14. SPI

Neste capítulo, são apresentados a interface serial periférica do ATmega328 (SPI – *Serial Peripheral Interface*) e dois exemplos de dispositivos com SPI: um sensor de temperatura, o TC72, e um cartão de memória *flash*, o SD *card*.

A interface SPI é utilizada por uma infinidade de circuitos integrados, como por exemplo: conversores DAs e ADs, memórias *flash* e EEPROM, relógios de tempo real, sensores de temperatura e pressão, potenciômetros digitais, LCDs e telas sensíveis ao toque. No ATmega, além da funcionalidade usual, a SPI também é utilizada para a gravação da memória do microcontrolador (gravação *in-system*) como apresentado no capítulo 23.

A SPI é um padrão de comunicação de dados serial criado pela Motorola que opera no modo *full duplex*. Não se limita a palavras de 8 bits, assim podem ser enviadas mensagens de qualquer tamanho com conteúdo e finalidade arbitrários. A SPI é um protocolo muito simples com taxas de operação superiores a 20MHz. É um protocolo serial síncrono proposto para ser usado como um padrão para estabelecer a comunicação entre microcontroladores e periféricos. Os dispositivos no protocolo SPI são classificados como mestre ou escravos e são empregadas 4 vias para a comunicação:

- MOSI (*Master Out – Slave In*): saída de dados do mestre, entrada no escravo.
- MISO (*Master In – Slave Out*): entrada de dados no Mestre, saída do escravo.
- SCK: *clock* serial, gerado pelo mestre
- $\overline{SS}$ (*Slave Select*) - seleção do escravo, ativo em zero.

Uma transferência de dados SPI é iniciada pelo dispositivo mestre. Ele é o responsável por gerar o sinal SCK para a transferência de dados.

Em um sistema SPI, apenas um dispositivo é configurado como mestre, os outros dispositivos são configurados como escravos. Na fig. 14.1, é apresentada uma conexão onde existe apenas um dispositivo escravo.

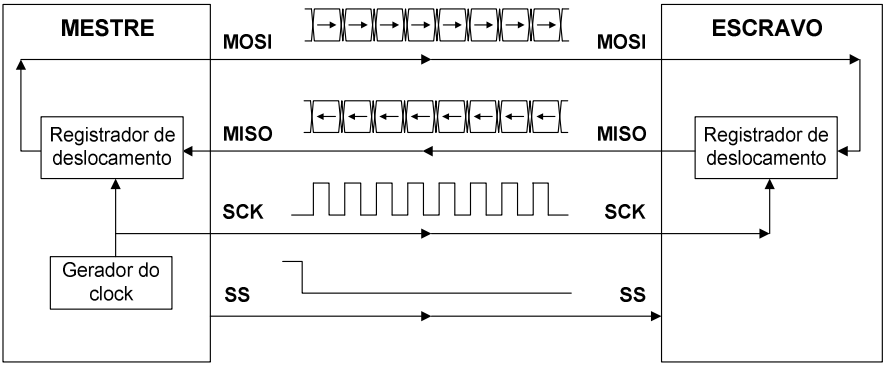

Fig. 14.1 – Conexão SPI entre um mestre e um único escravo.

Em sistemas com mais de um dispositivo escravo são possíveis dois tipos de conexão:

- Seleção individual de cada escravo: o mestre seleciona individualmente o escravo com que deseja comunicar-se, existirá um pino $\overline{SS}$ dedicado para cada escravo. Desta forma, um dispositivo escravo não interfere no outro (fig. 14.2).
- Seleção coletiva dos escravos: os escravos são conectados em um grande anel. A saída de um escravo é conectada a entrada de outro e o circuito de dados é fechado com o mestre. O mestre seleciona todos os escravos ao mesmo tempo através de um pino $\overline{SS}$ comum (fig. 14.3).

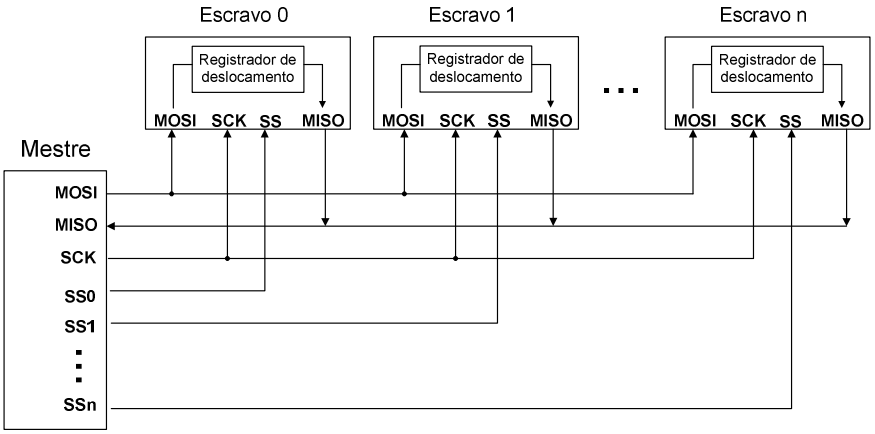

Fig. 14.2 – Conexão entre o mestre e vários escravos: seleção individual.

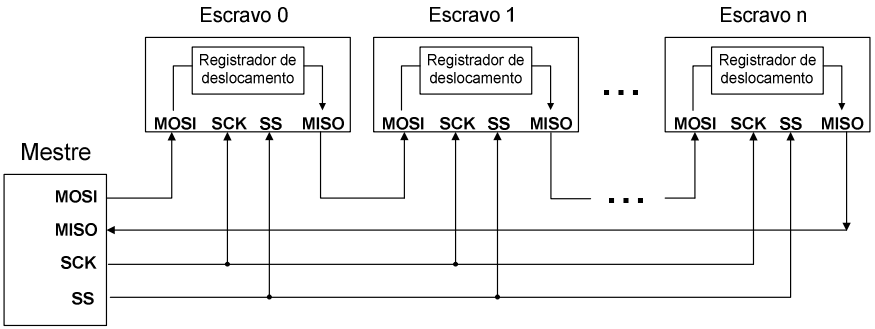

Fig. 14.3 – Conexão entre o mestre e vários escravos: seleção coletiva com ligação em anel.

As principais vantagens da SPI são:

- Protocolo simples, fácil de programar e, caso o microcontrolador não possua o módulo SPI, é fácil implementá-lo através da programação.
- Interface simples, sem pinos bidirecionais.
- Frequência de comunicação elevada, maior que 20 MHz; depende apenas das características dos dispositivos empregados.

- Dados trafegam em modo *full-duplex* (envia e recebe ao mesmo tempo).

Suas principais desvantagens são:

- A forma de transferência e amostragem de dados varia de acordo com o dispositivo.
- O número de bytes transmitidos para a realização da comunicação pode variar de acordo com os dispositivos empregados (protocolo da comunicação).
- Em sistemas com múltiplos escravos, ou a transmissão torna-se menos eficiente, caso da comunicação em anel, ou são empregados muitos pinos para a seleção dos escravos (3 + n, onde n representa o número de escravos).

14.1 SPI DO ATMEGA328

A Interface Serial Periférica no ATmega328 pode trabalhar com até metade da velocidade de trabalho da CPU, podendo chegar a 10 MHz, e possui as seguintes características:

- *Full-duplex* (envia e recebe dados ao mesmo tempo), transferência síncrona de dados com 3 vias.
- Operação mestre ou escravo.
- Escolha do bit a ser transferido primeiro: bit mais significativo (MSB) ou menos significativo (LSB).
- Várias taxas programáveis de *clock*.
- Sinalizador de interrupção ao final da transmissão.
- Sinalizador para proteção de colisão de escrita.
- *Wake-up* do modo *Idle*.
- Velocidade dupla no modo *Master* ($f_{osc}/2$).

O sistema é composto por dois registradores de deslocamento e um gerador mestre de *clock*. O mestre SPI inicializa o ciclo de comunicação quando coloca o pino $\overline{SS}$ (*Slave Select*) do escravo desejado em nível baixo.

Mestre e escravo preparam o dado a ser enviado nos seus respectivos registradores de deslocamento e o mestre gera os pulsos necessários de *clock* no pino SCK para a troca de dados (ao mesmo tempo um bit é enviado e outro é recebido). Os dados são sempre deslocados do mestre para o escravo no pino MOSI (*Master Out – Slave In*) e do escravo para o mestre no pino MISO (*Master In – Slave Out*). Após cada pacote de dados, o mestre sincroniza o escravo colocando o pino $\overline{SS}$ em nível alto.

Quando configurada como mestre, a interface SPI não tem controle automático sobre o pino $\overline{SS}$. Isso deve ser feito por software, antes da comunicação começar. Quando se escreve um byte no registrador de dados da SPI, a geração do *clock* inicia automaticamente e o hardware envia os oito bits para o escravo. Após esse envio, o *clock* da SPI para, ativando o aviso de final de transmissão (*flag* SPIF). Se o bit de interrupção da SPI estiver habilitado (SPIE) no registrador SPCR, uma interrupção será gerada. O mestre pode continuar a enviar o próximo byte, escrevendo-o no registrador SPDR, ou sinalizar o final da transmissão ($\overline{SS}$ em nível alto). O último byte de chegada é mantido no registrador de armazenagem (*buffer*).

O sistema possui um registrador de transmissão e um duplo de recepção. Isso significa que o byte a ser transmitido não pode ser escrito no registrador de dados da SPI antes do final da transmissão do byte. Quando um dado é recebido, ele deve ser lido do registrador SPI antes do próximo dado ser completamente recebido, caso contrário, o primeiro dado é perdido.

No modo escravo, para garantir a correta amostragem do sinal de *clock*, os períodos mínimo e máximo do *clock* recebido devem ser maiores que 2 ciclos de *clock* da CPU.

Quando o modo SPI é habilitado, a direção dos pinos MOSI, MISO, SCK e $\overline{SS}$ é ajustada conforme tab. 14.1.

Tab. 14.1 – Direção dos pinos para a SPI quando habilitada.

Pino	Direção, SPI Mestre	Direção, SPI Escravo
MOSI	definido pelo usuário	entrada
MISO	entrada	definido pelo usuário
SCK	definido pelo usuário	entrada
$\overline{SS}$	definido pelo usuário	entrada

O código a seguir mostra funções para inicializar a SPI como mestre e executar uma transmissão simples.

```
//========================================================================== //
//Funções para inicializar a SPI no modo Mestre e transmitir um dado        //
//========================================================================== //
void SPI_Mestre_Inic()
{
    DDRB = (1<<PB5)|(1<<PB3);//ajusta MOSI e SCK como saída, demais
                             //pinos como entrada
    SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0);//habilita SPI, Mestre,
                                        //taxa de clock ckl/16
}
//---------------------------------------------------------------------------
void SPI_Mestre_Transmit(char dado)
{
    SPDR = dado;               //inicia a transmissão
    while(!(SPSR & (1<<SPIF)));//espera a transmissão ser completada
}
//===========================================================================
```

O código a seguir mostra funções para inicializar a SPI como escravo e executar uma recepção simples.

```
//========================================================================== //
//    Funções para inicializar a SPI no modo Escravo e receber um dado      //
//========================================================================== //
void SPI_Escravo_Inic( )
{
    DDRB = (1<<PB4);//ajusta o pino MISO como saída, demais pinos como entrada
    SPCR = (1<<SPE);//habilita SPI
}
//---------------------------------------------------------------------------
char SPI_Escravo_Recebe( )
{
    while(!(SPSR & (1<<SPIF)));//espera a recepção estar completa
    return SPDR;               //retorna o registrador de dados
}
//===========================================================================
```

SPCR – SPI *Control Register*

Bit		7	6	5	4	3	2	1	0
	SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
Lê/Escr.		L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E
Valor Inicial		0	0	0	0	0	0	0	0

Bit 7 – SPIE – SPI *Interrupt Enable*
Quando este bit estiver em 1, a interrupção da SPI é executada se o bit SPIF do registrador SPSR também estiver em 1. O bit I do registrador SREG deve estar habilitado.

Bit 6 – SPE – SPI *Enable*
Bit para habilitar a SPI.

Bit 5 – DORD – *Data Order*
Em 1, transmite-se primeiro o LSB, em 0 transmite o MSB.

Bit 4 – MSTR – *Master/Slave Select*
Em 1, seleciona o modo mestre; caso contrário, o modo escravo. Se o pino $\overline{SS}$ é configurado como entrada e estiver em nível zero enquanto MSTR estiver ativo, MSTR será limpo e o bit SPIF do SPSR será ativo. Então, será necessário ativar MSTR para reabilitar o modo mestre.

Bit 3 – CPOL – *Clock Polarity*
Este bit em 1 mantém o pino SCK (quando inativo) em nível alto; em zero, mantém o pino em nível baixo.

Bit 2 – CPHA – *Clock Phase*
Este bit determina se o dado será coletado na subida ou descida do sinal de *clock*.

Bits 1:0 – SPR1:0 – SPI *Clock Rate Select* 1 e 0
Estes dois bits controlam a taxa do sinal SCK quando o microcontrolador é configurado como mestre; sem efeito quando configurado como escravo (tab. 14.2).

Tab. 14.2 – Seleção da frequência de operação para o modo mestre (f_{osc} = freq. de operação da CPU).

SPI2X	SPR1	SPR0	Frequência do SCK
0	0	0	$f_{osc}/4$
0	0	1	$f_{osc}/16$
0	1	0	$f_{osc}/64$
0	1	1	$f_{osc}/128$
1	0	0	$f_{osc}/2$
1	0	1	$f_{osc}/8$
1	1	0	$f_{osc}/32$
1	1	1	$f_{osc}/64$

SPSR – SPI *Status Register*

Bit		7	6	5	4	3	2	1	0
	SPSR	SPIF	WCOL	-	-	-	-	-	SPI2X
Lê/Escreve		L	L	L	L	L	L	L	L/E
Valor Inicial		0	0	0	0	0	0	0	0

Bit 7 – SPIF – SPI *Interrupt Flag*
Colocado em 1 quando uma transferência serial for completada.

Bit 6 – WCOL – *Write COLision Flag*
É ativo se o registrador SPDR é escrito durante uma transmissão.

Bit 0 – SPI2X – *Double* SPI *Speed Bit*
Em 1 duplica a velocidade de transmissão quando no modo mestre. No modo escravo, a SPI trabalha garantidamente com $f_{osc}/4$.

SPDR – SPI *Data Register*

Bit		7	6	5	4	3	2	1	0
	SPDR	MSB							LSB
Lê/Escreve		L/E	L/E	L/E	L/E	L/E	L/E	L/E	L/E
Valor Inicial indefinido		X	X	X	X	X	X	X	X

Escrever neste registrador inicia uma transmissão. Ao lê-lo, carrega-se o conteúdo do *buffer* do registrador de entrada.

Modos de operação

Existem 4 combinações de fase e polaridade de *clock* com respeito aos dados seriais, determinadas pelos bits de controle CPHA e CPOL, conforme tab. 14.3 e figs. 14.4-5.

Tab. 14.3 – Funcionalidade dos bits CPOL e CPHA.

Configuração	Borda do SCK	Borda do SCK	MODO SPI
CPOL = 0, CPHA = 0	amostragem (subida)	ajuste (descida)	0
CPOL = 0, CPHA = 1	ajuste (subida)	amostragem (descida)	1
CPOL = 1, CPHA = 0	amostragem (descida)	ajuste (subida)	2
CPOL = 1, CPHA = 1	ajuste (descida)	amostragem (subida)	3

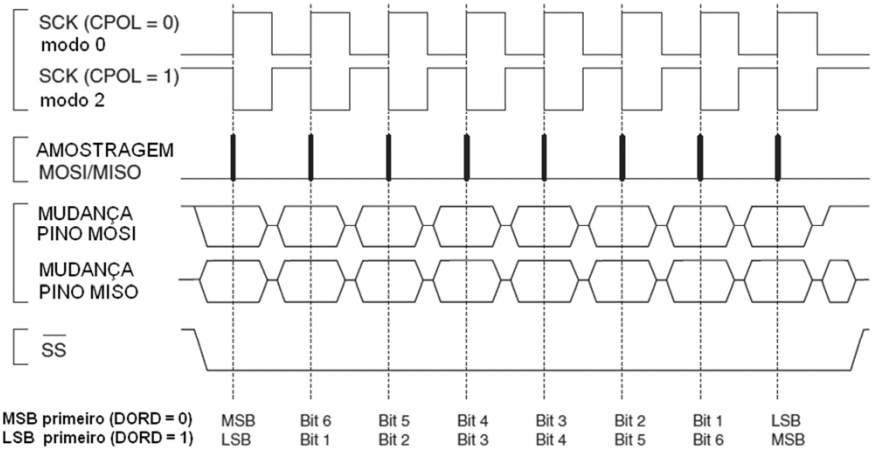

Fig. 14.4 – Formato da transferência SPI com CPHA=0.

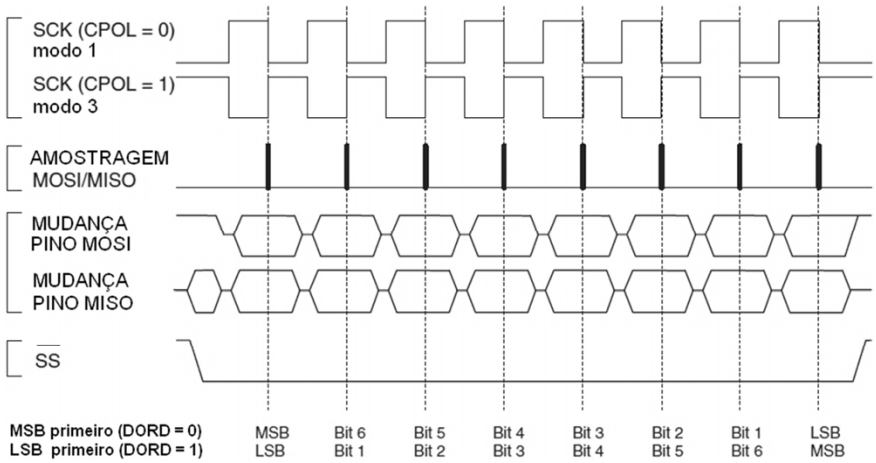

Fig. 14.5 – Formato da transferência SPI com CPHA=1.

14.2 SENSOR DE TEMPERATURA TC72

O circuito integrado TC72 é um termômetro digital com interface SPI. É capaz de medir temperaturas entre -55 ºC e +125 ºC. Suas principais características são:

- 10 bits de resolução (0,25 ºC/bit);
- precisão de ± 3 ºC de -55 ºC a +125 ºC ou ± 2 ºC de -40 ºC a +85 ºC;
- tensão de alimentação entre 2,65 V e 5,5 V;
- tempo de conversão de 150 ms;
- modo de conversão contínuo ou única conversão;
- frequência máxima de trabalho de 7,5 MHz;
- consumo de 150 μA no modo de conversão contínua;
- encapsulamento de 8 pinos MSOP ou DFN (3 × 3 mm).

A temperatura é representada por um conjunto de 10 bits em complemento de dois com uma resolução de 0,25 ºC por bit menos significativo. É escalonada de -128 ºC a +127 ºC e armazenada em dois registradores de 8 bits, cujo bit mais significativo indica se a temperatura é negativa ou positiva. Na tab. 14.4, são apresentados exemplos para valores convertidos e, nas tabs. 14.6-7, os registradores para o trabalho e o significado dos seus bits.

Tab. 14.4 – Formato para alguns valores de temperatura.

Temperatura	Valor binário MSByte*/LSByte**
+125 ºC	0111 1101 / 0000 0000
+25 ºC	0001 1001 / 0000 0000
+0,5 ºC	0000 0000 / 1000 0000
+0,25 ºC	0000 0000 / 0100 0000
0 ºC	0000 0000 / 0000 0000
-0,25 ºC	1111 1111 / 1100 0000
-25 ºC	1110 0111 / 0000 0000
-55 ºC	1100 1001 / 0000 0000

* MSByte = Byte Mais Significativo.
**LSByte = Byte Menos Significativo.

Tab. 14.5 – Registradores do TC72 (nos registradores de temperatura são apresentados os pesos dos bits).

Registrador	End. Leitura	End. Escrita	D7	D6	D5	D4	D3	D2	D1	D0	Valor na Inicializ.
Controle	0x00	0x80	0	0	0	OS*	0	0	0	SHDN**	0x05
Temperatura LSByte	0x01	-	2^{-1}	2^{-2}	0	0	0	0	0	0	0x00
Temperatura MSByte	0x02	-	Sinal	2^{6}	2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	0x00

*OS = *One-Shot* (bit de conversão única).
**SHDN = *Shutdown* (bit para desligar o TC72, deixá-lo em *Standby*, consumo de 1μA).

Tab. 14.6 – Modos de operação de acordo com os bits OS e SHDN.

Modo de Operação	OS	SHDN
Conversão contínua	0	0
Desliga	0	1
Conversão contínua	1	0
Conversão única	1	1

O trabalho com o TC72 deve ser feito da seguinte forma:

- A entrada CE (habilitação) do TC72 deve ser colocada em 1 lógico para habilitar a transferência SPI (obs.: nível lógico contrário ao usual para habilitação pela SPI).
- O dado pode ser deslocado na borda de subida ou descida do *clock*; o padrão é na borda de descida.
- A transferência de dados consiste de um byte de endereço seguido por um ou múltiplos dados (2 a 4 bytes).
- O bit mais significativo do byte de endereço determinará uma operação de leitura (bit 7 = 0) ou uma escrita (bit 7 = 1).
- Uma operação de leitura de múltiplos bytes iniciará a partir do endereço mais alto em direção ao mais baixo.
- O usuário pode enviar o endereço do byte alto para ler os dois bytes de temperatura e o byte do registrador de controle. Na fig. 14.6, é ilustrada a leitura e escrita de múltiplos bytes.

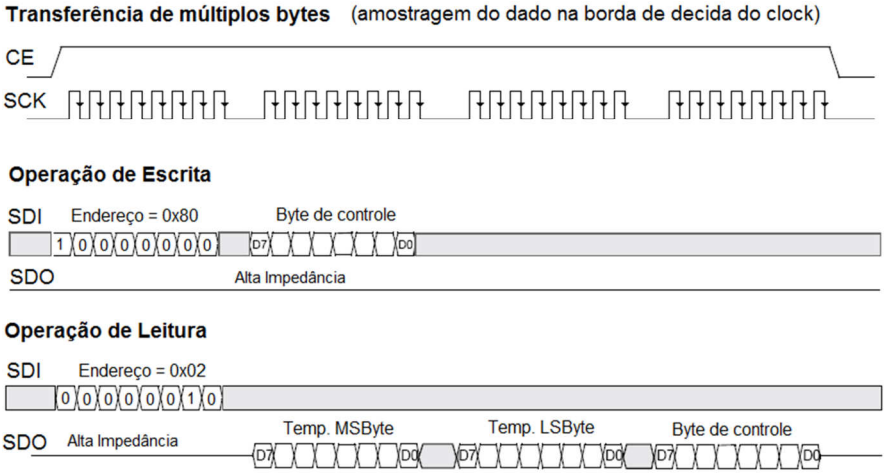

Fig. 14.6 – Transferência de múltiplos bytes para o TC72.

Em resumo, os passos para o trabalho com o TC72 são:

1. CE = 1 para habilitá-lo.
2. Transmitir 0x80 numa operação de escrita, mais o byte de configuração de controle (para conversão contínua = 0x00 ou 0x10). Após, CE =0 para finalizar a configuração.
3. Novamente CE = 1.
4. Transmitir 0x02.
5. Transmitir qualquer valor para ler o byte mais significativo da temperatura.
6. Transmitir qualquer valor para ler o byte menos significativo da temperatura. Se desejado pode ser enviado mais um byte para ler o byte de controle.
7. CE = 0 para desabilitar o TC72, permitindo uma nova conversão.
8. Esperar pelo menos 150 ms e voltar ao passo 3.

A seguir, é apresentado um programa para o trabalho com o TC72. A temperatura com uma resolução de 1ºC é mostrada em um LCD 16 × 2 (o resultado pode ser visto na fig. 14.7).

def_principais.h (arquivo de cabeçalho do programa principal)

```
#ifndef _DEF_PRINCIPAIS_H
#define _DEF_PRINCIPAIS_H

#define F_CPU 16000000UL  //define a frequência do microcontrolador - 16MHz

#include <avr/io.h>     //definições do componente especificado
#include <util/delay.h> //biblioteca para o uso das rotinas de _delay_ms e _delay_us()
#include <avr/pgmspace.h>//para o uso do PROGMEM, gravação de dados na memória flash

//Definições de macros para o trabalho com bits

#define   set_bit(y,bit)  (y|=(1<<bit))//coloca em 1 o bit x da variável Y
#define   clr_bit(y,bit)  (y&=~(1<<bit))//coloca em 0 o bit x da variável Y
#define   cpl_bit(y,bit)  (y^=(1<<bit))//troca o estado lógico do bit x da variável Y
#define   tst_bit(y,bit)  (y&(1<<bit)) //retorna 0 ou 1 conforme leitura do bit

#endif
```

TC72.c (programa principal)

```
//=============================================================================== //
//        Sensor Digital de temperatura TC72 com interface SPI                   //
//        Resolução de amostragem para o LCD de 1 grau Centígrado                //
//=============================================================================== //
#include "def_principais.h"
#include "LCD.h"

#define habilita_TC72()   set_bit(PORTB,PB2)   //nível lógico do pino SS é contrário ao usual
#define desabilita_TC72() clr_bit(PORTB,PB2)

void inic_SPI();
unsigned char SPI(unsigned char dado);
//-------------------------------------------------------------------------------------
int main()
{
     unsigned char temp, tempH,tempL;
     char TEMP[3];

     DDRD = 0b11111100;//LCD
     PORTD = 0b00000011;

     inic_SPI();
     inic_LCD_4bits();

     cmd_LCD(0x80,0);
     escreve_LCD("TEMPERATURA");
     cmd_LCD(0xCE,0);
     cmd_LCD(0xDF,1);
     cmd_LCD('C',1);
```

```
     habilita_TC72();     //CE=1
     SPI(0x80);           //Prepara a escrita
     SPI(0x00);           //Envia byte de controle, conversão continua
     desabilita_TC72();   //CE=0

     while (1)
     {

          _delay_ms(150);

          habilita_TC72();
          SPI(0x02);           //Prepara para a leitura
          tempH=SPI(0x00);     //Lê MSB
          tempL=SPI(0x00);     //Lê LSB
          desabilita_TC72();

          cmd_LCD(0xCA,0);

          if(tst_bit(tempH,7))
          {
               cmd_LCD('-',1);
               tempH = 128 - tempH;//converte para temperatura positiva
          }
          else
               cmd_LCD('+',1);

          //MSB contém o valor da temperatura com resolução de 1 grau centígrado
          temp = tempH & 0b01111111;//zera o bit de sinal
          ident_num(temp, TEMP);
          cmd_LCD(TEMP[2],1);
          cmd_LCD(TEMP[1],1);
          cmd_LCD(TEMP[0],1);
     }
}
//---------------------------------------------------------------------------------
//Inicialização da SPI
//---------------------------------------------------------------------------------
void inic_SPI()
{
     //configuração dos pinos de entrada e saída da SPI
     DDRB = (DDRB & 0b11000011)|(0b00101100);
     SPCR = (1<<SPE)|(1<<MSTR)|(SPR1)|(1<<CPHA);
     /*habilita SPI master, modo 0, CLK = f_osc/64 (250 kHz), dado ajustado na subida e amostragem
     na descida do sinal de clock*/
}
//---------------------------------------------------------------------------------
//Envia e recebe um byte pela SPI
//---------------------------------------------------------------------------------
unsigned char SPI(unsigned char dado)
{
     SPDR = dado;                    //envia um byte
     while(!(SPSR & (1<<SPIF)));     //espera envio
     return SPDR;                    //retorna o byte recebido
}
//---------------------------------------------------------------------------------
```

LCD.h e **LCD.c** (como apresentados no capítulo 5).

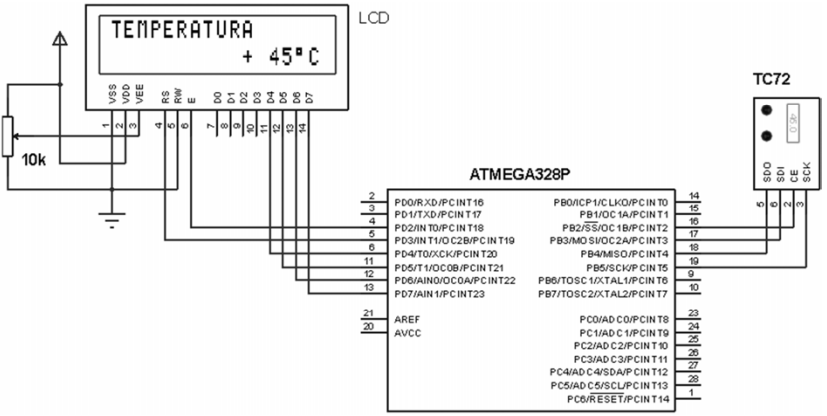

Fig. 14.7 – ATmega328 e o TC72.

14.3 CARTÃO DE MEMÓRIA SD/MMC

Os cartões de memória *flash*, como os encontrados em celulares, câmeras digitais, *tablets* e computadores, são atualmente muito utilizados. Entre a infinidade de dispositivos de memória disponíveis, os cartões mais comuns são o SD (*Secure Digital memory card*) e o MMC (*Multi Media Card*). Eles podem usar o protocolo SPI e, portanto, têm sido muito empregados em sistemas embarcados. Sua principal aplicação é no armazenamento de dados em sistemas que necessitam gravar variáveis temporais, como por exemplo, *data loggers*.

Os SD e os MMC são compatíveis entre si. Todavia, o MMC é mais fino e o seu suporte não permite o encaixe do SD; já o contrário é possível, o suporte para o SD permite o encaixe do MMC. Os SDs apresentam uma chave de segurança contra escritas acidentais, são mais populares e, ainda, são disponibilizados em tamanhos menores, como o mini SD e o micro SD (utilizado nos celulares e *tablets*). Os SDs padrão possuem capacidade de memória de até 4 GB. Todavia existem modelos que chegam até 32 GB (*High-Capacity SD Card* – SDHC). Na fig. 14.8, é apresentado a pinagem para o SD e o MCC, incluindo o diagrama esquemático para o SD.

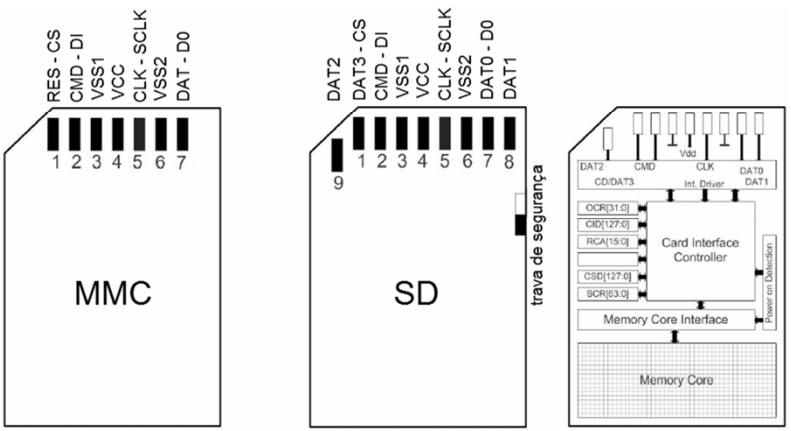

Fig. 14.8 – Pinagem para os cartões de memória MMC e SD, incluindo o diagrama esquemático para o SD (adaptado de: *SD Card Reader Using the M9S08JM60 Series Designer Reference Manual*).

A tensão de alimentação para os SD/MMCs deve estar compreendida entre 2,7 V e 3,6 V; geralmente, utiliza-se 3,3 V. Dessa forma, para sistemas alimentados com 5 V, é necessário o emprego de resistores configurados como divisores de tensão para a limitação dos sinais de tensão do microcontrolador para o cartão de memória (fig. 14.9). A corrente durante uma operação de escrita pode chegar a 100 mA.

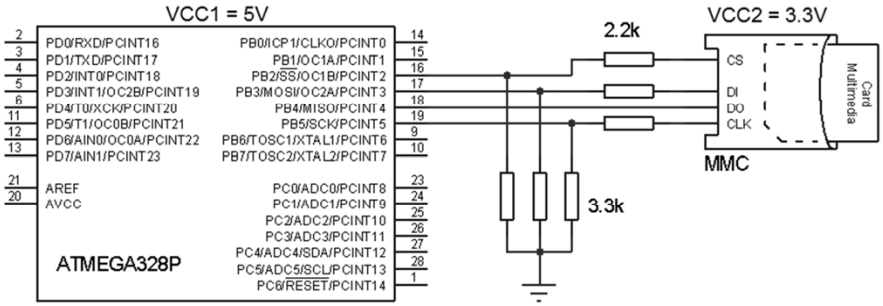

Fig. 14.9 – Circuito para adaptar os níveis lógicos de tensão de um microcontrolador alimentado com 5 V para a comunicação com um cartão SD/MMC.

Os cartões SD possuem dois modos de operação: o modo nativo - padrão, chamado modo de barramento SD, e o modo de barramento SPI (igual para o MMC). O modo nativo é bem mais complexo que o modo SPI e é empregado em sistemas que necessitam obter o máximo desempenho de escrita e leitura e podem dispor de uma programação mais complexa. Nesse modo, todos os pinos do cartão são utilizados e os dados são transmitidos com 4 pinos (D0-D3), um pino de *clock* (CLK) e um de comando (CMD). O modo SPI é o mais fácil de programar e é o preferido para uso com microcontroladores, pois utiliza apenas os pinos da SPI (MISO – DO, MOSI – DI, CLK e SS - CS), conforme apresentado na fig. 14.9.

No modo SPI, a transmissão de dados é feita por bytes de forma serial. Os comandos utilizam 6 bytes de formato, conforme fig. 14.10. Após um determinado tempo, o cartão de memória retorna um ou mais bytes de resposta, chamados R1, R2 ou R3. O índice indica qual comando será aplicado e pode ter um valor entre 0 e 63. O argumento utilizado vai depender de qual comando está sendo empregado; no caso de leitura ou escrita, conterá um endereço.

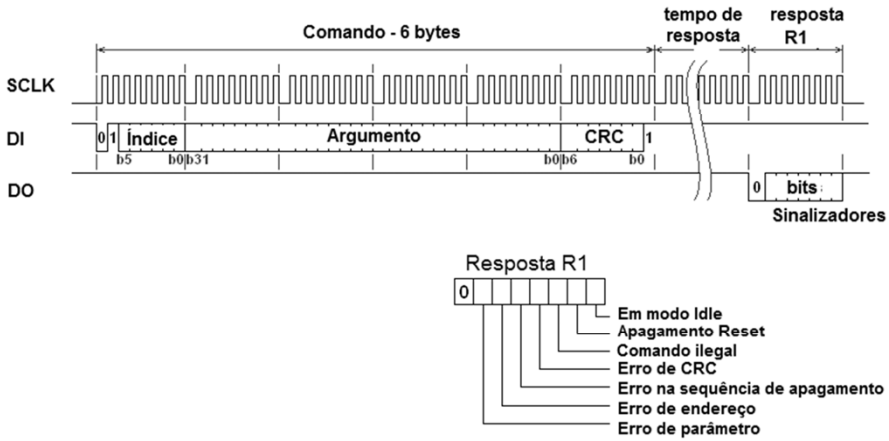

Fig. 14.10 - Formato do comando e resposta para o controle de um SD/MMC no modo SPI (adaptado de: http://elm-chan.org/docs/mmc/mmc_e.html).

Como a geração de *clock* é feita pelo dispositivo de controle (microcontrolador), esse deve ficar continuamente lendo os dados da memória até obter um valor válido. O pino DI (MOSI) deve ser mantido em nível lógico alto. Portanto, deve ser enviado o valor 0xFF na leitura de dados. Para realizar algum comando, o cartão de memória deve ser habilitado, colocando-se o pino CS em nível lógico zero. O CRC (*cheksum* – conferência de dados) é opcional e, geralmente, desabilitado no modo SPI.

Existem inúmeros comandos que podem ser utilizados. Para as operações básicas são necessários apenas os seguintes:

- CMD0 (índice = 0) <GO_IDLE_STATE>, inicialização da memória, entrada no modo *Idle*. O argumento para a função é 0x00000000 e deve ser utilizado o valor 0x95 para o CRC (única vez, depois ele estará desabilitado no modo SPI, a menos que programado o contrário). Assim, o CMD0 será 0x400000000095.
- CMD1 (índice = 1) <SEND_OP_COND>, quando a memória está no modo *Idle*, ela pede a condição de operação para saber se a inicialização foi efetuada corretamente. O formato do CMD1 será 0x4100000000FF (CRC não empregado = 0xFF).
- CMD16 (índice = 16) <SET_BLOCKLEN>, ajusta o comprimento do bloco de dados em bytes (512, 1024, 2048, 4096) para a escrita e leitura. O padrão é 512 e não precisa ser ajustado, a menos que o contrário seja desejado. O cartão de memória só permite a escrita e leitura em blocos.
- CMD17 (índice = 17) <READ_SINGLE_BLOCK>, leitura de um único bloco de dados. O formato de dados será 0x58xxxxxxxxFF, onde xx é o endereço de início da leitura, a qual sempre é feita em blocos com o tamanho determinado pelo CMD16.
- CMD24 (índice = 24) <WRITE_BLOCK>, escrita de um bloco de dados, cujo tamanho é determinado pelo CMD16. O formato será 0x51xxxxxxxxFF, onde xx é o endereço de início para a escrita do bloco de dados.

Na transmissão ou recepção do bloco de dados, sempre é transmitido primeiro o bit mais significativo (MSB), sendo que o bloco deve possuir 515 bytes (*Data Tokens*), dispostos da seguinte forma:

<1 byte indicando o início>:<512 bytes de dados>:<2 bytes de CRC>

O primeiro byte deve ser 0xFE, indicando o início do bloco de dados, depois seguem os 512 bytes dos dados e, por último, dois bytes do CRC dos dados. Quando este não for empregado, escreve-se 0xFFFF.

Quando o cartão de memória é ligado, ele encontra-se no modo nativo e precisa ser configurado para o modo SPI (nesse modo, o CRC é desabilitado por padrão). Os seguintes passos são necessários para inicializar corretamente um cartão SD:

- Enviar no mínimo 74 pulsos de *clock* para o cartão com o pino CS e DO em 1.
- Habilitar o cartão, fazendo CS = 0.
- Enviar o comando CMD0.
- Verificar a resposta R1 para se certificar que não existe algum bit de erro.
- Enviar repetidamente o comando CMD1, até que o bit de 'em modo *Idle*' da resposta R1 ser colocado em zero e não haver bits de erro.

Após a inicialização, a memória está pronta para o uso. Durante a inicialização, a frequência da SPI deve estar entre 100 kHz e 400 kHz. Após, pode ser aumentada para até 25 MHz para o cartão SD e 20 MHz para o MMC.

A forma básica de escrita é simples. Todavia, pode ser empregado o sistema de arquivo FAT16 para uso nos cartões SD padrão e FAT32 para cartões com capacidades maiores de memória. O emprego de um sistema de arquivos FAT permite a leitura dos dados diretamente em um computador. Entretanto, agrega maior complexidade ao software de controle.

A seguir, é apresentado um programa para escrita e leitura de um cartão SD/MMC pelo ATmega328. O programa principal escreve em três setores da memória (blocos de 512 bytes) e verifica se o último conjunto de dados escrito corresponde ao lido. Foi empregado um LED sinalizador para indicar a ocorrência de erro ou sucesso na operação. O programa analisa a ocorrência de erros na comunicação e usa variáveis para a contagem, permitindo um tempo máximo de resposta para o cartão.

def_principais.h (arquivo de cabeçalho do programa principal)

```
#ifndef DEF_PRINCIPAIS_H_
#define DEF_PRINCIPAIS_H_

#define F_CPU 16000000UL

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

//Definições de macros para o trabalho com bits

#define   set_bit(y,bit) (y|=(1<<bit))  //coloca em 1 o bit x da variável Y
#define   clr_bit(y,bit) (y&=~(1<<bit)) //coloca em 0 o bit x da variável Y
#define   cpl_bit(y,bit) (y^=(1<<bit))  //troca o estado lógico do bit x da variável Y
#define   tst_bit(y,bit) (y&(1<<bit))   //retorna 0 ou 1 conforme leitura do bit

#endif
```

SD_MMC_card.c (programa principal)

```
//-------------------------------------------------------------------------------//
//   Programa para a escrita e leitura de um SD/MCC                              //
//   Escrita e leitura em blocos de 512 bytes                                    //
//-------------------------------------------------------------------------------//

#include "def_principais.h"
#include "SD_MMC.h"

#define liga_led()        set_bit(PORTD,5)     //ativo em 1
#define desliga_led()     clr_bit(PORTD,5)

/*aloca 512 bytes da RAM para poder escrever em blocos, como necessário para o
                                                              trabalho com o SD card*/
unsigned char dados[512];

int main(void)
{
     unsigned int i;
```

```
	DDRD = 1<<5;    //PD5 possui o LED sinalizador de erro
	desliga_led();  /*Só liga o LED se houver erro, pisca led ao final se o
                                                        funcionamento foi correto*/
	inic_SPI();

	//inicializa MMC - após inicialização a velocidade da SPI pode ser aumentada
	if(inic_MMC()!=0)
		liga_led();

	//teste para a escrita em 3 setores da memória
	//-----------------------------------------------------------------------------
	//setor 1
	for(i=0;i<512;i++)   dados[i]=0xAA;

	if(escreve_MMC(dados,0,0)!=0)  //escreve 0xAA nos 512 bytes do setor 1
		liga_led();
	//-----------------------------------------------------------------------------
	//setor 2
	for(i=0;i<512;i++)   dados[i]=0xBB;

	if(escreve_MMC(dados,0,512)!=0)     //escreve 0xBB nos 512 bytes do setor 2
		liga_led();
	//-----------------------------------------------------------------------------
	//setor 1000 = 0x7CE00
	for(i=0;i<512;i++)   dados[i]=(uchar)i;

	if(escreve_MMC(dados,0x0007,0xCE00)!=0)  //escreve 0-511 nos 512 bytes do setor 1000
		liga_led();
	//-----------------------------------------------------------------------------
	//comparação da escrita apenas para o setor 1000
	if(le_MMC(dados,0x0007,0xCE00)!=0)       //lê as primeiras 512 posições
		liga_led();

	//-----------------------------------------------------------------------------
	//compara os dados escritos com os lidos
	for(i=0;i<512;i++)
	{
		if(dados[i]!=(uchar)i)
			liga_led();     //liga o led se o dado escrito for diferente do lido
	}

	//se o led não foi ligado fica piscando indicando sucesso na operação de escrita e leitura
	if(!tst_bit(PORTD,5))
	{
		while(1)
		{
			liga_led();
			_delay_ms(300);
			desliga_led();
			_delay_ms(300);
		}
	}
	//-----------------------------------------------------------------------------
	while(1);
}
```

SD_MMC.h (arquivo de cabeçalho do SD_MMC.c)

```
#ifndef SD_MMC_H_
#define SD_MMC_H_

#include "def_principais.h"

typedef   unsigned char uchar;
typedef   unsigned int  uint;

#define DI      PB3       //entrada de dados (dados da memória)
#define DO      PB4       //saída de dados (dados para a memória)
#define CLK     PB5       //clock
#define CS      PB2       //seleção da memória (habilitação)

//comandos básicos para o SD/MMC

#define habilita_MMC()    clr_bit(PORTB,CS)
#define desabilita_MMC()  set_bit(PORTB,CS)

void inic_SPI();
unsigned char SPI(unsigned char c);

void comando_MMC(uchar index, uint ArgH, uint ArgL, uchar CRC);
unsigned char resposta_MMC(uchar resposta);
unsigned char inic_MMC();
unsigned char escreve_MMC(uchar *buffer, uint enderecoH, uint enderecoL);
unsigned char le_MMC(uchar *buffer,uint enderecoH, uint enderecoL);

#endif
```

SD_MMC.c (arquivo com as funções para o trabalho com os cartões SD/MMC)

```
#include "SD_MMC.h"
//-------------------------------------------------------------------------------------
//Inicialização da SPI
//-------------------------------------------------------------------------------------
void inic_SPI()
{    //configuração dos pinos de entrada e saída da SPI
     DDRB = (DDRB & 0b11000011)|(0b00101100);

     habilita_MMC(); //CS = 0

     SPCR = (1<<SPE)|(1<<MSTR)|(SPR1);/*habilita SPI master, modo 0, CLK = f_osc/64
                                          (250 kHz), 100 kHz  <= f_osc <= 400 kHz*/
}
//-------------------------------------------------------------------------------------
//Envia e recebe um byte pela SPI
//-------------------------------------------------------------------------------------
unsigned char SPI(uchar dado)
{
     SPDR = dado;                   //envia um byte
     while(!(SPSR & (1<<SPIF)));    //espera envio

     return SPDR;                   //retorna o byte recebido
}
```

```
//---------------------------------------------------------------------------------------
//Envia um comando para o MMC - index + argumento + CRC (total de 6 bytes)
//---------------------------------------------------------------------------------------
void comando_MMC(uchar index, uint ArgH, uint ArgL, uchar CRC)
{
     //INDEX - 6 bits
     SPI(index);//sempre bit 7 = 0 e bit 6 = 1, b5:b0 (index)

     //ARGUMENTO - 32 bits
     SPI((uchar)(ArgH >> 8));
     SPI((uchar)ArgH);
     SPI((uchar)(ArgL >> 8));
     SPI((uchar)ArgL);

     //CRC
     SPI(CRC);//bit 0 sempre 1
}
//---------------------------------------------------------------------------------------
//Lê continuamente o MMC até obter a resposta desejada ou estourar o tempo permitido
//---------------------------------------------------------------------------------------
unsigned char resposta_MMC(uchar resposta)
{
     unsigned int cont = 0xFFF;//contador para espera de um máximo tempo da resposta

     do
     {
          if(SPI(0xFF)==resposta)
               return 0; //resposta normal laço acabou antes do estouro de contagem
          cont--;
     }while(cont!=0);
     return 1;           //falha na resposta
}
//---------------------------------------------------------------------------------------
//Inicializa o MMC para trabalhar no modo SPI
//---------------------------------------------------------------------------------------
unsigned char inic_MMC()
{
     unsigned char i;

     desabilita_MMC();
     for(i=0; i<10; i++) SPI(0xFF); //80 pulsos de clock
     habilita_MMC();

     //CMD0 (GO TO IDLE STATE - RESET) - único vez que envia o checksum (0x95)
     comando_MMC(0x40,0,0,0x95);

     if (resposta_MMC(0x01)) { desabilita_MMC(); return 1;}/*retorna 1 para indicar
                                             que houve erro (não entrou no modo Idle)*/
     desabilita_MMC();
     SPI(0xFF);          //alguns clocks após o modo Idle
     habilita_MMC();

     i=0xFF;
     /*envia o CMD1 até resposta ser zero (não estar mais no modo Idle), não haver
                                    erro, e estar dentro do nr. máximo de repetições*/
     do
     {
          i--;
          //CMD1 (SEND_OP_COND) - verifica quando saiu do modo Idle
          comando_MMC(0x41,0,0,0xFF);

     } while ((resposta_MMC(0x00)!=0) && (i!=0));
```

```
     //retorna 2 para indicar que houve erro não saiu do modo Idle)
     if(i==0){ desabilita_MMC(); return 2;}

     desabilita_MMC();
     SPI(0xFF);      //alguns clocks após modo Idle

     return 0;       //retorna 0 para indicar sucesso na operação
}
//-----------------------------------------------------------------------------
//Escreve a partir do endereço especificado (setor), sempre em blocos de 512 bytes
//Os endereços podem ser  0, 512, 1024, 1536 ...
//Calculados como :
//   (Nr_Setor - 1) x 512. Por exemplo: Nr_Setor = 1000, resulta no endereço 0x7CE00
//-----------------------------------------------------------------------------
unsigned char escreve_MMC(uchar *buffer, uint enderecoH, uint enderecoL)
{
     unsigned int i;

     habilita_MMC();

     //CMD24 (WRITE_SINGLE_BLOCK) - escreve um único bloco de 512 bytes
     comando_MMC(0x58,enderecoH,enderecoL,0xFF);

     //retorna 1 para indicar erro do CMD24
     if (resposta_MMC(0x00))   { desabilita_MMC(); return 1;}

     SPI(0xFE);//indica o início do bloco de dados

     for(i=0;i<512;i++)//transfere os 512 bytes do buffer para o MMC
     {
          SPI(*buffer);
          buffer++;
     }

     //no final envia dois bytes sem utilidade (dummy checksum)
     SPI(0xFF);
     SPI(0xFF);

     //retorna 2 para indicar erro na escrita
     if ((SPI(0xFF) & 0x0F) != 0x05) {desabilita_MMC();  return 2;}

     //espera final de escrita
     i=0xFFFF;
     do
     {    i--;
     } while ((SPI(0xFF)== 0x00) && i);

     //retorna 3 para indicar erro no final da escrita
     if(i==0){ desabilita_MMC(); return 3;}

     desabilita_MMC();    //desabilita o SD card
     SPI(0xFF);

     return 0; //retorna 0 para indicar sucesso na operação
}
//-----------------------------------------------------------------------------
//Lê a partir do endereço especificado (setor), sempre em blocos de 512 bytes
//(como na escrita)
//-----------------------------------------------------------------------------
```

```
unsigned char le_MMC(uchar *buffer,uint enderecoH, uint enderecoL)
{
     unsigned int i;

     habilita_MMC();

     //CMD17 (READ_SINGLE_BLOCK) - leitura de um único bloco de 512 bytes
     comando_MMC(0x51,enderecoH,enderecoL,0xFF);

     //retorna 1 para indicar erro na leitura
     if (resposta_MMC(0x00))   {desabilita_MMC();   return 1;}

     //retorna 2 para indicar erro na leitura
     if (resposta_MMC(0xFE))   {desabilita_MMC();   return 2;}

     //leitura dos 512 bytes para o vetor de memória desejado (buffer)
     for(i=0; i < 512; i++)
     {
          *buffer=SPI(0xFF);  //envia 0xFF para receber o dado
          buffer++;
     }

     //no final envia dois bytes, retornam o CRC/checksum byte (sem importância no programa)
     SPI(0xFF);
     SPI(0xFF);

     desabilita_MMC();
     SPI(0xFF);

     return 0; //retorna 0 para indicar sucesso na operação
}
//---------------------------------------------------------------------------------
```

Exercícios:

14.1 – Altere o programa para o TC72 (pág. 329), apresentando a temperatura com uma resolução de 0,25 °C (fig. 14.7).

14.2 – Elaborar um programa para a leitura do sensor de temperatura MAX6675 do circuito da fig. 14.11.

O MAX6675 digitaliza o sinal proveniente de um termopar tipo K[38] e sua saída de dados é de 12 bits com compatibilidade SPI. Sua resolução é de 0,25°C, com medição de temperatura entre 0 e 1024°C.

[38] O termopar tipo K é um sensor de temperatura composto por Chromel (90% de Níquel e 10% de Cromo) e Alumel (95% de Níquel, 2% de Manganês, 2% de Alumínio e 1% de Si1ício). Os dois materiais são conectados e na sua junção é gerada uma pequena diferença de potencial dependendo da temperatura a que são submetidos. Fonte http://pt.wikipedia.org/wiki/Termopar.

Seu protocolo de comunicação é apresentado na fig. 14.12. Nela se observa que os bits mais significativos do dado são enviados primeiro. O formato do dado é apresentado na tab. 14.7.

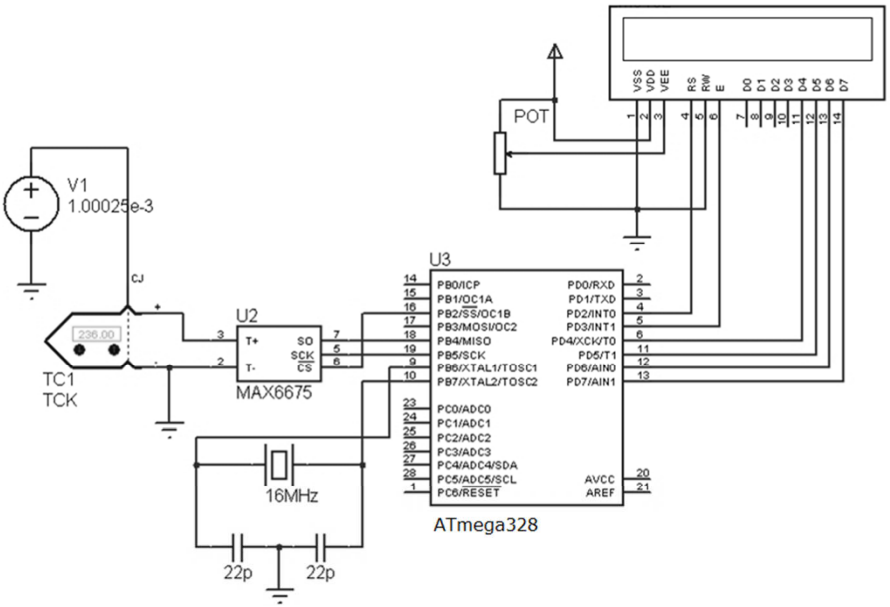

Fig. 14.11 – Usando o sensor MAX6675 (a fonte V1 é para a compensação da tensão gerada pelo termopar, utilizada para a simulação).

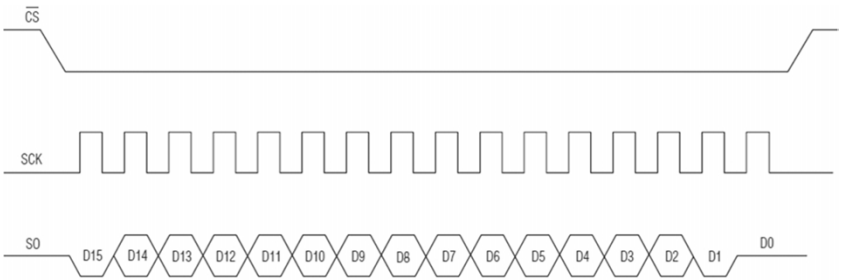

Fig. 14.12 – Protocolo de comunicação do MAX6675.

Tab. 14.7 - Formato de dados enviado pelo MAX6675.

	Bit de Sinal	Leitura de 12 bits da temperatura												Entrada do termopar	ID do dispositivo	Estado
Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	MSB											LSB		0	*Tri-State*

Para o emprego da SPI do ATmega, basta escrever um dado qualquer no registrador de dados da SPI (SPDR). O controle do pino $\overline{CS}$ deve ser feito por software e os dados serão recebidos no pino MISO. O trecho de programa abaixo ilustra o processo.

```
//-------------------------------------------------------------------------------
//MAX6675
     ...                  //configurações para uso da SPI
     clr_bit(PORTB,SS);   //pino SS em zero
     temp_MSB =le_MAX();  //lê o byte + signific.(temp_MSB é um unsigned char)
     temp_LSB =le_MAX();  //lê o byte - signific.(temp_LSB é um unsigned char)
     set_bit(PORTB,SS);   //pino SS em 1

     valor_int = (temp_LSB>>3) | (temp_MSB<<5);         /*são 12 bits de dados, os
                                                          demais não são analisados*/
     valor_int = valor_int/4;  /*divide o valor por 4 para uma resolução de 1 grau,
                                 ou valor_int=(temp_LSB>>5)|(temp_MSB<<3) – já divide.
                                                      valor_int é um unsigned int*/
     ...                  //aqui vai a rotina para mostrar o valor no display
//-------------------------------------------------------------------------------
unsigned char  le_MAX()
{
     SPDR = 0x00;                  //envia um dado qualquer para recebe um byte
     while(!(SPSR & (1<<SPIF)));   //aguarda o envio do dado
     return SPDR;                  //retorna o valor recebido
}
//-------------------------------------------------------------------------------
```

14.3 – Como o MAX6675 poderia ser utilizado para o projeto de um forno para solda de componentes SMD?

14.4 – Pesquise o sistema de arquivos FAT16/FAT32 e desenvolva um programa para escrever e ler dados nesses formatos em cartões SD/MMC.
