Trabalho prático N.º 11

Objetivos

- Compreender os mecanismos básicos que envolvem a programação série usando o protocolo SPI.
- Implementar funções de comunicação no PIC32 que permitam a interação com uma memória EEPROM com interface SPI.

Introdução

A interface SPI (acrónimo de *Serial Peripheral Interface*) é uma interface de comunicação série bidirecional *full-duplex* vocacionada para ligações de pequena distância entre periféricos e microcontroladores. É uma interface de comunicação síncrona com relógio explícito do *master*, isto é, o relógio é gerado pelo *master* que o disponibiliza para todos os *slaves*. Utiliza uma arquitetura *master-slave* com ligação ponto a ponto que funciona em modo "data exchange": por cada bit que é enviado para o *slave* é recebido 1 bit. Assim, ao fim de N ciclos de relógio o *master* enviou uma palavra de N bits e recebeu, do *slave*, uma palavra também com N bits. A título de exemplo, se o *master* pretender ler do *slave* uma palavra de 16 bits, tem que transmitir uma palavra de 16 bits (que será descartada pelo *slave* se de facto se tratar apenas de um operação de leitura). A Figura 24 mostra, de forma esquemática, o funcionamento da interface SPI.

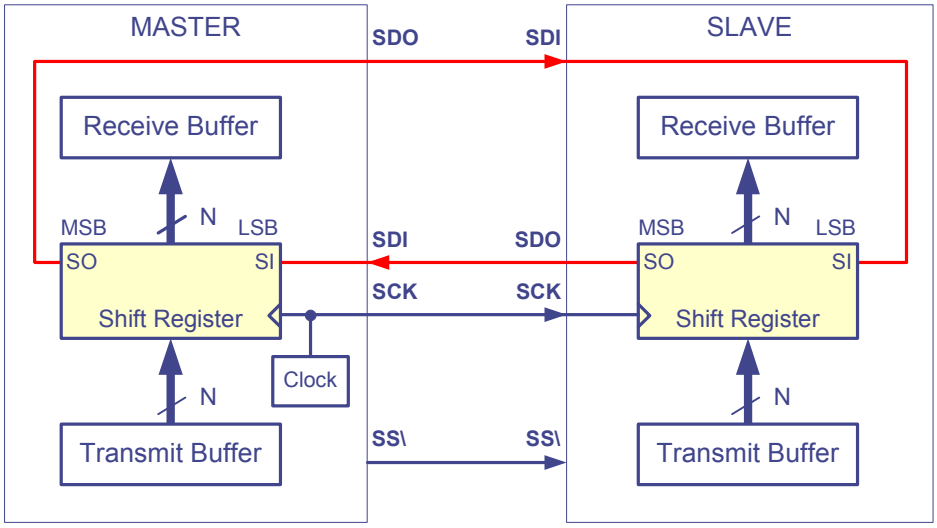


Figura 24. Esquema de princípio da comunicação série SPI.

O PIC32, na versão usada na placa DETPIC32, disponibiliza 3 módulos SPI, numerados pelo fabricante como SPI2, SPI3 e SPI4. A Figura 25 apresenta o diagrama de blocos simplificado do módulo SPI. Os elementos-base de cada um desses módulos são: *shift-register* (embora não representado na figura, existem dois *shift-registers* separados – um para receção e outro para transmissão), um FIFO de transmissão, um FIFO de recepção e um gerador de *baudrate*.

O comprimento de palavra a usar na comunicação é configurável, podendo ser 8, 16 ou 32 bits. O número de posições de cada FIFO é dependente do comprimento de palavra selecionado, sendo de 16 posições se o comprimento de palavra for 8 bits, de 8 posições se o comprimento de palavra for 16 bits e de 4 posições se o comprimento de palavra for 32 bits.

O acesso aos FIFOs de transmissão e de receção é efetuado através do registo **`SPIxBUF`**: uma escrita neste registo traduz-se no acesso indireto ao FIFO de transmissão, enquanto que a leitura desse registo traduz-se num acesso ao FIFO de receção.

O *Baudrate generator* gera o sinal de relógio que é enviado, através da linha **`SCK`**, para todos os *slaves* do sistema. Este gerador apenas é ativado quando o módulo é configurado como

master. Além disso, o sinal de relógio só é gerado quando há comunicação, estando num estado de repouso (configurável) quando o módulo não está a transmitir informação.
Após *power-on* ou *reset* todos os 3 módulos SPI estão inativos. A ativação é efetuada através do bit ON do registo **`SPIxCON`**. A ativação de um dado módulo SPI configura automaticamente os pinos correspondentes do PIC32 como entrada ou saída, consoante os casos, sobrepondo-se esta configuração à efetuada através do(s) registo(s) **`TRISx`**.

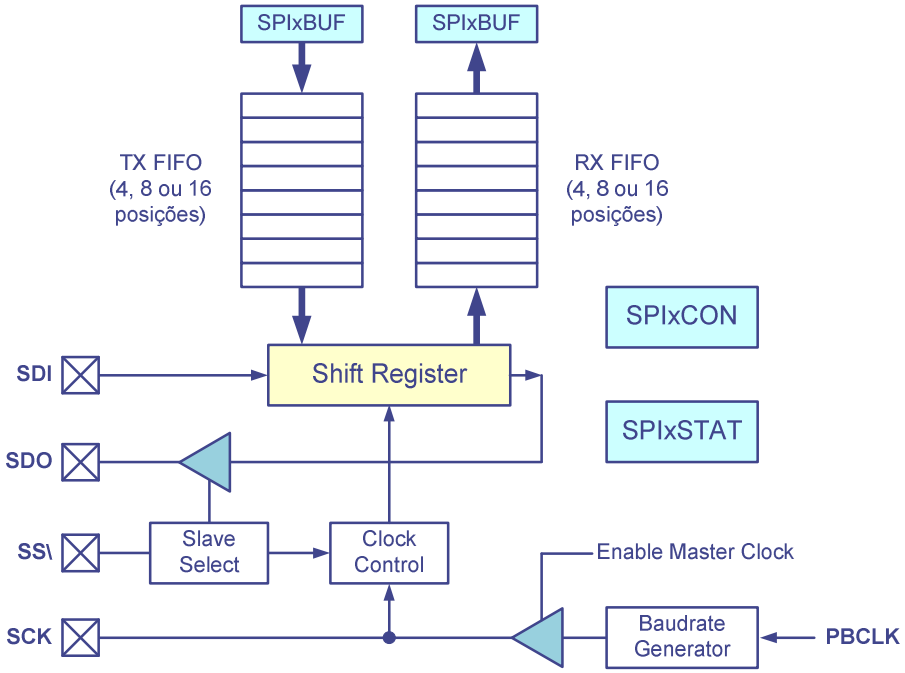


Figura 25. Diagrama de blocos simplificado do módulo SPI do PIC32.

Gerador de *baudrate*

O gerador de *baudrate* utiliza uma arquitetura semelhante à de um *timer*, em que o sinal de relógio de entrada é o *Peripheral Bus Clock* (20 MHz na placa DETPIC32). Dada a obrigatoriedade, imposta pelo funcionamento da interface SPI, de o relógio ter um *duty-cycle* de 50%, este bloco inclui, à saída do comparador, um divisor por 2.

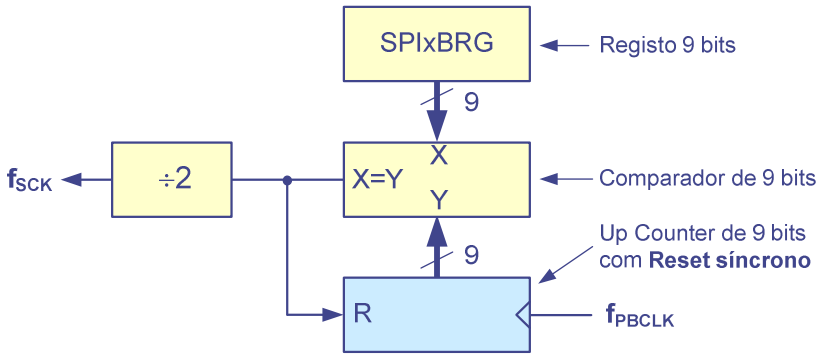


Figura 26. Diagrama de blocos do gerador de *baudrate*.

A frequência à saída deste módulo é, então,

$$f_{SCK} = f_{PBCLK} / (2 * (SPIxBRG + 1))$$

pelo que o valor (arredondado) de **`SPIxBRG`** é dado por:

$$SPIxBRG = (f_{PBCLK} + f_{SCK}) / (2 * f_{SCK}) - 1$$

em que **SPIxBRG** representa a constante armazenada no registo com o mesmo nome. De notar que o valor máximo da constante de divisão é 512 (o *timer* tem uma arquitetura de 9 bits).

Configuração do módulo SPI

A configuração do módulo SPI (como *master*) é, no essencial, efetuada de acordo com o *slave* com que se vai comunicar. Numa aplicação típica com vários *slaves* o módulo SPI é reconfigurado antes de se iniciar a comunicação com um dado *slave* tendo em conta as suas características específicas. Há, assim, um conjunto de ações de configuração gerais e um conjunto de acções de configuração que dependem das características do *slave*. As ações de configuração que dependem das características do *slave* são:

- Configurar o gerador de *baudrate*: cálculo da constante **SPIxBRG** e escrita no respetivo registo. A frequência máxima de relógio é um parâmetro intrínseco de cada *slave*.
- Configurar o nível lógico do relógio a que corresponde a situação de repouso.
- Configurar a transição ativa do relógio, isto é, em que há transmissão de dados: transição do estado ativo para o estado de repouso, ou o contrário.
- Configurar o instante em que o *master* armazena a informação recebida da linha.
- Configurar o comprimento de palavra: pode ser 8, 16 ou 32 bits.

As ações de configuração gerais, não dependentes do *slave*, são:

- Configurar o módulo como *master*.
- Ativar a utilização dos FIFOs de transmissão e receção (é possível a utilização do módulo SPI sem estes FIFOs)
- Ativar a utilização da linha *slave select* (**SS**) que permite selecionar automaticamente o *slave* durante a comunicação. Esta opção é útil quando o sistema apenas tem 1 *slave*. Se não for esse o caso, a linha de seleção tem que ser controlada por *software*, tipicamente através de um porto de saída por cada *slave*.
- Limpar o FIFO de receção.
- Limpar a *flag* de *overflow* na receção (se esta *flag* ficar activa o módulo descarta todas as palavras recebidas).
- Ativar o módulo SPI.

Todas estas configurações têm que ser efetuadas com o módulo desativado, pelo que essa deve ser a primeira ação do procedimento de configuração.

Programação com o módulo SPI

O módulo SPI realiza, no essencial, uma única operação: transmissão de uma palavra para o *slave* que estiver selecionado. Um vez que o sistema funciona em modo "data exchange", a transmissão de uma palavra de **n** bits envolve sempre a receção simultânea de uma palavra com a mesma dimensão (como referido anteriormente, a dimensão da palavra é configurável). Assim, o envio de um byte para o *slave* é feito através da cópia desse valor para o FIFO de transmissão através do registo **SPIxBUF**, podendo o valor recebido ser ignorado (a transmissão começa logo que o valor é copiado para o FIFO). Por outro lado, para a leitura de um byte de um *slave* é necessário transmitir um byte (dependendo das situações, o valor desse byte pode não ser relevante). Por cada bit enviado pelo *master* o *slave* transmite também um bit, pelo que após 8 ciclos de relógio o *master* recebe o byte. Esse byte recebido pelo *shift-register* é depois copiado para o FIFO de receção de onde pode ser lido pelo programa através do registo **SPIxBUF**.

Um aspeto importante a ter em consideração na programação com o módulo SPI é garantir que este termina todas as ações de transferência em curso, antes de passar para qualquer outra operação. O bit **`SPIBUSY`** (*SPI Activity Status bit*) do registo **`SPIxSTAT`** (ver página 23-9 do manual) fornece essa indicação.

Tendo estes aspetos em consideração, a programação com SPI é, essencialmente, orientada para as funções de interação e para o protocolo característicos de um dado dispositivo *slave*. Neste trabalho prático vai ser explorado o modo de funcionamento e programação da interface SPI do PIC32, usando como dispositivo *slave* uma memória EEPROM de 512 bytes (512x8). Na secção seguinte faz-se uma breve descrição dessa memória, cobrindo, essencialmente, a estrutura interna, o modo de operação e o protocolo de comunicação associado a cada uma das operações que é possível realizar. Esta descrição **não dispensa** a consulta do manual do fabricante que se encontra disponível no site da disciplina.

Memória EEPROM 25LC040A

Descrição geral

O circuito integrado 25LC040A da Microchip é uma memória de tecnologia não volátil EEPROM (*Electrically Erasable Programmable Read Only Memory*) com uma capacidade de 512 bytes (endereços **`0x000`** a **`0x1FF`**) e interface de comunicação série SPI. O comprimento de palavra, para qualquer operação sobre a memória, é de 8 bits.

O *shift-register* interno da interface SPI da memória recebe um novo bit da linha SI na transição ascendente do relógio e envia um novo bit para a linha SO na transição descendente seguinte (ver Figura 28). Este comportamento determina desde logo o modo como o relógio do *master* deve ser configurado, do ponto de vista da escolha das transições ativas, devendo ser compatível com o modo de funcionamento da memória. Assim, se a memória usa, como transição ativa do relógio para receção, a transição ascendente, o *master* deve usar como transição ativa para transmissão a transição descendente. O mesmo raciocínio deve ser seguido para a configuração da transição ativa de receção no *master*.

A componente de armazenamento da memória está organizada como uma matriz com 32 linhas e 16 colunas, sendo cada ponto da matriz constituído por 8 células de armazenamento de 1 bit (i.e. cada ponto da matriz corresponde a um ponto de armazenamento de 1 byte de informação). Assim, a componente de armazenamento pode ser vista como uma matriz de 32x16 bytes, sendo que cada linha dessa matriz armazena um total de 16 bytes. O número de palavras (bytes, neste caso) armazenado numa linha da matriz designa-se por página.

A funcionalidade completa da memória pode ser explorada através de 6 comandos distintos, sumariamente apresentados na Figura 27 (ver página 7 do manual da memória).

TABLE 2-1: INSTRUCTION SET

Instruction Name	Instruction Format	Description
READ	0000 $A_8$011	Read data from memory array beginning at selected address
WRITE	0000 $A_8$010	Write data to memory array beginning at selected address
WRDI	0000 x100	Reset the write enable latch (disable write operations)
WREN	0000 x110	Set the write enable latch (enable write operations)
RDSR	0000 x101	Read STATUS register
WRSR	0000 x001	Write STATUS register

Note: A_8 is the 9th address bit, which is used to address the entire 512 byte array.

Figura 27. Comandos de interação com a memória EEPROM.

Tal como representado na figura anterior, os 3 bits menos significativos do *instruction format* definem a operação a realizar. No caso de a operação ser uma leitura ou uma escrita o bit 3 (4º

bit menos significativo) é usado para enviar o bit mais significativo do endereço da memória (bit A_8).

Sequência de leitura de uma posição de memória

O protocolo com a descrição da sequência de operações a realizar para desencadear a leitura de uma posição de memória está representado, sob a forma de um diagrama temporal, na Figura 28 (ver também página 7 do manual). Assim, da análise dessa figura pode verificar-se que, em primeiro lugar, é enviado o byte de comando (**`READ`**, ver Figura 27) que inclui o bit mais significativo do endereço (A_8), seguido do byte representativo dos 8 bits menos significativos do endereço. Para receber o valor armazenado na correspondente posição de memória o *master* tem ainda que enviar um byte adicional, cujo valor é irrelevante e que será ignorado pela memória.

FIGURE 2-1: READ SEQUENCE

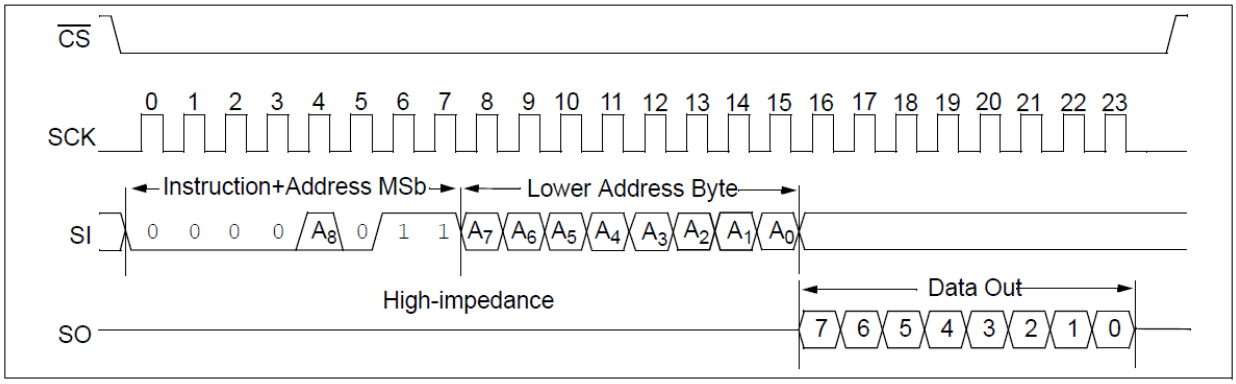


Figura 28. Sequência de leitura de uma posição de memória.

Uma operação de leitura é ignorada pela memória se ainda estiverem a decorrer operações internas relacionadas com uma escrita. De notar que o fim efetivo de uma operação de escrita não coincide com o fim das ações de comunicação. Ou seja, após ter terminado o processo de comunicação, a memória necessita de tempo adicional para consumar a operação. Esse tempo é destinado a: 1) fazer o apagamento da posição onde se pretende efetuar a escrita e 2) efetuar a escrita do byte recebido.

A memória disponibiliza um registo de 8 bits com as funções de controlo e de *status*, designado pelo fabricante por registo **`STATUS`**, que permite, entre outras coisas obter informação sobre a existência de um processo de escrita em curso. A Figura 29 representa a estrutura desse registo, onde se pode verificar que apenas os 4 bits menos significativos têm informação útil (ver também página 10 do manual).

TABLE 2-2: STATUS REGISTER

7	6	5	4	3	2	1	0
–	–	–	–	W/R	W/R	R	R
X	X	X	X	BP1	BP0	WEL	WIP
W/R = writable/readable. R = read-only.							

Figura 29. Estrutura do registo STATUS da memória.

Em particular, o bit menos significativo (**`WIP`**, *write in progress*) indica, quando a 1, que a memória está ocupada num operação de escrita, ou seja, que a última operação de escrita ainda está em curso. Assim, uma operação de leitura ou escrita deve sempre ser precedida da verificação (por *polling*) do estado deste bit através da leitura do registo **`STATUS`** (ver Figura 27). Faz-se, mais à frente, a descrição da sequência para efetuar a leitura desse registo.

Sequência de escrita de uma posição de memória

A Figura 30 apresenta, sob a forma de um diagrama temporal a sequência de operações para efetuar a escrita de uma posição de memória (ver também página 8 do manual). Tal como para a leitura, em primeiro lugar é enviado o byte de comando (**WRITE**, ver Figura 27) que inclui o bit mais significativo do endereço, seguido do byte menos significativo do endereço e, finalmente, do byte a escrever.

FIGURE 2-2: BYTE WRITE SEQUENCE

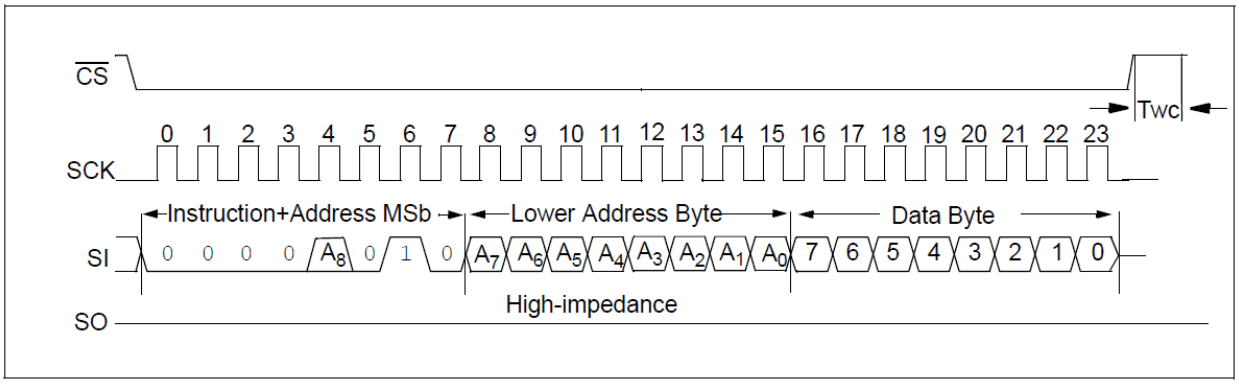


Figura 30. Sequência de escrita de uma posição de memória.

De modo a evitar a alteração acidental da informação armazenada, a memória EEPROM está sempre protegida contra operações de escrita. Ou seja, antes de efetuar uma sequência de escrita é sempre necessário desproteger a memória. A desproteção é efetuada através do envio do comando de ativação da escrita (**WREN**, ver Figura 27) que ativa o bit **WEL** (*write enable latch*) do registo **STATUS**.

Após o fim de uma operação de escrita (uma operação de escrita termina, do ponto de vista da comunicação, quando o sinal **CS** é desativado) a memória regressa ao estado de escrita protegida, pelo que, para cada operação de escrita é sempre necessário efetuar, em primeiro lugar, a desproteção.

Sequência de leitura do registo de STATUS

A Figura 31 descreve a sequência para ler o registo de **STATUS** da memória (ver também página 10 do manual). A sequência começa com o envio do comando respetivo (**RDSR**, ver Figura 27), seguido do envio de um byte, cujo valor é irrelevante, para permitir à EEPROM enviar o conteúdo do registo.

FIGURE 2-6: READ STATUS REGISTER TIMING SEQUENCE (RDSR)

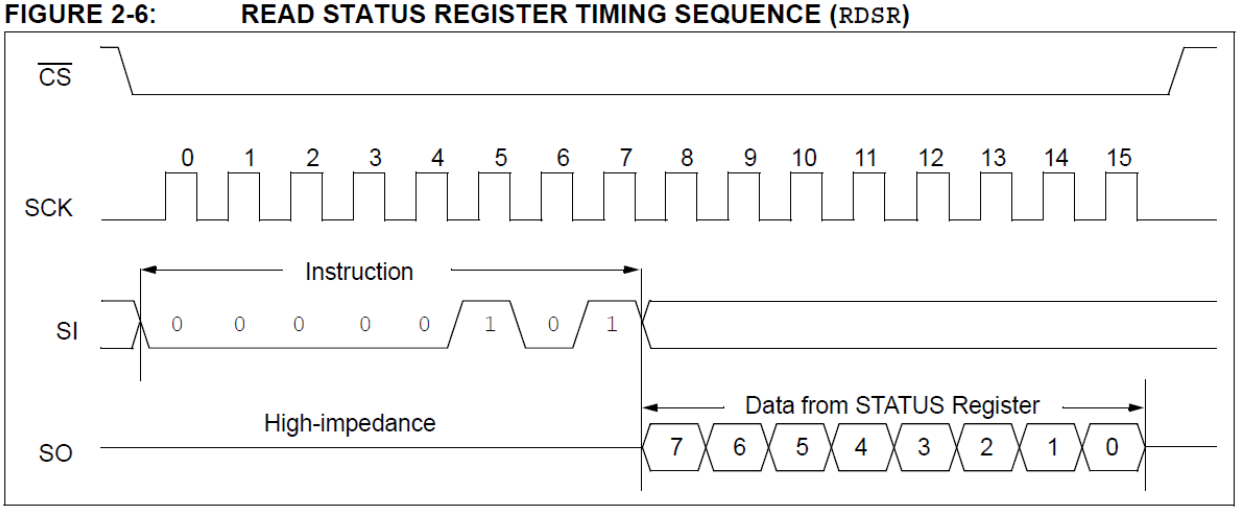


Figura 31. Sequência de leitura do registo de STATUS

O registo **STATUS** pode ser lido em qualquer momento independentemente de estarem ou não a ser realizadas outras operações internas.

Escrita no registo de STATUS

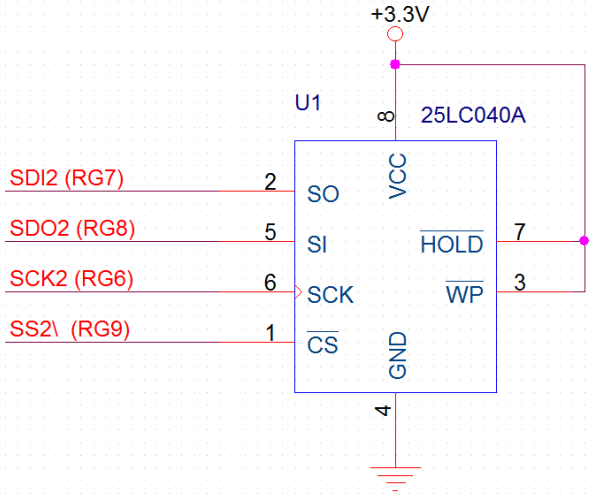
O registo de **STATUS** pode ser escrito usando um protocolo em que são enviados dois bytes em sequência: um byte com o comando **WRSR** e outro byte com o valor a escrever (ver página 11 do manual da EEPROM).

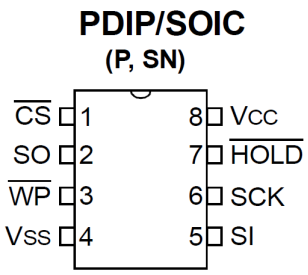
O fabricante disponibiliza também uma sequência específica para a desproteção da escrita (ativação do bit **WEL** do registo de **STATUS**). Esta sequência envolve apenas a transmissão de 1 byte correspondente ao código do comando (**WREN**, ver Figura 27).

Trabalho a realizar

Parte I

1. Monte, na placa branca, a memória EEPROM 25LC040A, de acordo com a figura seguinte. Os sinais **SDI2** (**RG7**), **SDO2** (**RG8**), **SCK2** (**RG6**) e **SS2** (**RG9**) correspondem à ligação ao módulo SPI número 2 do PIC32.





2. Escreva uma função para configurar o gerador de *baudrate*:

```
void spi2_setClock(unsigned int clock_freq)
{
   // Write SPI2BRG register(see introduction for details)
}
```

3. Implemente a função para inicializar o módulo SPI (consulte o manual do SPI).

```
void spi2_init(void)
{
   volatile char trash;
   // Disable SPI2 module
   // Configure clock idle state as logic level 0
   // Configure the clock active transition: from active
   //   state to idle state
   // Configure SPI data input sample phase bit (middle of data
   //   output time)
   // Configure word length (8 bits)
   // Enable Enhanced buffer mode (this allows the usage of FIFOs RX,TX)
   // Enable slave select support (Master Mode Slave Select)
   // Enable master mode
   // Clear RX FIFO:
   while(SPI2STATbits.SPIRBE == 0)    // while RX FIFO not empty read
      trash = SPI2BUF;                // FIFO and discard read character
   // Clear overflow error flag
   // Enable SPI2 module
}
```

4. As funções que se vão implementar de seguida são dependentes do modo de funcionamento da EEPROM já explicado na introdução deste trabalho prático. Vamos começar por implementar a função que permite a leitura do registo de **STATUS**, uma vez que essa função é necessária para todas as restantes:

```
char eeprom_readStatus(void)
{
   volatile char trash;

   // Clear RX FIFO
   // Clear overflow error flag bit
   SPI2BUF = RDSR;   // Send RDSR command
   SPI2BUF = 0;      // Send anything so that EEPROM clocks data into SO
   while(SPI2STATbits.SPIBUSY);  // wait while SPI module is working
   trash = SPI2BUF;  // First char received is garbage (received while
                     //   sending command)
   return SPI2BUF;   // Second received character is the STATUS value
}
```

   De modo a tornar o programa mais legível, deverá definir os códigos dos comandos que estão definidos no manual do fabricante. Por exemplo:

```
#define RDSR   0x05
#define WRITE  0x02
```

5. Teste as funções que já escreveu. Para isso escreva o programa principal que chame as funções de inicialização e que, em ciclo infinito, leia o registo de **STATUS** e imprima o seu valor usando *system calls* (apenas os 4 bits menos significativos do byte recebido têm informação útil). Se as funções funcionarem adequadamente, o valor recebido nos 4 bits menos significativos deverá ser 0 (**BP1**, **BP0**, **WEL** e **WIP** todos a 0),

```
void main(void)
{
   spi2_init();
   spi2_setClock(EEPROM_CLOCK);
   for(;;)
   {
      // Call "eeprom_readStatus() function
      // Print read value
   }
}
```

   O sinal de relógio para a EEPROM pode, de acordo com o fabricante, ter uma frequência máxima de 5 MHz. Não é, contudo, aconselhável a utilização de uma frequência tão elevada numa montagem em placa branca, pelo que se recomenda a utilização de uma frequência mais baixa, por exemplo 500 KHz (a frequência mais pequena que poderá utilizar é 20 kHz. Porquê?).

6. Escreva agora a função para implementar a sequência simplificada de escrita no registo **STATUS** (apenas para os comandos **WREN** e **WRDIS**):

```
void eeprom_writeStatusCommand(char command)
{
   while( eeprom_readStatus() & 0x01 );     // Wait while WIP is true
                                            //   (write in progress)
   // Copy "command" value to SPI2BUF  (TX FIFO)
   // Wait while SPI module is working (SPIBUSY set)
}
```

7. Teste a função anterior acrescentando ao programa que escreveu no exercício 5 a chamada da função **eeprom_writeStatusCommand()** com o argumento **WREN**. O valor

retornado pela função de leitura do registo de **STATUS** deve agora ser 2 (isto é, a proteção de escrita está desativada).

8. Escreva a função para escrita de um byte numa posição de memória. A função deve ter como parâmetros de entrada o endereço de escrita e o valor a escrever:

```
void eeprom_writeData(int address, char value)
{
   // Apply a mask to limit address to 9 bits
   // Read STATUS and wait while WIP is true (write in progress)
   // Enable write operations (activate WEL bit in STATUS register, using
         eeprom_writeStatusCommand() function )
   // Copy WRITE command and A8 address bit to the TX FIFO:
   SPI2BUF = WRITE | ((address & 0x100) >> 5);
   // Copy address (8 LSBits) to the TX FIFO
   // Copy "value" to the TX FIFO
   // Wait while SPI module is working (SPIBUSY)
}
```

9. Escreva, finalmente, a função para leitura de um byte de uma posição de memória. A função deve ter como parâmetro de entrada o endereço e deve retornar o valor lido:

```
char eeprom_readData(int address)
{
   volatile char trash;
   // Clear RX FIFO
   // Clear overflow error flag bit
   // Apply a mask to limit address to 9 bits
   // Read STATUS and wait while WIP is true (write in progress)
   // Copy READ command and A8 address bit to the TX FIFO
   // Copy address (8 LSBits) to the TX FIFO
   // Copy any value (e.g. 0x00) to the TX FIFO
   // Wait while SPI module is working (SPIBUSY)
   // Read and discard 2 characters from RX FIFO (use "trash" variable)
   // Read RX FIFO and return the corresponding value
}
```

10. Para o teste das funções que escreveu nos exercícios anteriores escreva a função **main()**, de modo a realizar, em ciclo infinito, as seguintes operações (utilize system calls para a interação com o utilizador):
    - Lê um caracter
    - Se for 'R' (read) lê um endereço (**addr**), e imprime o valor lido da memória.
    - Se for 'W' (write) lê um endereço e um valor (**addr**, **val**), e escreve na EEPROM no endereço **addr** o valor **val**.

```
void main(void)
{
   for(;;)
   {
      // Read character
      // If character is 'R' then ...
      (...)
   }
}
```

11. Deslige a alimentação durante, pelo menos, 30 segundos, e repita o exercício anterior efetuando apenas leituras dos endereços de memória em que escreveu.
12. Altere a função que implementou no exercício anterior de modo a que, após a leitura do endereço e do valor, o programa escreva sucessivamente nas 16 posições de memória seguintes o valor anterior incrementado de 1. De seguida o programa deve ler os 16 valores da EEPROM e imprimi-los em hexadecimal (formato: "endereço inicial: valor valor ...).

Parte II

1. O objetivo deste exercício é observar os sinais da interface SPI com o osciloscópio. Para isso retome o código que escreveu no exercício 10 e coloque a chamada à função de leitura da EEPROM em ciclo infinito. Ligue as duas pontas de prova do osciloscópio, uma à linha **CS** e outra à linha **SI** da EEPROM e selecione como entrada de *trigger* do osciloscópio o canal que ligou à linha **CS**. Observe o sinal na linha **SI** e identifique os valores transferidos. Repita o procedimento colocando a ponta de prova na linha **SO** da EEPROM.
2. À semelhança do que já fez no trabalho prático anterior, pretende-se agora organizar o código produzido de forma a que ele possa facilmente ser integrado em outras aplicações. Para isso escreva o ficheiro "**eeprom.h**" com os protótipos de todas as funções e os símbolos públicos. Construa também o ficheiro "**eeprom.c**" com o código das funções.

```
// eeprom.h
#ifndef EEPROM_H
#define EEPROM_H
// Declare symbols here (READ, WRITE, ...)
(...)
// Declare function prototypes here
(...)
#endif
```

3. Retome o código que escreveu no último exercício do trabalho prático n.º 10. Faça as alterações que permitam o registo de temperatura em EEPROM, bem como a sua visualização, de acordo com a seguinte especificação:
   1) os valores de temperatura deverão ser armazenados a partir do endereço **0x002** da EEPROM; a posição de memória **0x000** deverá conter o número de valores de temperatura armazenados;
   2) quando for recebido da linha série o caracter 'R' (*reset*) o sistema deve colocar a zero o número de temperaturas armazenado na memória (endereço **0x000** da EEPROM);
   3) quando for recebido da linha série o caracter 'L' (*log*) o sistema deve iniciar o registo, na EEPROM (a partir do endereço **0x002**), do valor instantâneo da temperatura; deverão ser armazenadas 4 amostras por minuto (i.e. uma amostra a cada 15s), até um máximo de 64; a posição de memória **0x000** deve ser incrementada a cada nova escrita na EEPROM (ou seja, a posição de memória **0x000** deverá conter o número de valores de temperatura válidos armazenados em memória);
   4) quando for recebido o caracter 'S' (*show*) o sistema deve parar o registo de temperaturas e enviar para a porta série os valores de temperatura entretanto armazenados;

   Note que o comando 'R' terá que ser efetuado, pelo menos uma vez, no início, para se colocar a posição de memória **0x000** a zero.
4. Altere o código do exercício anterior de modo a efetuar o registo contínuo de temperatura, isto é, sem a limitação dos 64 valores. Para isso deverá implementar na memória um *buffer* circular de 64 posições, a partir do endereço **0x002**. A posição de memória **0x000** deverá conter o número de valores armazenados (com um máximo de 64) e a posição **0x001** deverá conter o endereço onde a próxima escrita deverá ser efetuada.

Elementos de apoio

- Slides das aulas teóricas.
- 25LC040A - 4K SPI Bus Serial EEPROM (disponível no site da disciplina).
- PIC32 Family Reference Manual, Section 23 – SPI.