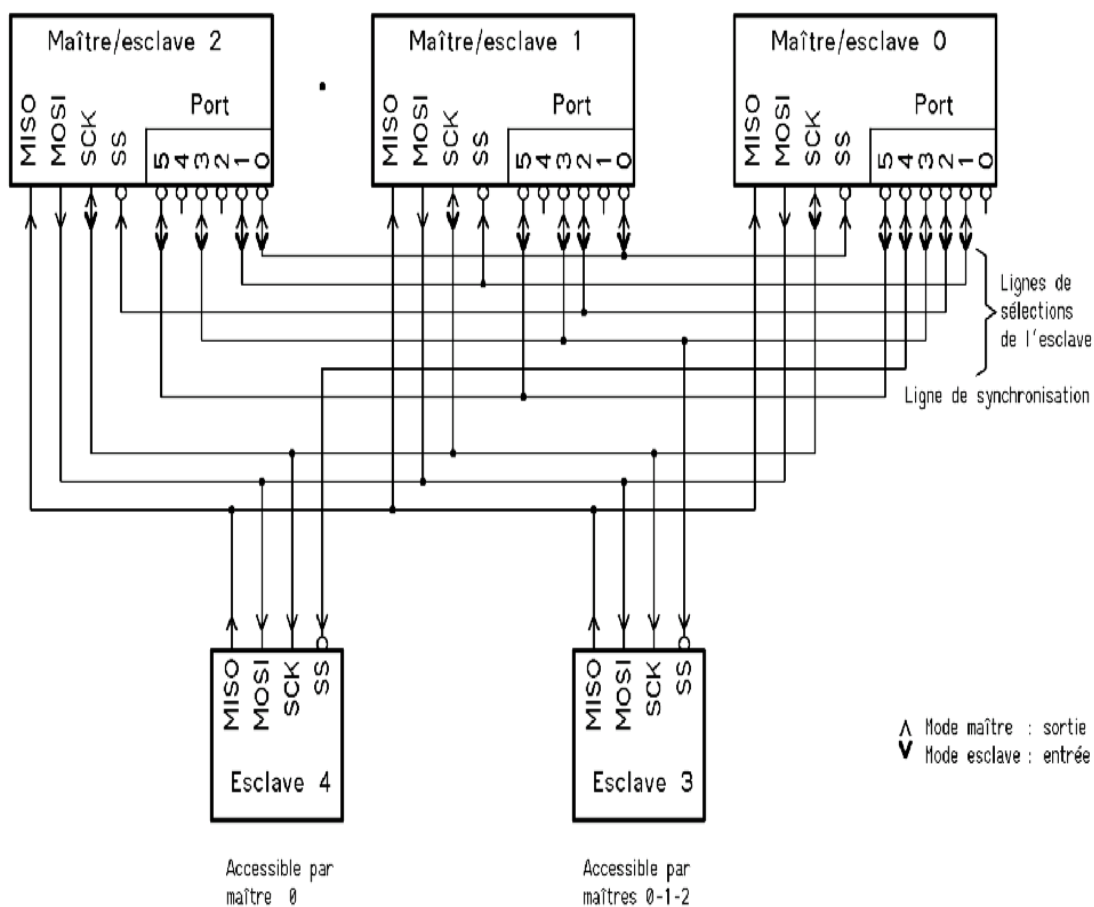


Motorola définit les lignes de la manière suivante :

- MISO = Master In, Slave Out.
- MOSI = Master Out, Slave In
- SCK = Serial Clock
- SS = Slave Select

8.1.3. CONNEXIONS MULTI-MASTER

Il ne doit y avoir qu'un seul maître à la fois, mais les rôles peuvent être permutés. Cela implique le passage pour le SCK de sortie à entrée lorsque le maître devient esclave.



8.1.4. LES MODES MASTER ET SLAVE

En mode maître, le composant (en général le microcontrôleur) doit fournir l'horloge et activer la sélection de l'esclave.

En mode esclave, le composant (en général un périphérique) reçoit l'horloge et réagit à l'entrée de sélection.

En général un microcontrôleur peut être configuré en maître ou en esclave, c'est le cas du PIC.

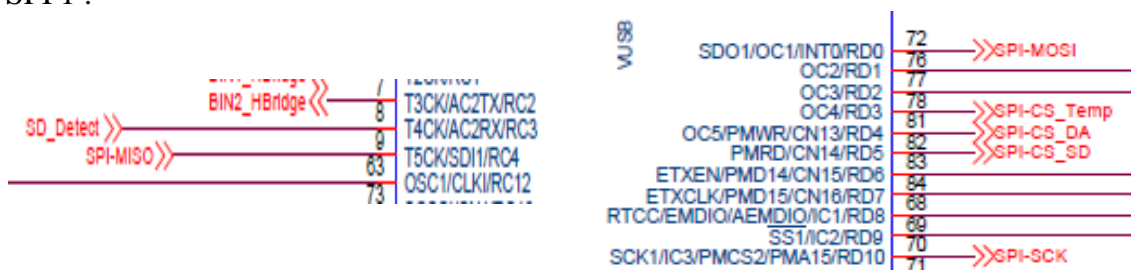
8.2. RÉALISATION DU BUS SPI AVEC LE PIC32MX

Le PIC32MX795F512L possède 4 modules SPI. Voici l'attribution des broches.

Pin Name	Pin Number ⁽¹⁾			Pin Type	Buffer Type	Description
	64-Pin QFN/TQFP	100-Pin TQFP	121-Pin XBGA			
SCK1	—	70	D11	I/O	ST	Synchronous serial clock input/output for SPI1
SDI1	—	9	E1	I	ST	SPI1 data in
SDO1	—	72	D9	O	—	SPI1 data out
SS1	—	69	E10	I/O	ST	SPI1 slave synchronization or frame pulse I/O
SCK3	49	48	K9	I/O	ST	Synchronous serial clock input/output for SPI3
SDI3	50	52	K11	I	ST	SPI3 data in
SDO3	51	53	J10	O	—	SPI3 data out
SS3	43	47	L9	I/O	ST	SPI3 slave synchronization or frame pulse I/O
SCK2	4	10	E3	I/O	ST	Synchronous serial clock input/output for SPI2
SDI2	5	11	F4	I	ST	SPI2 data in
SDO2	6	12	F2	O	—	SPI2 data out
SS2	8	14	F3	I/O	ST	SPI2 slave synchronization or frame pulse I/O
SCK4	29	39	L6	I/O	ST	Synchronous serial clock input/output for SPI4
SDI4	31	49	L10	I	ST	SPI4 data in
SDO4	32	50	L11	O	—	SPI4 data out
SS4	21	40	K6	I/O	ST	SPI4 slave synchronization or frame pulse I/O

8.2.1. CÂBLAGE DU SPI SUR LE KITPIC32MX

Comme on peut le voir dans les éléments de schéma ci-dessous, le kit utilise le module SPI 1 :

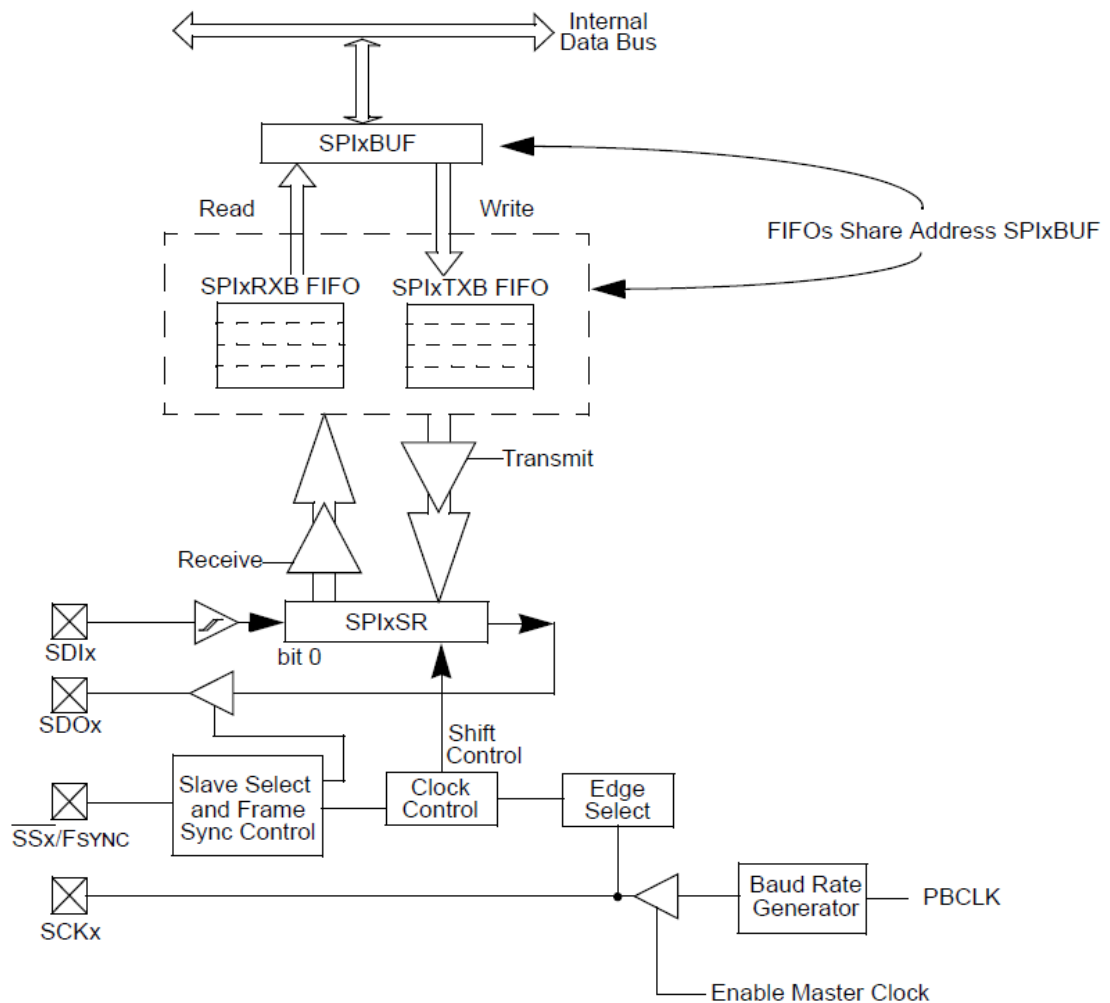


Broche du PIC32MX795F512L (100 pin TQFP)	Nom schéma	No broche boîtier 100
SCK1/RD10	SPI-SCK	70
SDO1/RD0	SPI-MOSI	72
SDI1/RC4	SPI-MISO	9
SS1/RD9	DAC_CLR	69

👉 Remarque : Le slave select 1 _SS1 est utilisé pour le DAC !

8.2.2. SCHÉMA-BLOC DU MODULE SPI

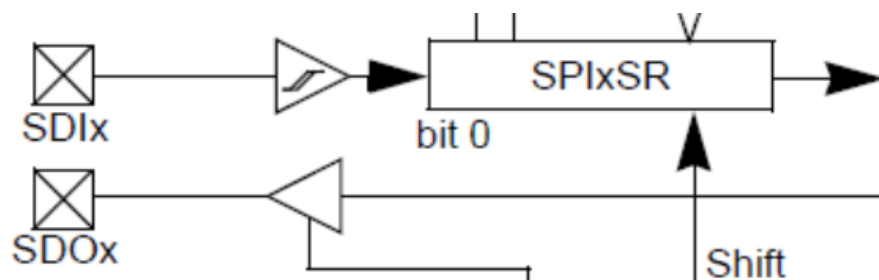
Voici le schéma bloc du module SPI :



Note: Access SPIxTXB and SPIxRXB FIFOs via SPIxBUF register.

8.2.3. PARTICULARITÉ DU SPIxSR

Comme le montre l'extrait ci-dessous, le registre à décalage **SPIxSR** va sortir un bit du registre pour chaque bit reçu. Cela signifie que lors de l'écriture, le slave fournit une info. Une transmission SPI est full-duplex.



Cette particularité est à gérer en relation avec les spécificités de chaque slave SPI.

8.2.4. LES POSSIBILITÉS DU MODULE SPI

Le tableau ci-dessous résume les possibilités du module SPI.

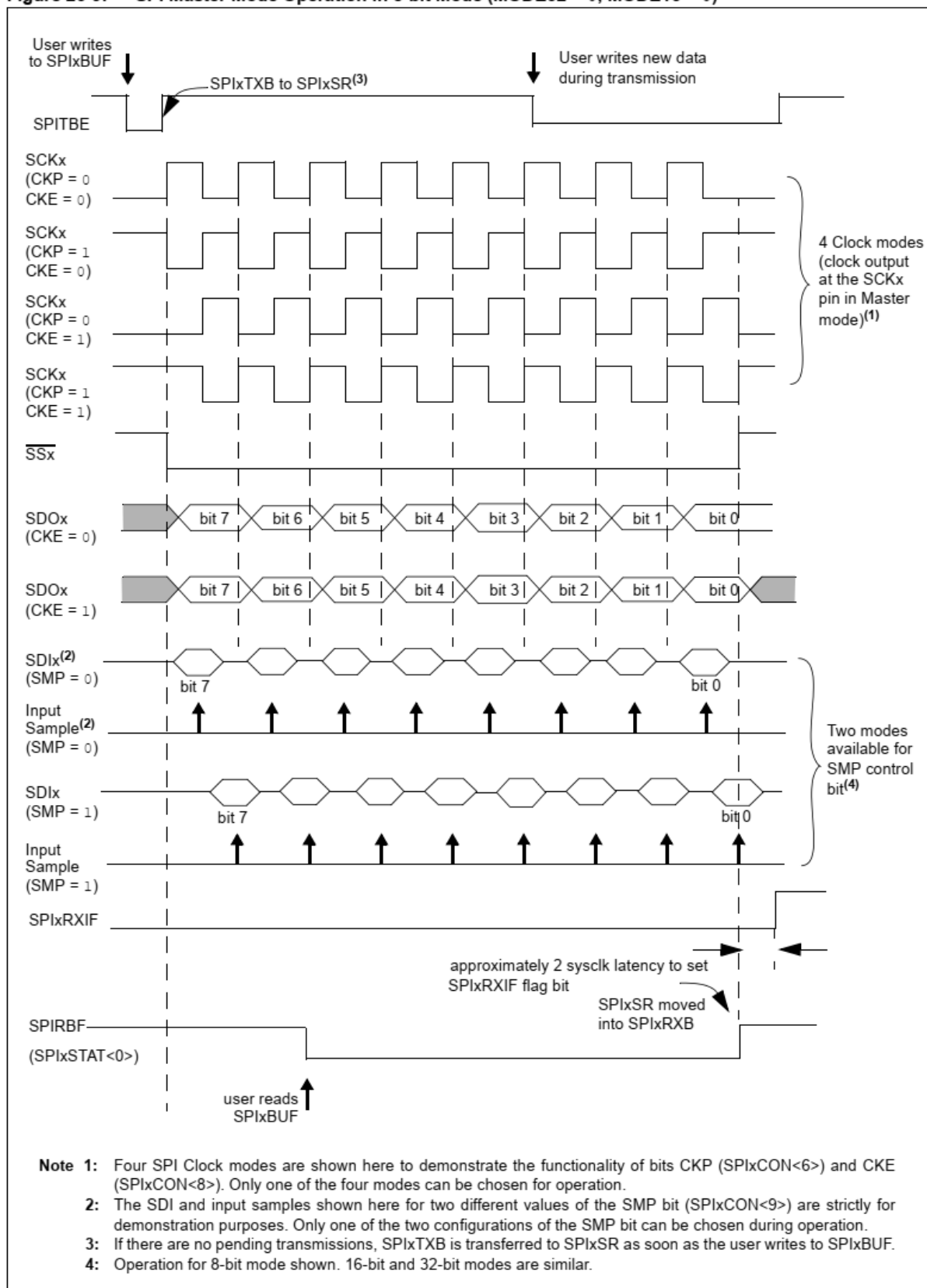
Table 23-1: SPI Features

Available SPI Modes	SPI Master	SPI Slave	Frame Master	Frame Slave	8-Bit, 16-Bit, and 32-Bit Modes	Selectable Clock Pulses and Edges	Selectable Frame Sync Pulses and Edges	Slave Select Pulse
Normal Mode	Yes	Yes	—	—	Yes	Yes	—	Yes
Framed Mode	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No

On en déduit la possibilité de travailler en mode 8, 16 ou 32 bits. Le mode "Framed" ne sera pas traité dans ce chapitre.

8.2.5. TIMING EN MODE MASTER (8 BITS)

Figure 23-9: SPI Master Mode Operation in 8-bit Mode (MODE32 = 0, MODE16 = 0)

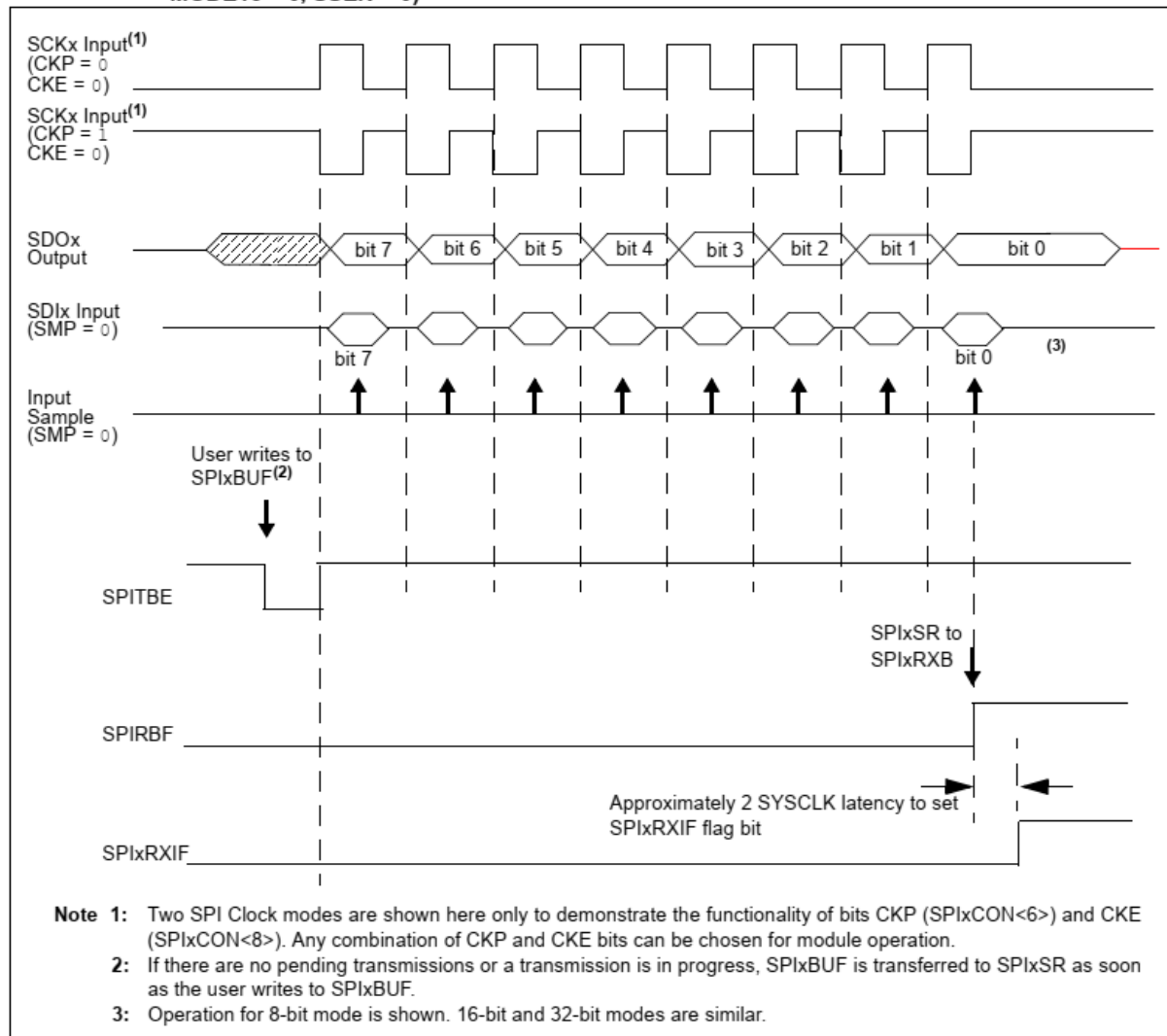


On peut observer les possibilités de configuration : les flancs de l'horloge en relation avec la donnée, ce qui permet d'adapter le master à un slave demandant un des modes. En mode 8 bits, le module SPI sérialise/dé-sérialise un octet à la fois.

8.2.6. TIMING EN MODE SLAVE

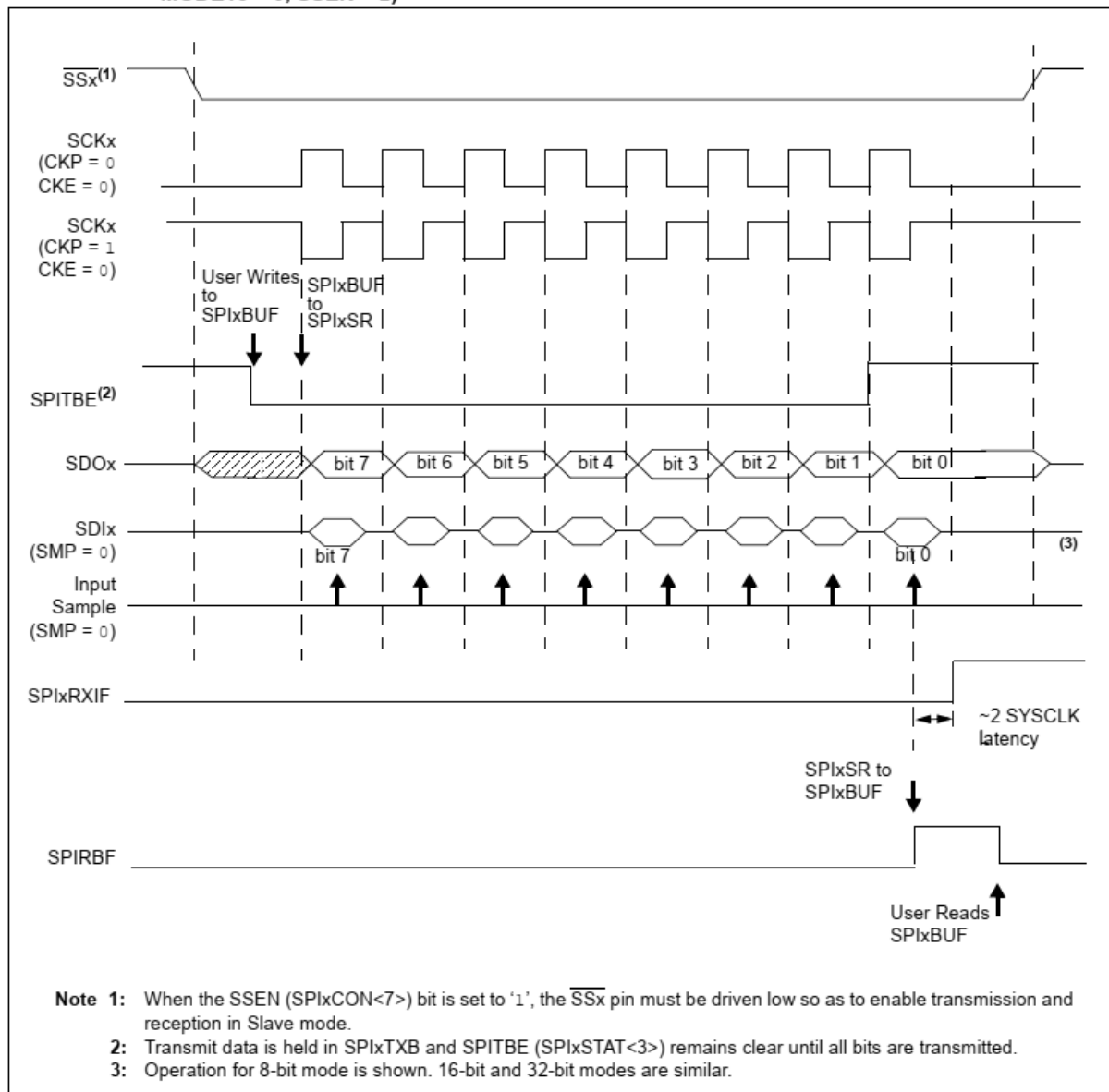
8.2.6.1. SLAVE SELECT PIN DISABLED

Figure 23-10: SPI Slave Mode Operation in 8-bit Mode with Slave Select Pin Disabled (MODE32 = 0, MODE16 = 0, SSEN = 0)



8.2.6.2. SLAVE SELECT PIN ENABLED

Figure 23-11: SPI Slave Mode Operation in 8-bit Mode with Slave Select Pin Enabled (MODE32 = 0, MODE16 = 0, SSEN = 1)



8.3. LES FONCTIONS SPI DE LA PLIB_SPI

Avec l'introduction de Harmony, les fonctions des bibliothèques périphériques ne font plus partie du compilateur. La PLIB_SPI est décrite au paragraphe SPI Peripheral Library, dans la documentation de Harmony (version 1.06 ci-dessous).

8.3.1. LES FONCTIONS DE CONFIGURATION

Nous disposons de nombreuses fonctions de configuration dont la description sera limitée à celles demandant des explications.

Name	Description
PLIB_SPI_BaudRateClockSelect	Selects the type of clock is used by the Baud Rate Generator.
PLIB_SPI_BaudRateSet	Sets the baud rate to the desired value.
PLIB_SPI_ClockPolaritySelect	Enables clock polarity.
PLIB_SPI_CommunicationWidthSelect	Selects the data width for the SPI communication.
PLIB_SPI_Disable	Disables the SPI module.
PLIB_SPI_Enable	Enables the SPI module.
PLIB_SPI_ErrorInterruptDisable	Enables SPI error interrupts.
PLIB_SPI_ErrorInterruptEnable	Enables SPI error interrupts
PLIB_SPI_FIFOCountGet	Reads the SPI Buffer Element Count bits for either receive or transmit.
PLIB_SPI_FIFODisable	Disables the SPI enhanced buffer.
PLIB_SPI_FIFOEnable	Enables the SPI enhanced buffer.
PLIB_SPI_FIFOInterruptModeSelect	Selects the SPI buffer interrupt mode.
PLIB_SPI_FIFOShiftRegisterIsEmpty	Returns the current status of the SPI shift register.
PLIB_SPI_InputSamplePhaseSelect	Selects the SPI data input sample phase.
PLIB_SPI_IsBusy	Returns the current SPI module activity status.
PLIB_SPI_MasterEnable	Enables the SPI in Master mode.
PLIB_SPI_OutputDataPhaseSelect	Selects serial output data change.
PLIB_SPI_PinDisable	Enables the selected SPI pins.
PLIB_SPI_PinEnable	Enables the selected SPI pins.
PLIB_SPI_PrescalePrimarySelect	Selects the primary prescale for SPI Master mode.
PLIB_SPI_PrescaleSecondarySelect	Selects the secondary prescale for SPI Master mode.
PLIB_SPI_ReadDataIsSignExtended	Returns the current status of the receive (RX) FIFO sign-extended data.
PLIB_SPI_SlaveEnable	Enables the SPI in Slave mode.
PLIB_SPI_SlaveSelectDisable	Disables Master mode slave select.
PLIB_SPI_SlaveSelectEnable	Enables Master mode slave select.
PLIB_SPI_StopInIdleDisable	Continues module operation when the device enters Idle mode.
PLIB_SPI_StopInIdleEnable	Discontinues module operation when the device enters Idle mode.

8.3.1.1. LA FONCTION `PLIB_SPI_BAUDRATECLOCKSELECT`

⊗ Cette fonction n'est pas supportée par le PIC32MX.

8.3.1.2. LA FONCTION `PLIB_SPI_BAUDRATESET`

La fonction `PLIB_SPI_BaudRateSet`, permet d'établir la fréquence du SCK. Voici son prototype:

```
void PLIB_SPI_BaudRateSet(SPI\_MODULE\_ID index,
                          uint32\_t clockFrequency,
                          uint32\_t baudRate)
```

Il faut fournir la fréquence de l'horloge (en principe PB_CLOCK) ainsi que la fréquence voulue. Il faut cependant veiller à ce que le rapport des fréquences soit un nombre entier pair (2, 4, 6, 8, ...).

Par exemple avec l'horloge à 80 MHz pour obtenir SCK à 20 MHz le rapport est de 4, on écrira :

```
PLIB_SPI_BaudRateSet(KitSpi1, SYS_CLK_FREQ, 20000000);
```

8.3.1.3. LA FONCTION PLIB_SPI_COMMUNICATIONWIDTHSELECT

La fonction `PLIB_SPI_CommunicationWidthSelect` permet d'établir si la communication est effectuée par paquets de 8, 16 ou 32 bits. Voici son prototype :

```
void PLIB_SPI_CommunicationWidthSelect(SPI_MODULE_ID index,
                                         SPI_COMMUNICATION_WIDTH width)
```

Le type énuméré `SPI_COMMUNICATION_WIDTH` permet le choix.

```
typedef enum {
    SPI_COMMUNICATION_WIDTH_8BITS = 0,
    SPI_COMMUNICATION_WIDTH_16BITS = 1,
    SPI_COMMUNICATION_WIDTH_32BITS = 2
} SPI_COMMUNICATION_WIDTH;
```

8.3.1.4. LA FONCTION PLIB_SPI_CLOCKPOLARITYSELECT

La fonction `PLIB_SPI_ClockPolaritySelect` permet de sélectionner la polarité de l'horloge.

```
void PLIB_SPI_ClockPolaritySelect(SPI_MODULE_ID index,
                                   SPI_CLOCK_POLARITY polarity)
```

Le type énuméré `SPI_CLOCK_POLARITY` permet le choix entre 2 polarités :

```
typedef enum {
    SPI_CLOCK_POLARITY_IDLE_LOW = 0,
    SPI_CLOCK_POLARITY_IDLE_HIGH = 1
} SPI_CLOCK_POLARITY;
```

Cette fonction agit sur le bit CKP du registre SPIxCON.

bit 6	CKP: Clock Polarity Select bit
	1 = Idle state for clock is a high level; active state is a low level
	0 = Idle state for clock is a low level; active state is a high level

8.3.1.5. LA FONCTION PLIB_SPI_INPUTSAMPLEPHASESELECT

La fonction `PLIB_SPI_InputSamplePhaseSelect` permet d'établir à quel moment les bits en entrée sont échantillonnés. Son prototype est le suivant :

```
void PLIB_SPI_InputSamplePhaseSelect(SPI_MODULE_ID index,
                                       SPI_INPUT_SAMPLING_PHASE phase)
```

Le type énuméré `SPI_INPUT_SAMPLING_PHASE` permet le choix entre 2 variantes :

```
typedef enum {
    SPI_INPUT_SAMPLING_PHASE_IN_MIDDLE = 0,
    SPI_INPUT_SAMPLING_PHASE_AT_END = 1
} SPI_INPUT_SAMPLING_PHASE;
```

Cette fonction agit sur le bit SMP du registre SPIxCON.

bit 9 **SMP**: SPI Data Input Sample Phase bit
 Master mode (MSTEN = 1):
 1 = Input data sampled at end of data output time
 0 = Input data sampled at middle of data output time
 Slave mode (MSTEN = 0):
 SMP value is ignored when SPI is used in Slave mode. The module always uses SMP = 0.

8.3.1.6. LA FONCTION **PLIB_SPI_OutputDataPhaseSelect**

La fonction **PLIB_SPI_OutputDataPhaseSelect** permet d'établir sur quelle transition de l'horloge les données sont émises. Son prototype est le suivant :

```
void PLIB_SPI_OutputDataPhaseSelect(SPI_MODULE_ID index,
                                     SPI_OUTPUT_DATA_PHASE phase)
```

Le type énuméré **SPI_OUTPUT_DATA_PHASE** permet d'indiquer le choix.

```
typedef enum {
    SPI_OUTPUT_DATA_PHASE_ON_IDLE_TO_ACTIVE_CLOCK = 0,
    SPI_OUTPUT_DATA_PHASE_ON_ACTIVE_TO_IDLE_CLOCK = 1
} SPI_OUTPUT_DATA_PHASE;
```

Cette fonction agit sur le bit CKE du registre SPIxCON.

bit 8 **CKE**: SPI Clock Edge Select bit
 1 = Serial output data changes on transition from active clock state to Idle clock state (see CKP bit)
 0 = Serial output data changes on transition from Idle clock state to active clock state (see CKP bit)

8.3.1.7. LA FONCTION **PLIB_SPI_PinEnable**

La fonction **PLIB_SPI_PinEnable** permet d'activer les lignes SPI mentionnées.

```
void PLIB_SPI_PinEnable(SPI_MODULE_ID index, SPI_PIN pin)
```

Le type énuméré **SPI_PIN** permet d'indiquer la ligne à activer.

```
typedef enum {
    SPI_PIN_DATA_OUT,
    SPI_PIN_DATA_IN,
    SPI_PIN_SLAVE_SELECT
} SPI_PIN;
```

8.3.1.8. LA FONCTION **PLIB_SPI_IsBusy**

La fonction **PLIB_SPI_IsBusy** n'est pas une fonction de configuration, elle permet de savoir si le system SPI est actif (sérialisation/désérialisation ou au repos).

```
bool PLIB_SPI_IsBusy(SPI_MODULE_ID index);
```

Returns

- true - SPI module is currently busy with some transactions
- false - SPI module is currently idle

8.3.1.9. LA FONCTION `PLIB_SPI_FIFOInterruptModeSelect`

La fonction `PLIB_SPI_FIFOInterruptModeSelect` permet de configurer le comportement de l'interruption SPI en relation avec la situation des tampons de réception et d'émission, ceci pour autant que l'on ait activé l'utilisation du buffer avancé (à l'aide de la fonction `PLIB_SPI_FIFOEnable`).

Son prototype est le suivant :

```
void PLIB_SPI_FIFOInterruptModeSelect(SPI_MODULE_ID index,
                                      SPI_FIFO_INTERRUPT mode)
```

Le type énuméré `SPI_FIFO_INTERRUPT` permet les choix suivants :

```
typedef enum {
    SPI_FIFO_INTERRUPT_WHEN_TRANSMISSION_IS_COMPLETE = 3,
    SPI_FIFO_INTERRUPT_WHEN_BUFFER_IS_EMPTY = 7,
    SPI_FIFO_INTERRUPT_WHEN_TRANSMIT_BUFFER_IS_NOT_FULL = 0,
    SPI_FIFO_INTERRUPT_WHEN_TRANSMIT_BUFFER_IS_1HALF_EMPTY_OR_MORE = 1,
    SPI_FIFO_INTERRUPT_WHEN_TRANSMIT_BUFFER_IS_COMPLETELY_EMPTY = 2,
    SPI_FIFO_INTERRUPT_WHEN_RECEIVE_BUFFER_IS_FULL = 4,
    SPI_FIFO_INTERRUPT_WHEN_RECEIVE_BUFFER_IS_1HALF_FULL_OR_MORE = 5,
    SPI_FIFO_INTERRUPT_WHEN_RECEIVE_BUFFER_IS_NOT_EMPTY = 6
} SPI_FIFO_INTERRUPT;
```

L'utilisation de l'interruption est particulièrement utile en mode SLAVE pour réagir rapidement aux données reçues afin de répondre.

8.3.2. LES FONCTIONS DU TRANSMETTEUR

Les 3 fonctions retournent un booléen indiquant la situation du tampon de transmission.

Name	Description
<code>PLIB_SPI_TransmitBufferIsEmpty</code>	Returns the current status of the transmit buffer.
<code>PLIB_SPI_TransmitBufferIsFull</code>	Returns the current transmit buffer status of the SPI module.
<code>PLIB_SPI_TransmitUnderRunStatusGet</code>	Returns the current status of the transmit underrun.

8.3.3. LES FONCTIONS DU RÉCEPTEUR

Les 3 premières fonctions retournent un booléen indiquant la situation du tampon de réception. Il est à remarquer un manque de cohérence dans les noms des fonctions puisque le "Buffer" devient "FIFO" lorsque l'on cherche à savoir s'il est vide.

Name	Description
<code>PLIB_SPI_ReceiverBufferIsFull</code>	Returns the current status of the SPI receive buffer.
<code>PLIB_SPI_ReceiverFIFOIsEmpty</code>	Returns the current status of the SPI receive FIFO.
<code>PLIB_SPI_ReceiverHasOverflowed</code>	Returns the current status of the SPI receiver overflow.
<code>PLIB_SPI_ReceiverOverflowClear</code>	Clears the SPI receive overflow flag.

8.3.4. LES FONCTIONS DE "DATA TRANSFER"

Le système SPI dispose d'un tampon pour la transmission et la réception, d'où l'existence de fonction pour la gestion de ce tampon (Buffer).

Name	Description
<code>PLIB_SPI_BufferClear</code>	Clears the SPI buffer.
<code>PLIB_SPI_BufferRead</code>	Returns the SPI buffer value.
<code>PLIB_SPI_BufferAddressGet</code>	Returns the address of the SPIxBUF (Transmit(SPIxTXB) and Receive (SPIxRXB)) register.
<code>PLIB_SPI_BufferRead16bit</code>	Returns 16-bit SPI buffer value.
<code>PLIB_SPI_BufferRead32bit</code>	Returns 32-bit SPI buffer value.
<code>PLIB_SPI_BufferWrite</code>	Write the data to the SPI buffer.
<code>PLIB_SPI_BufferWrite16bit</code>	Writes 16-bit data to the SPI buffer.
<code>PLIB_SPI_BufferWrite32bit</code>	Write 32-bit data to the SPI buffer.

Nous limitons l'étude aux fonctions de transfert 8 bits ainsi qu'à la fonction d'effacement du buffer.

8.3.4.1. LA FONCTION `PLIB_SPI_BUFFERCLEAR`

La fonction `PLIB_SPI_BufferClear` permet d'effacer le contenu du tampon (émission et réception). Exemple :

```
PLIB_SPI_BufferClear(KitSpil);
```

8.3.4.2. LA FONCTION `PLIB_SPI_BUFFERREAD`

La fonction `PLIB_SPI_BufferRead` permet d'obtenir un élément 8 bits du tampon de réception. Son prototype est le suivant :

```
uint8_t PLIB_SPI_BufferRead(SPI_MODULE_ID index)
```

👉 Avant de lire il faut s'assurer que le tampon ne soit pas vide !

8.3.4.3. LA FONCTION `PLIB_SPI_BUFFERWRITE`

La fonction `PLIB_SPI_BufferWrite` permet de déposer un élément 8 bits dans le tampon de transmission. Son prototype est le suivant :

```
void PLIB_SPI_BufferWrite(SPI_MODULE_ID index,
                          uint8_t data)
```

👉 Avant d'écrire, il faut s'assurer que le tampon ne soit pas plein !

8.4. ETUDE DU DRIVER SPI FOURNI PAR MHC

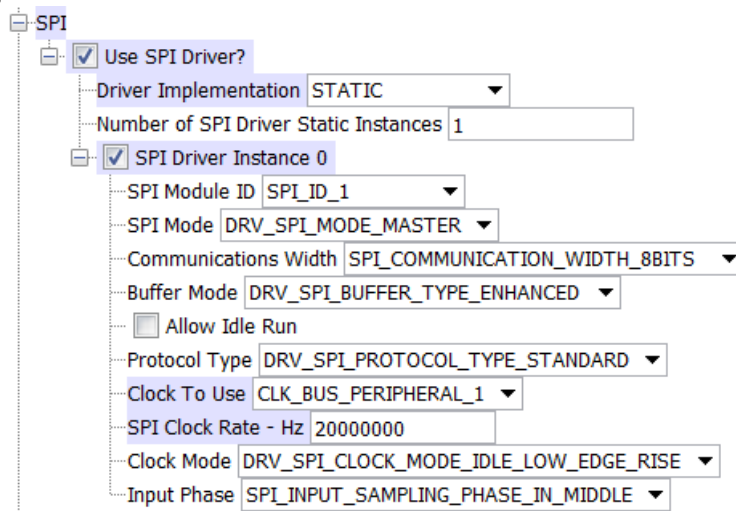
L'étude du driver SPI static généré par le MHC) nous fournit un exemple d'utilisation des fonctions de la PLIB_SPI.

Cette partie a été réalisée avec les logiciels suivants :

- Harmony v1.06
- MPLABX IDE v3.10
- XC32 v1.40

8.4.1. CONFIGURATION DU DRIVER

Voici la configuration effectuée.



Les modifications apportées sont l'utilisation du CLK_BUS_PERIPHERAL_1 au lieu du 2 et une réduction du SPI Clock Rate à 20 MHz.

Le choix des valeurs pour les sections Clock Mode et Input Phase sont à effectuer en relation avec le composant slave que l'on va utiliser.

8.4.2. LA FONCTION DRV_SPI0_INITIALIZE

Voici le contenu de la fonction DRV_SPI0_Initialize

```
void DRV_SPI0_Initialize(void)
{
    PLIB_SPI_Disable(SPI_ID_1);
    PLIB_SPI_MasterEnable(SPI_ID_1);
    PLIB_SPI_SlaveSelectEnable(SPI_ID_1);
    PLIB_SPI_StopInIdleDisable(SPI_ID_1);
    PLIB_SPI_ClockPolaritySelect(SPI_ID_1,
                                SPI_CLOCK_POLARITY_IDLE_LOW);
    PLIB_SPI_OutputDataPhaseSelect(SPI_ID_1,
                                   SPI_OUTPUT_DATA_PHASE_ON_IDLE_TO_ACTIVE_CLOCK);
    PLIB_SPI_InputSamplePhaseSelect(SPI_ID_1,
                                    SPI_INPUT_SAMPLING_PHASE_IN_MIDDLE);
    PLIB_SPI_CommunicationWidthSelect(SPI_ID_1,
                                       SPI_COMMUNICATION_WIDTH_8BITS);
    PLIB_SPI_FramedCommunicationDisable(SPI_ID_1);
    PLIB_SPI_FIFOEnable(SPI_ID_1);
}
```

```

    PLIB_SPI_BaudRateSet(SPI_ID_1,
                        SYS_CLK_PeripheralFrequencyGet
                        (CLK_BUS_PERIPHERAL_1), 20000000);
    PLIB_SPI_Enable(SPI_ID_1);
}

```

Nous reprendrons le principe de la configuration effectuée dans DRV_SPI0_Initialize pour l'appliquer à des fonctions d'initialisation spécifiques au DAC et au LM70.

8.4.3. LA FONCTION DRV_SPI0_RECEIVERBUFFERISFULL

La fonction DRV_SPI0_ReceiverBufferIsFull nous montre comment tester si le tampon de réception est plein :

```

bool DRV_SPI0_ReceiverBufferIsFull(void)
{
    return (PLIB_SPI_ReceiverBufferIsFull(SPI_ID_1));
}

```

8.4.4. LA FONCTION DRV_SPI0_TRANSMITTERBUFFERISFULL

La fonction DRV_SPI0_TransmitterBufferIsFull nous montre comment tester si le tampon d'émission est plein :

```

bool DRV_SPI0_TransmitterBufferIsFull(void)
{
    return (PLIB_SPI_TransmitBufferIsFull(SPI_ID_1));
}

```

8.4.5. LA FONCTION DRV_SPI0_BUFFERADDWITEREAD

La fonction DRV_SPI0_BufferAddWriteRead nous montre comment gérer un échange bidirectionnel :

```

int32_t DRV_SPI0_BufferAddWriteRead(const void * txBuffer,
                                     void * rxBuffer, uint32_t size)
{
    bool continueLoop;
    int32_t txcounter = 0;
    int32_t rxcounter = 0;
    uint8_t unitsTxed = 0;
    const uint8_t maxUnits = 16;
    do {
        continueLoop = false;
        unitsTxed = 0;
        if (PLIB_SPI_TransmitBufferIsEmpty(SPI_ID_1))
        {
            while (!PLIB_SPI_TransmitBufferIsFull(SPI_ID_1)
                    && (txcounter < size)
                    && unitsTxed != maxUnits)
            {
                PLIB_SPI_BufferWrite(SPI_ID_1,
                                      ((uint8_t*)txBuffer)[txcounter]);
                txcounter++;
                continueLoop = true;
            }
        }
    } while (continueLoop);
    if (PLIB_SPI_ReceiveBufferIsFull(SPI_ID_1))
    {
        while (!PLIB_SPI_ReceiveBufferIsEmpty(SPI_ID_1)
                && (rxcounter < size)
                && unitsRxed != maxUnits)
        {
            PLIB_SPI_BufferRead(SPI_ID_1,
                                ((uint8_t*)rxBuffer)[rxcounter]);
            rxcounter++;
            continueLoop = true;
        }
    }
    return (txcounter + rxcounter);
}

```

```
        unitsTxed++;
    }
}

while (txcounter != rxcounter)
{
    while (PLIB_SPI_ReceiverFIFOIsEmpty(SPI_ID_1));
    ((uint8_t*) rxBuffer)[rxcounter] =
        PLIB_SPI_BufferRead(SPI_ID_1);
    rxcounter++;
    continueLoop = true;
}
if (txcounter > rxcounter)
{
    continueLoop = true;
}
}while(continueLoop);
return txcounter;
}
```

Cette fonction transmet un certain nombre de bytes, avec une limite fixée à 16, puis elle lit ce que le slave a fourni.

8.5. RÉALISATION DE L'UTILITAIRE SPI

Les fonctions utilitaires SPI s'inspirent des fonctions SPI du compilateur CCS. L'utilitaire ne s'occupe pas de l'initialisation car elle est spécifique pour chaque composant.

8.5.1. RÉALISATION DE LA FONCTION SPI_WRITE1

Cette fonction effectue l'écriture de 8 bits de données sur le SPI canal 1. L'utilisateur doit gérer le Chip Select avant et après l'utilisation de la fonction.

Cette fonction est fournie dans les fichiers Mc32SpiUtil.c et Mc32SpiUtil.h.

```
void spi_writel( uint8_t Val){
    bool SpiBusy;

    PLIB_SPI_BufferWrite(SPI_ID_1, Val);

    // Attente de la fin de la transmission
    do {
        SpiBusy = PLIB_SPI_IsBusy(SPI_ID_1) ;
    } while (SpiBusy == true);
}
```

La transmission s'effectue en déposant une valeur dans le tampon d'émission (TransmitBuffer) avec la fonction **PLIB_SPI_BufferWrite**.

☞ On aurait pu s'assurer que le tampon d'émission ne soit pas plein avant d'y déposer une nouvelle valeur (à l'aide de la fonction **PLIB_SPI_TransmitBufferIsFull**). Ceci n'est toutefois pas nécessaire car on attend la fin de la transmission avant de quitter la fonction. Ainsi, il y aura forcément de la place disponible dans le tampon d'émission au prochain appel de `spi_writel`.

Le dépôt d'une valeur dans le tampon d'émission déclenche la transmission. Il est nécessaire d'attendre la fin de cette transmission pour gérer le /CS en synchronisation avec la transmission. Utilisation de la fonction **PLIB_SPI_IsBusy**.

8.5.2. RÉALISATION DE LA FONCTION SPI_READ1

Cette fonction effectue la lecture de 8 bits de données sur le SPI canal 1. L'utilisateur doit gérer le Chip Select avant et après l'utilisation de la fonction.

Remarque : la fonction a en paramètre la valeur à transmettre et elle retourne la valeur lue.

```
uint8_t spi_read1( uint8_t Val){
    int SpiBusy;
    uint32_t lu;

    PLIB_SPI_BufferWrite(SPI_ID_1, Val);
    // Attends fin transmission
    do {
        SpiBusy = PLIB_SPI_IsBusy(SPI_ID_1) ;
    } while (SpiBusy == 1);

    // Attend arrivée dans fifo
    while (PLIB_SPI_ReceiverFIFOIsEmpty(SPI_ID_1));
```

```
#ifdef MARKER_READ
    LED3_W = 1;
#endif
    lu = PLIB_SPI_BufferRead(SPI_ID_1);
#ifdef MARKER_READ
    LED3_W = 0;
#endif
    return lu;
}
```

Pour lire, il faut générer des coups d'horloge, ce qui s'effectue avec une fonction d'écriture. Au fur et à mesure que les coups d'horloge sont envoyés, le slave fournit les bits de données.

Il faut attendre la fin de la transmission avec **PLIB_SPI_IsBusy**.

Utilisation de la fonction **PLIB_SPI_ReceiverFIFOIsEmpty** pour attendre tant que le tampon de réception est vide. Ceci est important car si on lit trop vite on manque la valeur.

☹ Si les conditions de départ ne sont pas correctes (tampon vide, pas d'erreur Receive Overflow), il y a risque d'être bloqué ou de ne pas lire la bonne valeur.

L'action **PLIB_SPI_BufferClear**(KitSpi1) est nécessaire dans la configuration pour cela.

8.5.2.1. SÉQUENCE DES ACTIONS DE LECTURE

Il s'agit d'une séquence de lecture de la température du LM70. La séquence est précédée de la reconfiguration, car l'application gère les 2 composants SPI.

```
SPI_ConfigureLM70();

CS_LM70 = 0;
MSB = spi_read1(0xFF);
LSB = spi_read1(0xFF);
//Fin de transmission
CS_LM70 = 1;
```

8.5.2.2. OBSERVATION DU MOMENT DE LECTURE DU TAMPON

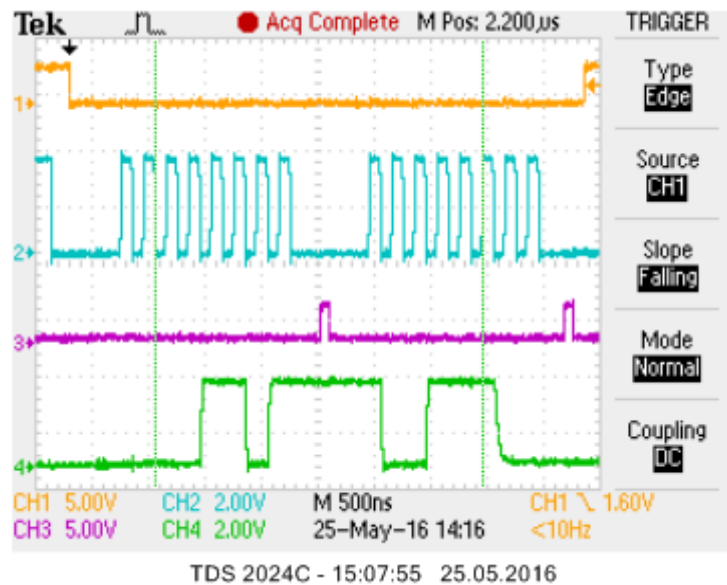
Pour vérifier à quel moment de la transmission s'effectue la lecture du tampon de réception, nous ajoutons une action sur la LED_3 pour observer cela.

Canal 1 : CS_LM70
RD3
broche 78

Canal 2 : SPI-SCK
SCK1/RD10
broche 70

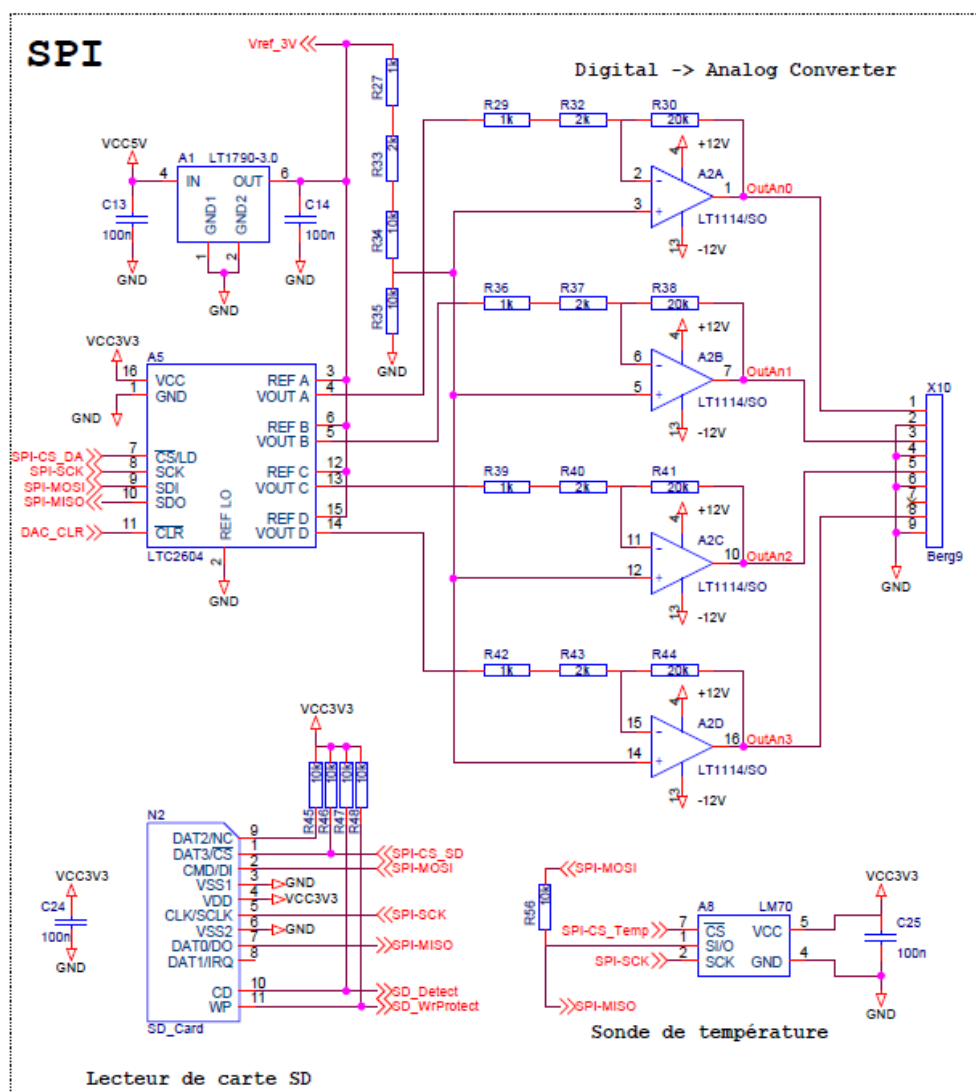
Canal 3 : LED_3

Canal 4 : SPI-MISO
SDI1/RC4
broche 9



On peut observer que l'impulsion sur le canal 3 a lieu à la fin de la série de coups d'horloge. Donc les datas sont lues au bon moment, ce qui se confirme par le résultat en température.

8.7. PÉRIPHÉRIQUES SPI DU KIT PIC32MX



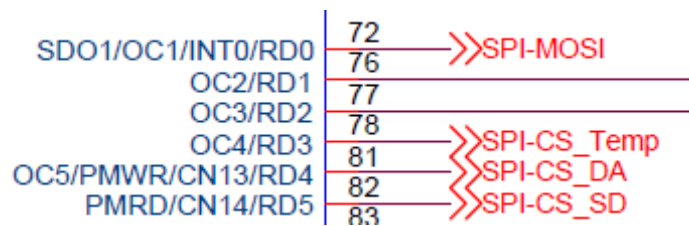
8.8. COMMUNICATION AVEC LE DAC LTC2604

Le dac LTC2604 utilise le module SPI 1.

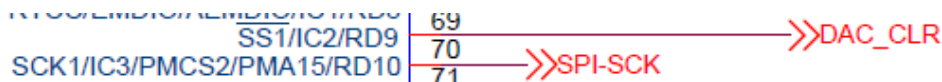
Broche du PIC32MX795F512L (100 pin TQFP)	Nom schéma	No broche boitier 100
SCK1/IC3/PMCS2/PMA15/RD10	SPI-SCK	70
SDO1/OC1/INT0/RD0	SPI-MOSI	72
T5CK/SDI1/RC4	SPI-MISO	9

8.8.1. CHIP SELECT DU LTC2604

Utilisation d'une ligne de port standard (RD4) pour le signal \overline{CS}



On dispose aussi de RD9 pour effectuer un reset du composant.



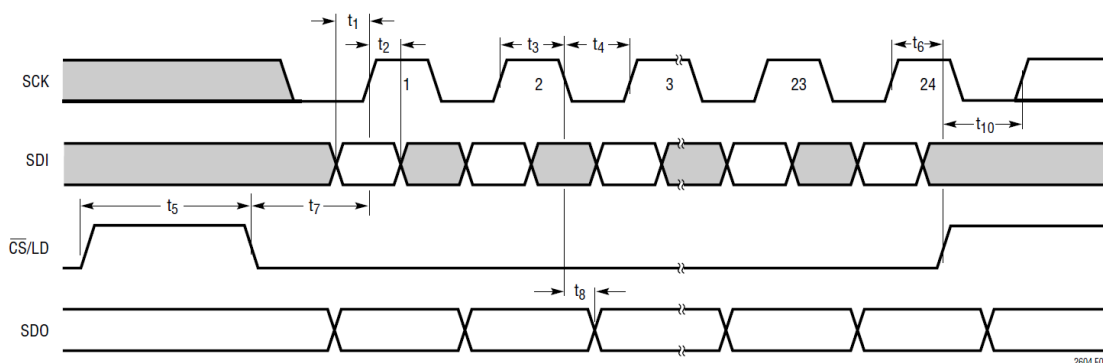
8.8.2. CONFIGURATION SPI NÉCESSAIRE AU LTC2604

Au niveau du timing, voici les caractéristiques :

SYMBOL	PARAMETER	CONDITIONS	LTC2604/LTC2614/LTC2624			UNITS
			MIN	TYP	MAX	
V _{CC} = 2.5V to 5.5V						
t ₁	SDI Valid to SCK Setup		●	4		ns
t ₂	SDI Valid to SCK Hold		●	4		ns
t ₃	SCK High Time		●	9		ns
t ₄	SCK Low Time		●	9		ns
t ₅	CS/LD Pulse Width		●	10		ns
t ₆	LSB SCK High to CS/LD High		●	7		ns
t ₇	CS/LD Low to SCK High		●	7		ns
t ₈	SDO Propagation Delay from SCK Falling Edge	C _{LOAD} = 10pF V _{CC} = 4.5V to 5.5V V _{CC} = 2.5V to 5.5V	● ●		20 45	ns ns
t ₉	CLR Pulse Width		●	20		ns
t ₁₀	CS/LD High to SCK Positive Edge		●	7		ns
	SCK Frequency	50% Duty Cycle	●		50	MHz

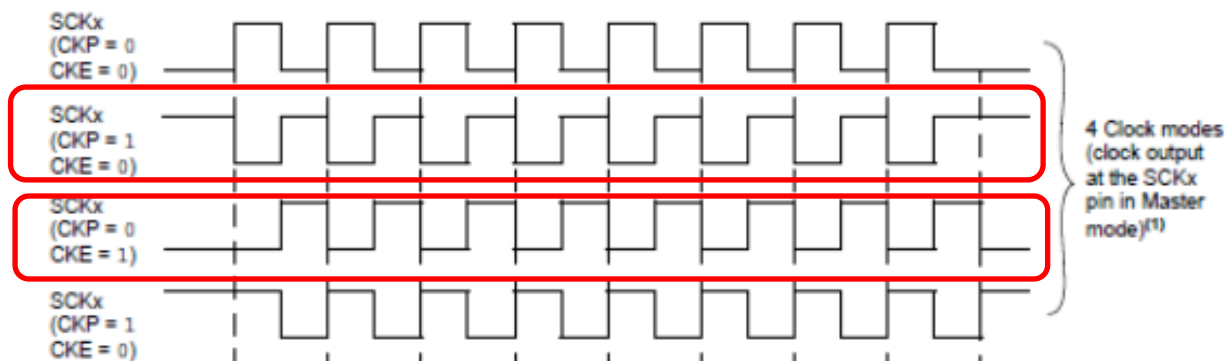
Avec la division du PB_CLOCK par 2, 4, 6, 8, etc., il est possible de travailler à 40 MHz. En pratique, on travaillera à 20 MHz.

Pour le choix du mode d'horloge, le fabricant fournit un diagramme :



On doit donc avoir le flanc montant de l'horloge au milieu des datas.

Ce qui correspond à deux situations : CKP = 1 et CKE = 0 OU CKP = 0 et CKE = 1



8.8.2.1. FONCTION DE CONFIGURATION DU SPI POUR LE DAC

Voici le contenu de la fonction SPI_ConfigureLTC2604 qui est fournie dans le fichier Mc32gestSPiDac.c. Cette fonction reprend les mêmes éléments que la fonction DRV_SPI0_Initialize, mais avec l'ajout d'une action PLIB_SPI_BufferClear.

```
void SPI_ConfigureLTC2604(void)
{
    PLIB_SPI_Disable(KitSpil);
    PLIB_SPI_BufferClear(KitSpil);
    PLIB_SPI_StopInIdleDisable(KitSpil);
    PLIB_SPI_PinEnable(KitSpil, SPI_PIN_DATA_OUT);
    PLIB_SPI_CommunicationWidthSelect(KitSpil,
                                       SPI_COMMUNICATION_WIDTH_8BITS);
    // Config SPI clock à 20 MHz
    PLIB_SPI_BaudRateSet(KitSpil,
                        SYS_CLK_PeripheralFrequencyGet(CLK_BUS_PERIPHERAL_1),
                        20000000);

    // Config polarité traitement des signaux SPI
    // pour input à confirmer
    // Polarité clock OK
    // Phase output à confirmer
    PLIB_SPI_InputSamplePhaseSelect(KitSpil,
                                    SPI_INPUT_SAMPLING_PHASE_IN_MIDDLE );
}
```

```

    PLIB_SPI_ClockPolaritySelect(KitSpi1,
                                SPI_CLOCK_POLARITY_IDLE_HIGH);
    PLIB_SPI_OutputDataPhaseSelect(KitSpi1,
                                   SPI_OUTPUT_DATA_PHASE_ON_IDLE_TO_ACTIVE_CLOCK);
    PLIB_SPI_MasterEnable(KitSpi1);
    PLIB_SPI_FramedCommunicationDisable(KitSpi1);
    PLIB_SPI_FIFOEnable(KitSpi1);    // Enhanced buffer mode
    PLIB_SPI_Enable(KitSpi1);

    // Contrôle de la configuration
    ConfigReg = SPI1CON;
    BaudReg = SPI1BRG;
}

```

Comme on peut le constater, les fonctions sont assez nombreuses, car elles correspondent aux groupes de bits du registre SPIxCON.

Pour vérifier la configuration on peut lire le registre de configuration.

[SPI1CON]	31	30	29	28	27	24	17
	FRMEN	FRMSYNC	FRMPOL	MSEN	FRMSYPW	FRMCNT	- SPIFE
	0	0	0	0	0	000	- 0
	16	15	13	12	11	10	9
	ENHBUF	ON	- SIDL	DISSDO	MODE32	MODE16	SMP
	1	1	-	0	0	0	0
	8	7	6	5	2	0	
	CKE	SEN	CKP	MSTEN	- STXISEL	SRXISEL	
	0	0	1	1	-	00	00

On obtient la valeur 0x00018020, ce qui nous donne CKP = 1 et CKE = 0. Cela correspond au premier choix avec l'horloge qui démarre au niveau haut.

Si on modifie :

```

    PLIB_SPI_ClockPolaritySelect(KitSpi1,
                                SPI_CLOCK_POLARITY_IDLE_HIGH);

```

En :

```

    PLIB_SPI_ClockPolaritySelect(KitSpi1,
                                SPI_CLOCK_POLARITY_IDLE_LOW);

```

On obtient CKP = 0 et CKE = 0 et on n'obtient plus de signal sur la sortie du DAC !

8.8.3. DÉTAIL DU SPIxCON

 Register 23-1: SPIxCON: SPI Control Register^(1,2,3)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
FRMEN	FRMSYNC	FRMPOL	MSEN ⁽⁴⁾	FRMSYPW ⁽⁴⁾	FRMCNT<2:0> ⁽⁴⁾		
bit 31					bit 24		
r-x	r-x	r-x	r-x	r-x	r-x	R/W-0	R/W-0
—	—	—	—	—	—	SPIFE	ENHBUF ⁽⁴⁾
bit 23					bit 16		
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ON	FRZ	SIDL	DISSDO	MODE32	MODE16	SMP	CKE
bit 15					bit 8		
R/W-0	R/W-0	R/W-0	r-x	R/W-0	R/W-0	R/W-0	R/W-0
SSEN	CKP	MSTEN	—	STXISEL<1:0> ⁽⁴⁾	SRXISEL<1:0> ⁽⁴⁾		
bit 7					bit 0		

Legend:
R = Readable bit
W = Writable bit
P = Programmable bit
r = Reserved bit
U = Unimplemented bit
-n = Bit Value at POR: ('0', '1', x = Unknown)

- bit 31 **FRMEN**: Framed SPI Support bit
 1 = Framed SPI support is enabled (\overline{SSx} pin used as FSYNC input/output)
 0 = Framed SPI support is disabled
- bit 30 **FRMSYNC**: Frame Sync Pulse Direction Control on \overline{SSx} pin bit (Framed SPI mode only)
 1 = Frame sync pulse input (Slave mode)
 0 = Frame sync pulse output (Master mode)
- bit 29 **FRMPOL**: Frame Sync Polarity bit (Framed SPI mode only)
 1 = Frame pulse is active-high
 0 = Frame pulse is active-low
- bit 28 **MSEN**: Master Mode Slave Select Enable bit⁽⁴⁾
 1 = Slave select SPI support enabled. The \overline{SS} pin is automatically driven during transmission in Master mode. Polarity is determined by the FRMPOL bit.
 0 = Slave select SPI support is disabled.
- bit 27 **FRMSYPW**: Frame Sync Pulse Width bit⁽⁴⁾
 1 = Frame sync pulse is one character wide
 0 = Frame sync pulse is one clock wide

 Register 23-1: SPIxCON: SPI Control Register^(1,2,3) (Continued)

- bit 26-24 **FRMCNT<2:0>**: Frame Sync Pulse Counter bits. Controls the number of data characters transmitted per pulse.⁽⁴⁾
 111 = Reserved; do not use
 110 = Reserved; do not use
 101 = Generate a frame sync pulse on every 32 data characters
 100 = Generate a frame sync pulse on every 16 data characters
 011 = Generate a frame sync pulse on every 8 data characters
 010 = Generate a frame sync pulse on every 4 data characters
 001 = Generate a frame sync pulse on every 2 data characters
 000 = Generate a frame sync pulse on every data character
Note: This bit is only valid in FRAMED_SYNC mode.
- bit 23-18 **Reserved**: Write '0'; ignore read
- bit 17 **SPIFE**: Frame Sync Pulse Edge Select bit (Framed SPI mode only)
 1 = Frame synchronization pulse coincides with the first bit clock
 0 = Frame synchronization pulse precedes the first bit clock
- bit 16 **ENHBUF**: Enhanced Buffer Enable bit⁽⁴⁾
 1 = Enhanced Buffer mode is enabled
 0 = Enhanced Buffer mode is disabled

bit 15	ON: SPI Peripheral On bit 1 = SPI Peripheral is enabled 0 = SPI Peripheral is disabled Note: When using the 1:1 PBCLK divisor, the user's software should not read or write the peripheral's SFRs in the SYSCLK cycle immediately following the instruction that clears the module's ON bit.
bit 14	FRZ: Freeze in Debug Exception Mode bit 1 = Freeze operation when CPU enters Debug Exception mode 0 = Continue operation when CPU enters Debug Exception mode Note: FRZ is writable in Debug Exception mode only, it is forced to '0' in Normal mode.
bit 13	SIDL: Stop in Idle Mode bit 1 = Discontinue operation when CPU enters in Idle mode 0 = Continue operation in Idle mode
bit 12	DISSDO: Disable SDOx pin bit 1 = SDOx pin is not used by the module. Pin is controlled by associated PORT register 0 = SDOx pin is controlled by the module
bit 11-10	MODE<32,16>: 32/16-Bit Communication Select bits 1x = 32-bit data width 01 = 16-bit data width 00 = 8-bit data width

- Note 1:** This register has an associated Clear register (SPIxCONCLR) at an offset of 0x4 bytes. Writing a '1' to any bit position in the Clear register will clear valid bits in the associated register. Reads from the Clear register should be ignored.
- 2:** This register has an associated Set register (SPIxCONSET) at an offset of 0x8 bytes. Writing a '1' to any bit position in the Set register will set valid bits in the associated register. Reads from the Set register should be ignored.
- 3:** This register has an associated Invert register (SPIxCONINV) at an offset of 0xC bytes. Writing a '1' to any bit position in the Invert register will invert valid bits in the associated register. Reads from the Invert register should be ignored.
- 4:** These bits are not available on all devices. Refer to the specific device data sheet for availability.

Register 23-1: SPIxCON: SPI Control Register^(1,2,3) (Continued)

bit 9	SMP: SPI Data Input Sample Phase bit Master mode (MSTEN = 1): 1 = Input data sampled at end of data output time 0 = Input data sampled at middle of data output time Slave mode (MSTEN = 0): SMP value is ignored when SPI is used in Slave mode. The module always uses SMP = 0.
bit 8	CKE: SPI Clock Edge Select bit 1 = Serial output data changes on transition from active clock state to Idle clock state (see CKP bit) 0 = Serial output data changes on transition from Idle clock state to active clock state (see CKP bit) Note: The CKE bit is not used in the Framed SPI mode. The user should program this bit to '0' for the Framed SPI mode (FRMEN = 1).
bit 7	SSEN: Slave Select Enable (Slave mode) bit 1 = \overline{SSx} pin used for Slave mode 0 = \overline{SSx} pin not used for Slave mode, pin controlled by port function.
bit 6	CKP: Clock Polarity Select bit 1 = Idle state for clock is a high level; active state is a low level 0 = Idle state for clock is a low level; active state is a high level
bit 5	MSTEN: Master Mode Enable bit 1 = Master mode 0 = Slave mode
bit 4	Reserved: Write '0'; ignore read
bit 3-2	STXISEL<1:0>: SPI Transmit Buffer Empty Interrupt Mode bits⁽⁴⁾ 11 = SPI_TBE_EVENT is set when the buffer is not full (has one or more empty elements) 10 = SPI_TBE_EVENT is set when the buffer is empty by one-half or more 01 = SPI_TBE_EVENT is set when the buffer is completely empty 00 = SPI_TBE_EVENT is set when the last transfer is shifted out of SPIxR and transmit operations are complete
bit 1-0	RTXISEL<1:0>: SPI Receive Buffer Full Interrupt Mode bits⁽⁴⁾ 11 = SPI_RBF_EVENT is set when the buffer is full 10 = SPI_RBF_EVENT is set when the buffer is full by one-half or more 01 = SPI_RBF_EVENT is set when the buffer is not empty 00 = SPI_RBF_EVENT is set when the last word in the receive buffer is read (i.e., buffer is empty)

8.8.4. SÉLECTION DE LA FRÉQUENCE DE SCK

Voici la formule qui détermine la fréquence de SCK en fonction de la fréquence de PB_CLOCK et du registre SPIxBRG.

$$F_{SCK} = \frac{F_{PB}}{2 \cdot (SPIxBRG + 1)}$$

Ce qui donne des divisions possibles par 2, 4, 6, 8, etc.

Avec PB_CLOCK = 80 MHz et SPIxBRG = 0, la fréquence maximum de SCK sera de 40 MHz.

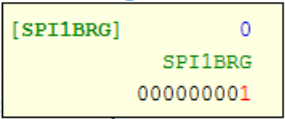
Lors de la configuration, nous avons établi :

```
PLIB_SPI_BaudRateSet(KitSpil,
    SYS_CLK_PeripheralFrequencyGet(CLK_BUS_PERIPHERAL_1),
    20000000);
```

Pour obtenir 20 MHz, il faut diviser par 4 donc SPIxBRG doit valoir 1. Ce que nous vérifions en lisant la valeur de SPI1BRG.

```
PLIB_SPI_Enable(KitSpil);

// Contrôle 1
ConfigReg = SPI1CON1;
BaudReg = SPI1BRG;
```



8.8.5. INITIALISATION DU LTC2604

Voici la fonction d'initialisation complète avec le reset du LTC2604.

```
void SPI_InitLTC2604(void)
{
    //Initialisation SPI DAC
    CS_DAC = 1;
    // Impulsion reset du DAC
    DAC_CLEAR = 0;
    delay_us(500);
    DAC_CLEAR = 1;

    // LTC2604 MAX 50 MHz choix 20 MHz
    SPI_ConfigureLTC2604();
}
```

8.8.6. ECRITURE D'UNE VALEUR SUR LE LTC2604

Voici la fonction d'écriture sur le LTC2604. Dans le but de partager le bus SPI avec un autre composant, la fonction d'écriture reconfigure le SPI avant l'action.

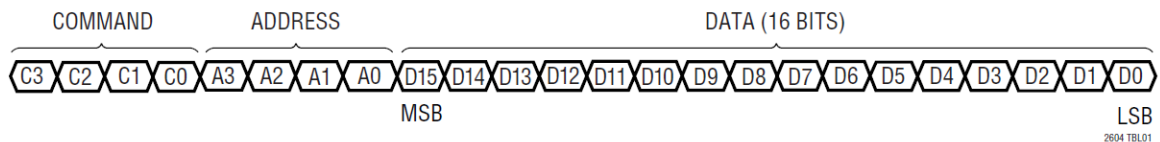
```
// Envoi d'une valeur sur le DAC LTC2604
// Avec reconfiguration du SPI
// Indication du canal 0 à 3
void SPI_CfgWriteToDac(uint8_t NoCh, uint16_t DacVal)
{
    uint8_t MSB;
    uint8_t LSB;

    // Reconfiguration du SPI
    SPI_ConfigureLTC2604();
    //Sélection du canal
    //3 -> Set and Update, 0/1/2/3 Sélection canal A/B/C/D,
    //                                     F tous canaux
    NoCh = NoCh + 0x30;
    MSB = DacVal >> 8;
    LSB = DacVal;

    CS_DAC = 0;
    spi_writel(NoCh);
    spi_writel(MSB);
    spi_writel(LSB);

    // Fin de transmission
    CS_DAC = 1;
} // SPI_CfgWriteToDac
```

Ce qui correspond à l'indication du fabricant :



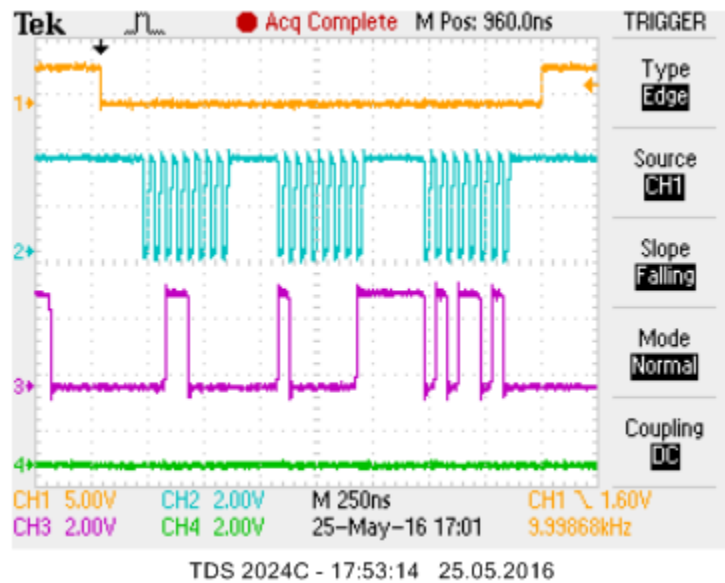
8.8.7. OBSERVATION DES SIGNAUX DU DAC

Canal 1 : CS_DAC
RD4
broche 81

Canal 2 : SPI-SCK
SCK1/RD10
broche 70

Canal 3 : SPI-MOSI
SDO1/RD0
broche 72

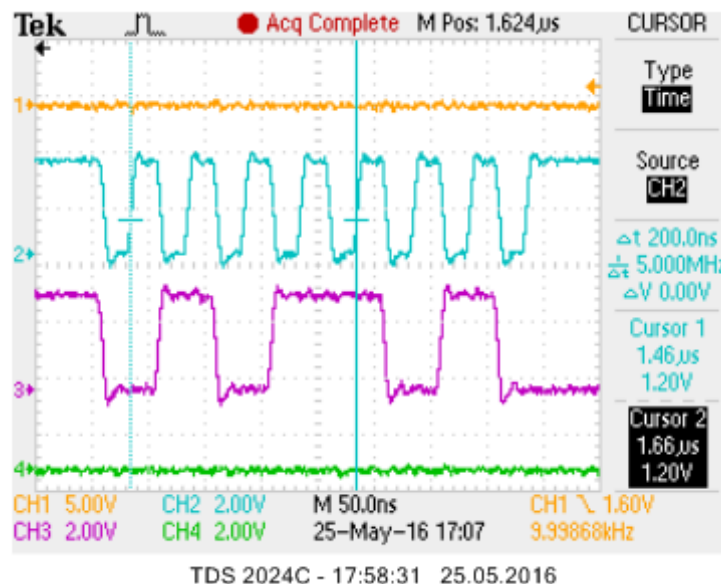
Canal 4 : sortie du
DAC 0



On observe les 3 salves de 8 coups d'horloge. La valeur envoyée est 0x81 pour le MSB et 0x5A sur le LSB (3^{ème} data).

8.8.7.1. DÉTAIL DU 3ÈME OCTET

La valeur du 3^{ème} octet (LSB) est imposée à 0x5A. On peut donc vérifier que le flanc montant du clock est au milieu du data. Au 1^{er} flanc montant on a bien un 0 et au 5^{ème} flanc un 1.



Les 4 périodes d'horloge valent 200 ns, ce qui nous donne 50 ns pour une période donc une fréquence de 20 MHz comme prévu.

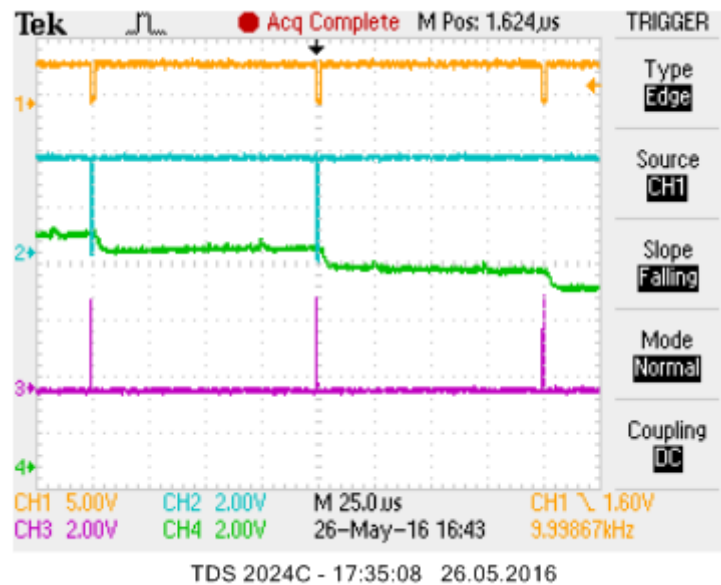
Le fait que l'horloge démarre sur un état haut ne pose pas de problème.

8.8.7.2. SIGNAL SUR LE DAC

Si on transmet des valeurs qui se suivent, comme par exemple :

```
SPI_CfgWriteToDac(0, DacVal);  
DacVal += 2048;
```

On peut observer les changements de valeur du signal, ce qui prouve la bonne interprétation par le DAC.



8.10. COMMUNICATION AVEC LE LM70

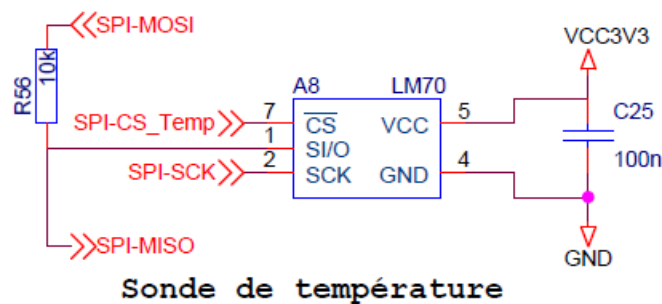
Nous allons illustrer principalement l'utilisation des fonctions de lecture du SPI par la communication avec le composant LM70 qui est un capteur de température.

8.10.1. CONNEXIONS ENTRE LE PIC32 ET LE LM70

Utilisation du SPI1 et d'une ligne de port standard (RD3) pour le signal \overline{CS}

Broche du PIC32MX795F512L (100 pin TQFP)	Nom schéma	No broche boîtier 100
SCK1/IC3/PMCS2/PMA15/RD10	SPI-SCK	70
SDO1/OC1/INT0/RD0	SPI-MOSI	72
SPI-MISO	SPI-MISO	9
UC3/RD2 OC4/RD3	SPI-CS_Temp	78

Comme le LM70 ne possède qu'une ligne bidirectionnelle, il est nécessaire d'utiliser une résistance pour combiner les lignes SPI-MOSI et SPI-MISO SDO du PIC.

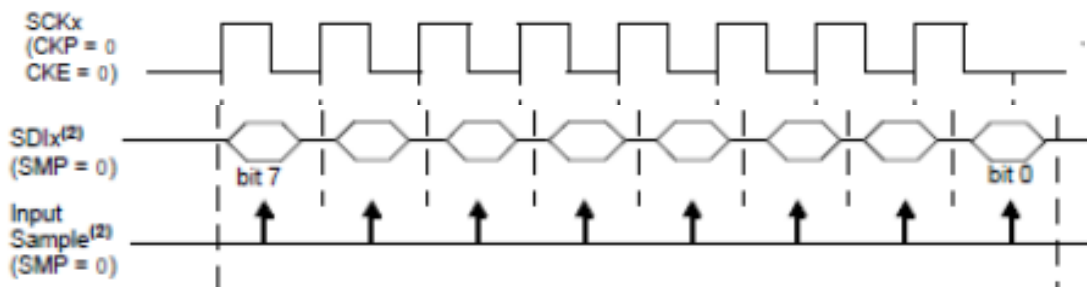


8.10.2. CONFIGURATION SPI NÉCESSAIRE AU LM70

Le fabricant du LM70 indique :

The LM70 operates as a slave and is compatible with SPI or MICROWIRE bus specifications. Data is clocked out on the falling edge of the serial clock (SC), while data is clocked in on the rising edge of SC.

Pour la lecture du LM70 par le maître, les données sont fournies par l'esclave au flanc descendant de SCK, tandis que pour l'écriture par le maître, les datas sont lues au flanc montant de l'horloge. Ce qui correspond à CKP = 0 et CKE = 0.



8.10.2.1. FONCTION DE CONFIGURATION DU SPI POUR LE LM70

Voici le contenu de la fonction SPI_ConfigureLM70 qui est fournie dans les fichiers Mc32GestSpiLM70.h et Mc32GestSpiLM70.c.

```
void SPI_ConfigureLM70(void)
{
    PLIB_SPI_Disable(KitSpil);

    PLIB_SPI_BufferClear(KitSpil);
    PLIB_SPI_StopInIdleDisable(KitSpil);
    PLIB_SPI_PinEnable(KitSpil, SPI_PIN_DATA_OUT);
    PLIB_SPI_CommunicationWidthSelect(KitSpil,
                                       SPI_COMMUNICATION_WIDTH_8BITS);
    // LM70 MAX 6.25 MHz choix 5 MHz
    PLIB_SPI_BaudRateSet(KitSpil,
                        SYS_CLK_PeripheralFrequencyGet(CLK_BUS_PERIPHERAL_1),
                        5000000);

    // Config polarité traitement des signaux SPI
    // pour input à confirmer
    // Polarité clock OK
    // Phase output à confirmer
    PLIB_SPI_InputSamplePhaseSelect(KitSpil,
                                    SPI_INPUT_SAMPLING_PHASE_IN_MIDDLE);
    PLIB_SPI_ClockPolaritySelect(KitSpil,
                                SPI_CLOCK_POLARITY_IDLE_LOW);
    PLIB_SPI_OutputDataPhaseSelect(KitSpil,
                                   SPI_OUTPUT_DATA_PHASE_ON_IDLE_TO_ACTIVE_CLOCK);
    PLIB_SPI_MasterEnable(KitSpil);
    PLIB_SPI_FramedCommunicationDisable(KitSpil);
    PLIB_SPI_FIFOEnable(KitSpil); // Enhanced buffer mode

    PLIB_SPI_Enable(KitSpil);

    // Contrôle de la configuration
    ConfigReg = SPI1CON;
    BaudReg = SPI1BRG;
}
```

8.10.2.2. VÉRIFICATION DE LA CONFIGURATION AVEC SPI1CON

[SPI1CON]	31	30	29	28	27	24	17
	FRMEN	FRMSYNC	FRMPOL	MSEN	FRMSYPW	FRMCNT	- SPIFE
	0	0	0	0	0	000	- 0
	16	15	13	12	11	10	9
	ENHBUF	ON	- SIDL	DISSDO	MODE32	MODE16	SMP
	1	1	- 0	0	0	0	0
	8	7	6	5	2	0	
	CKE	SSEN	CKP	MSTEN	- STXISEL	SRXISEL	
	0	0	0	1	- 00	00	

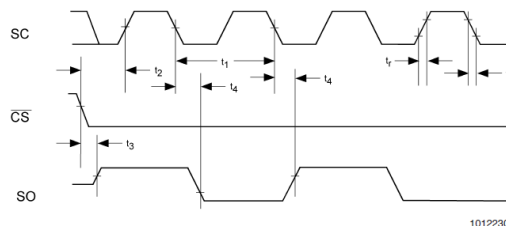
On a bien confirmation de CKE = 0 et CKP = 0.

Serial Bus Digital Switching Characteristics

Unless otherwise noted, these specifications apply for $V^+ = 2.65V$ to $3.6V$ for the LM70-3 and $V^+ = 4.5V$ to $5.5V$ for the LM70-5, C_L (load capacitance) on output lines = 100 pF unless otherwise specified. **Boldface limits apply for $T_A = T_J = T_{MIN}$ to T_{MAX} ; all other limits $T_A = T_J = +25^\circ\text{C}$, unless otherwise noted.**

Symbol	Parameter	Conditions	Typical (Note 7)	Limits (Note 8)	Units (Limit)
t_1	SC (Clock) Period			0.16 DC	μs (min) (max)
t_2	CS Low to SC (Clock) High Set-Up Time			100	ns (min)
t_3	CS Low to Data Out (SO) Delay			70	ns (max)
t_4	SC (Clock) Low to Data Out (SO) Delay			70	ns (max)
t_5	CS High to Data Out (SO) TRI-STATE			200	ns (min)
t_6	SC (Clock) High to Data In (SI) Hold Time			60	ns (min)
t_7	Data In (SI) Set-Up Time to SC (Clock) High			30	ns (min)

Timing Diagrams



10122304

Le minimum de $0,16\text{ }\mu\text{s}$ pour le serial clock correspond à un max de 6.25 MHz . Avec la division par 16 on obtient si la fréquence d'oscillateur est de 80 MHz , $80/16 = 5\text{ MHz}$.

Lors de la configuration nous avons établi :

```
PLIB_SPI_BaudRateSet(KitSpi1,
    SYS_CLK_PeripheralFrequencyGet(CLK_BUS_PERIPHERAL_1),
    5000000);
```

Pour obtenir 5 MHz , il faut diviser par 16 donc SPIxBRG doit valoir 7, car $2(7+1) = 16$. Ce que nous vérifions en lisant la valeur de SPI1BRG.

```
61  PLIB_SPI_BaudRateSet(KitSpi1,
62  // Contrôle le
63  ConfigReg = SPI1BRG;
    BaudReg = SPI1BRG;
```

[SPI1BRG] 0
SPI1BRG
000000111

...ce qui correspond bien à 7.

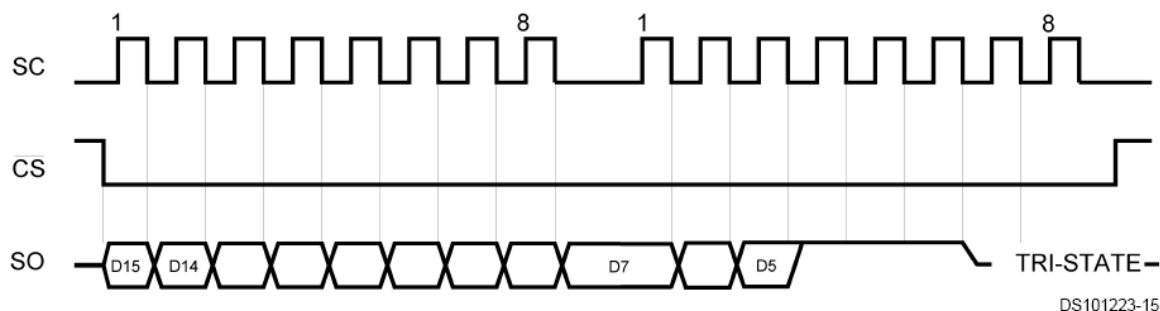
8.10.3. INITIALISATION DU LM70

La fonction **SPI_InitLM70** effectue la configuration du SPI pour le LM70 et effectue une séquence permettant de configurer les registres du LM70.

```
void SPI_InitLM70(void) {
    SPI_ConfigureLM70();
    // action de configuration
    CS_LM70 = 0;
    spi_read1(0xFF);
    spi_read1(0xFF);
    spi_read1(0); // pour écrire 0
    spi_read1(0); // pour écrire 0
    //Fin de transmission
    CS_LM70 = 1;
} // SPI_InitLM70
```

8.10.4. LECTURE DU LM70

Pour réaliser une lecture, il faut respecter la séquence ci-dessous :



b) Reading Continuous Conversion - Two Eight-Bit Frames

On remarque que le signal \overline{CS} doit être activé au début de l'action et désactivé à la fin de l'action. La lecture s'effectue en deux trains de 8 bits, ce qui correspond bien à l'usage du MSSP.

8.10.5. LA FONCTION `SPI_ReadRawTempLM70`

La fonction `SPI_ReadRawTempLM70` effectue la lecture en appelant 2 fois la fonction `spi_read1`. On obtient d'abord le poids fort et ensuite le poids faible. La valeurs des 2 octets envoyés n'a pas d'importance.

```
// Lecture du registre de température du LM70
// Version sans reconfiguration
int16_t SPI_ReadRawTempLM70(void)
{
    //Déclaration des variables
    uint8_t MSB;
    uint8_t LSB;
    int16_t RawTemp;

    CS_LM70 = 0;
    MSB = spi_read1(0xFF);
    LSB = spi_read1(0xFF);
    //Fin de transmission
    CS_LM70 = 1;

    RawTemp = MSB;
    RawTemp = RawTemp << 8;
    RawTemp = RawTemp | LSB;
    return RawTemp;
} // SPI_ReadRawTempLM70
```

Pour obtenir la température il faut combiner le msb et le lsb et manipuler la valeur en tenant compte des détails du registre de température :

1.5.2 TEMPERATURE REGISTER

(Read Only):

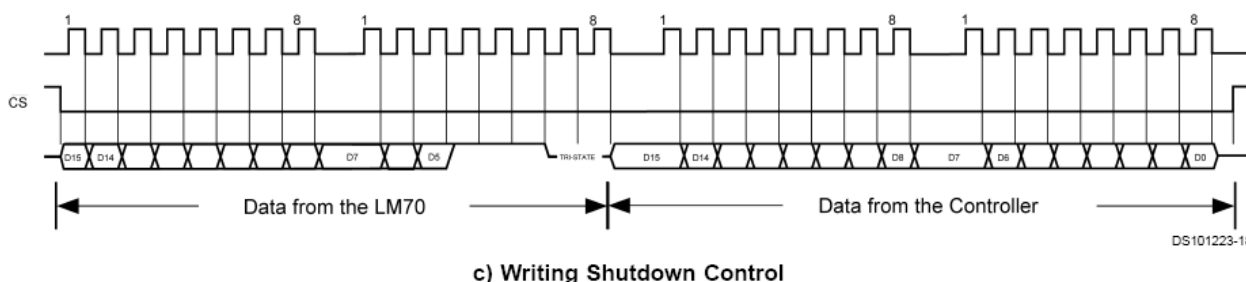
D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
MSB	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	LSB	1	1	1	X	X

D0–D1: Undefined. TRI-STATE will be output on SI/O.
D2–D4: Always set high.

D5–D15: Temperature Data. One LSB = 0.25°C. Two's complement format.

8.10.6. ECRITURE VERS LE LM70

Il y a écriture uniquement pour la configuration du mode de Shutdown. Il y a 2 cycles de lecture avant les 2 cycles d'écriture. C'est ce que nous déduisons du diagramme ci-dessous :



On remarque que le signal \overline{CS} doit être activé au début de l'action et désactivé à la fin.

1.5.1 CONFIGURATION REGISTER

(Selects shutdown or continuous conversion modes):

(Write Only):

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
X	X	X	X	X	X	X	X	Shutdown							

D0-D15 set to XX FF hex enables shutdown mode.

D0-D15 set to XX 00 hex enables continuous conversion mode.

Note: setting D0-D15 to any other values may place the LM70 into a manufacturer's test mode, upon which the LM70

will stop responding as described. These test modes are to be used for National Semiconductor production testing only. See Section 1.2 Serial Bus Interface for a complete discussion.

L'écriture s'adresse uniquement au registre de configuration. Il faut écrire XX 00 pour obtenir le mode continu. Dans l'exemple d'écriture on utilise 00 00.

8.10.6.1. EXEMPLE D'ÉCRITURE

Dans l'exemple ci-dessous, il y a écriture d'une valeur 16 bits valant 0. Pour éviter des problèmes avec le buffer de réception, il faut utiliser la fonction de lecture aussi pour écrire les 0. D'où :

```
// action de configuration
CS_LM70 = 0;
spi_read1(0xFF);
spi_read1(0xFF);
spi_read1(0); // pour écrire 0
spi_read1(0); // pour écrire 0
//Fin de transmission
CS_LM70 = 1;
```

8.10.7. OBSERVATION DES SIGNAUX DU LM70

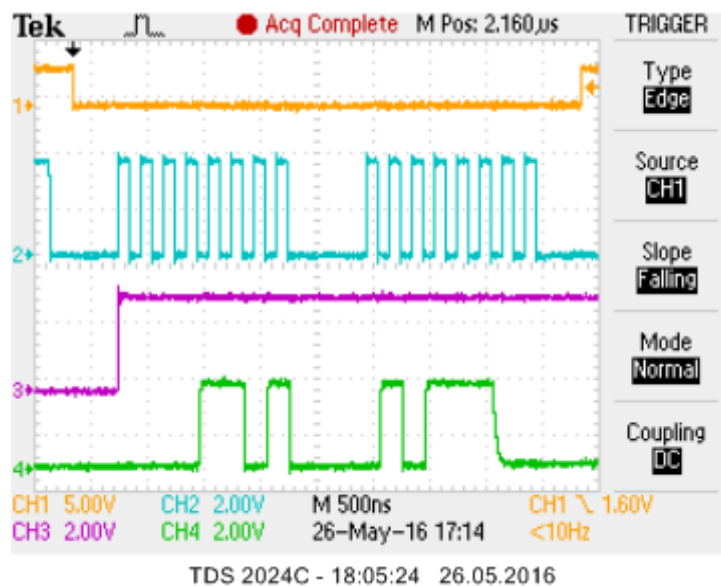
Voici la vue d'ensemble lors de la lecture du registre de température :

Canal 1 : CS_LM70
RD3
broche 78

Canal 2 : SPI-SCK
SCK1/RD10
broche 70

Canal 3 : SPI-MOSI
SDO1/RD0
broche 72

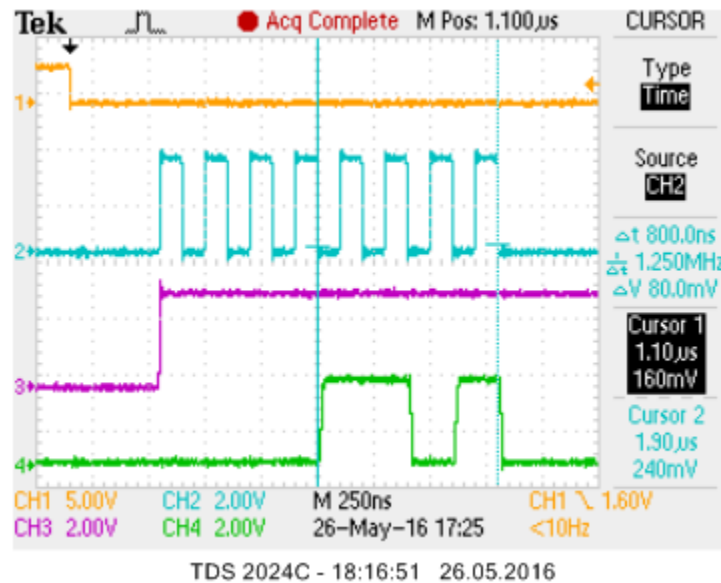
Canal 4 : SPI-MISO
SDI1/RC4
broche 9



On peut observer que le MOSI (canal 3) est à l'état haut durant l'action. Le LM70 fournit une valeur dès réception des coups d'horloge.

8.10.7.1. VUE DE DÉTAIL

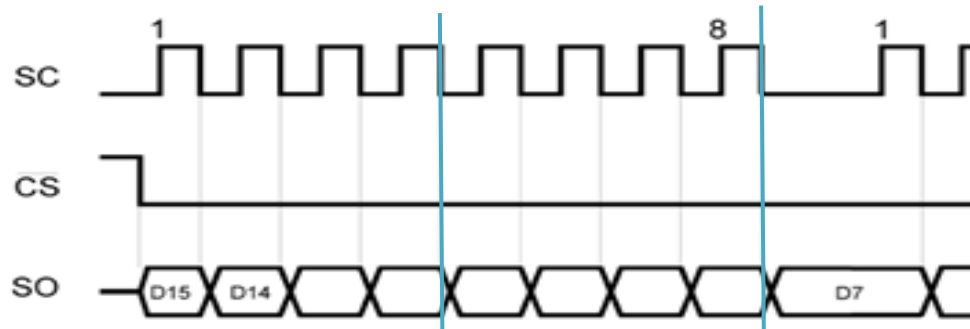
Voici la vue de détail, observation du traitement du 1^{er} octet (MSB, le plus stable).



La fréquence d'horloge est bien de 5 MHz car 4 périodes donnent 800 ns et pour 5 MHz la période est de 200 ns.

La valeur de l'octet de poids fort transmis est 0x0D, soit 0000'1101, ce que le master peut lire au flanc montants de l'horloge. Le LM70 fournit un nouveau bit lors de chaque flanc descendant de l'horloge (sauf le tout premier bit, qui est fourni dès le /CS bas). Au 1^{er} flanc descendant, on a donc le 2^{ème} bit qui est mis en sortie du LM70. Lors du dernier flanc descendant de l'horloge, le data passe déjà à l'octet suivant.

La figure datasheet nous confirme tout cela :



8.11. UTILISATION COMBINÉE DES 2 SLAVES SPI

Pour permettre d'utiliser sur le même bus SPI du LM70 et du LTC2604, il est nécessaire de reconfigurer le SPI avant chaque transaction. Cela permet donc d'utiliser des fréquences d'horloge différentes et des modes d'horloge différents.

Au niveau de l'initialisation on exécutera une fois la fonction d'initialisation du LTC2604 pour s'assurer de son Reset.

8.11.1. FONCTION LECTURE LM70 AVEC RECONFIGURATION

Lecture du registre de température du LM70, version avec reconfiguration pour partage du bus SPI avec un autre composant.

```
// Lecture du registre de température du LM70
// Version avec reconfiguration
int16_t SPI_CfgReadRawTempLM70(void)
{
    uint8_t MSB;
    uint8_t LSB;
    int16_t RawTemp;

    SPI_ConfigureLM70();

    CS_LM70 = 0;
    MSB = spi_read1(0xFF);
    LSB = spi_read1(0xFF);
    //Fin de transmission
    CS_LM70 = 1;

    RawTemp = MSB;
    RawTemp = RawTemp << 8;
    RawTemp = RawTemp | LSB;
    return RawTemp;
} // SPI_CfgReadRawTempLM70
```

8.11.2. FONCTION ÉCRITURE LTC2604 AVEC RECONFIGURATION

Cette fonction appelle la fonction de reconfiguration avant d'effectuer la transaction.

```
void SPI_CfgWriteToDac(uint8_t NoCh, uint16_t DacVal)
{
    //Déclaration des variables
    uint8_t MSB;
    uint8_t LSB;

    // Reconfiguration du SPI
    SPI_ConfigureLTC2604() ;

    //Sélection du canal
    //3 -> Set and Update, 0/1/2/3 Sélection canal A/B/C/D,
    //                                     F tous canaux
    NoCh = NoCh + 0x30;

    // Selon canal
    MSB = DacVal >> 8;
    LSB = DacVal;

    CS_DAC = 0;
    spi_writel(NoCh) ;
    spi_writel(MSB) ;
    spi_writel(LSB) ;
    // Fin de transmission
    CS_DAC = 1;

} // SPI_CfgWriteToDac
```

8.11.3. UTILISATION ET OBTENTION DES RÉSULTATS

Dans cet exemple, on transmet les valeurs au DAC dans une routine d'interruption. Pour garantir que la séquence de lecture du LM70 ne soit pas interrompue par le dialogue du DAC, on la place également dans l'interruption du DAC en utilisant un compteur.

8.11.3.1. CONTENU DE LA RÉPONSE À L'INTERRUPTION DU TIMER1

Voici le contenu de la réponse à l'interruption du Timer1.

```
// Cycle 100 us
// Envois échantillon sur DAC
// Lecture température LM70 et activation
// de l'application tous les 5000 cycles (500 ms)
void __ISR(_TIMER_1_VECTOR, ipl6AUTO)
    _IntHandlerDrvTmrInstance0(void)
{
    static int count = 0;
    static uint16_t DacVal = 0;
    static int16_t RawTemp;
    static float TempLm70;
```

```

    PLIB_INT_SourceFlagClear(INT_ID_0,INT_SOURCE_TIMER_1);
    count++;
    // DacVal = 0x815A; // pour observation signaux
    SPI_CfgWriteToDac(0, DacVal);
    DacVal += 2048;
    if (count >= 5000 ) {
        count = 0;
        delay_us(5); // pour séparation des signaux
        RawTemp = SPI_CfgReadRawTempLM70();
        LM70_ConvRawToDeg( RawTemp, &TempLm70);
        APP_UpdateTemp (RawTemp, TempLm70);
        APP_UpdateState(APP_STATE_SERVICE_TASKS);
    }
} // end ISR

```

8.11.3.2. AFFICHAGE TEMPÉRATURE DANS L'APPLICATION

L'affichage est réalisé dans l'application. Les fonctions de mise à jour appelées depuis l'interruption permettent un accès propre aux variables de l'application.

```

APP_DATA appData;
int16_t APP_RawTemp = 225;
float APP_TempLm70 = 21.7;

case APP_STATE_SERVICE_TASKS:
    BSP_LEDToggle(BSP_LED_2);

    // Affichage temperature du LM70
    lcd_gotoxy(1,3);
    printf_lcd("RawTemp = %08X", APP_RawTemp);
    lcd_gotoxy(1,4);
    printf_lcd("Temp = %6.1f", APP_TempLm70);
    appData.state = APP_STATE_WAIT;
break;

```

8.11.3.3. PROBLÈME AVEC LES VARIABLES FLOAT DANS L'INTERRUPTION

Dans l'interruption, nous avons déclaré les 2 variables pour le LM70 de la manière suivante :

```

static int16_t RawTemp;
static float TempLm70;

```

En écrivant :

```
TempLm70 = LM70_ConvRawToDeg( RawTemp);
```

Le résultat dans TempLm70 était totalement incohérent.

En modifiant en :

```
LM70_ConvRawToDeg( RawTemp, &TempLm70);
```

Le résultat est correct.

☹ La manipulation d'une variable float implique un appel à une fonction système, il semble que cela soit la cause du problème, sans pour autant l'expliquer.

En utilisant le passage par référence, ce problème disparaît.

8.11.4. REMARQUE SUR LES RECONFIGURATIONS

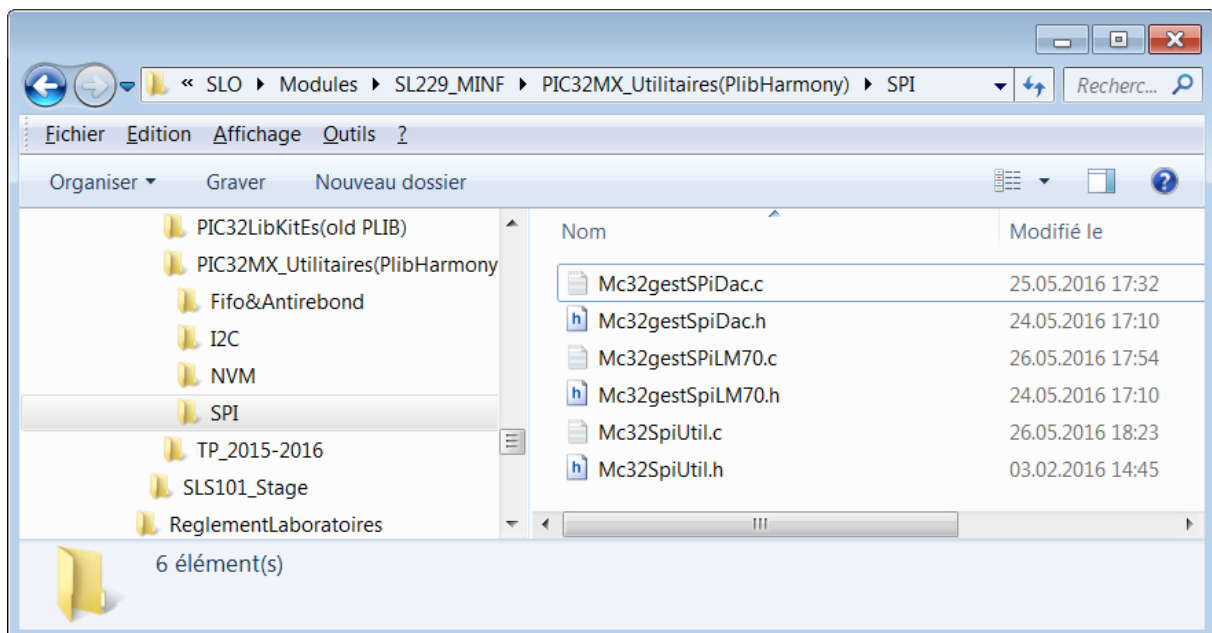
La reconfiguration avant chaque communication avec un des périphériques SPI permet de passer d'une horloge à 20 MHz pour le LTC2604 à une horloge à 5 MHz pour le LM70. Les modes d'horloge sont aussi adaptés.

La reconfiguration présente l'avantage d'effacer les erreurs, ce qui permet d'assurer une lecture propre du LM70, alors que l'on a utilisé la fonction d'écriture pour le LTC2604.

8.12. FICHIERS À DISPOSITION

Les bibliothèques permettant l'utilisation des composants SPI sont fournies dans le répertoire suivant :

...\Maitres-Eleves\SLO\Modules\SL229_MINF\PIC32MX_Utilitaires(PlibHarmony)\SPI



8.13. CONCLUSION

Ce document devrait permettre, en s'inspirant des principes utilisés, de s'adapter à la gestion par le bus SPI d'autres composants que ceux présentés.

Il s'agira à chaque fois d'adapter la configuration SPI (notamment la fréquence du clock, sa polarité et le nombre de tranches de 8 bits à lire ou à écrire). Une étude détaillée des datasheets est indispensable.

8.14. HISTORIQUE DES VERSIONS

8.14.1. VERSION 1.0 MAI 2014

Transformation du chapitre 13 (théorie PIC18) pour obtenir la 1^{ère} version de ce chapitre.

8.14.2. VERSION 1.1 MAI 2014

Remplacement des extraits de schéma PIC18 pour le LM70. Complément info sur le SPI1CON. Ajout observation des signaux du LM70.

8.14.3. VERSION 1.5 MARS 2015

Redevient un chapitre de théorie. Adaptation à la PLIB_SPI de Harmony V 1.00 et description d'une partie des fonctions.

8.14.4. VERSION 1.7 MAI 2016

Version 1.7 pour compatibilité avec l'ensemble des modules. Adaptation à la PLIB_SPI de Harmony V 1.06 et ajout de l'étude du driver SPI fourni par le MHC. Correction de la fonction spi_read et contrôle du bon fonctionnement.

8.14.5. VERSION 1.8 MARS 2017

Relecture générale par SCA.

8.14.6. VERSION 1.9 FÉVRIER 2018

Ajouts documents de référence. Corrections mineures.

8.14.7. VERSION 1.91 MARS 2019

Correction mineure fonction spi_write1.