Experimento 4 - Memória EEPROM

EA076 - Laboratório de Sistemas Embarcados

Turma C - Grupo 6

Gustavo Ciotto Pinton RA 117136

Anderson Une Bastos RA 093392

Campinas, 25 de abril de 2016

Introdução

O objetivo deste experimento foi integrar uma memória do tipo EEPROM ao microcontrolador e controlá-la via o protocolo I2C. Para tal, utilizamos o componente AT24C16B da Atmel e o módulo I2C, já implementado no *kit* Freescale KL25Z. Memórias do tipo EPROM, do inglês *erasable programmable read-only memory,* conservam os seus dados mesmo quando não alimentadas por nenhuma fonte de tensão, isto é, são componentes ditos *não voláteis* (ou *persistentes*), ao contrário das memórias do tipo RAM. Tais memórias permitem, conforme o nome, apenas leituras de valores, porém implementam igualmente mecanismos de reprogramação de seu conteúdo, explicando, assim, o *E* antes de *PROM*. Memórias EPROM têm seu conteúdo apagado quando submetidas à luz ultravioleta, sendo que neste tipo de dispositivo, não há como apagar frações singulares da memória. Logo, quando queremos apagar apenas um *byte*, é necessário limpar todos os demais dados, impondo, portanto, um grande problema. Uma alternativa é o uso de memórias EEPROM, *electrically erasable programmable read-only memory*, que permitem a modificação de *bytes* individualmente através de sinais elétricos.

O controle da memória EEPROM AT24C16B é feita por meio do protocolo I2C, que apresenta basicamente 2 sinais de controle. Um deles é o *clock* de sincronismo, chamado de *SCL*, e o outro, *SDA*, para transmissão *serial* dos dados. Ao contrário de muitas memórias que obtém os dados paralelamente através de muitos barramentos , a comunicação I2C é *serial* e utiliza apenas 2 sinais. As próximas seções visam a detalhar o seu funcionamento, assim como a montagem realizada no experimento.

Implementação das conexões (itens 1, 2 e 3)

A tabela contida na seção 10.3.1 de [1] contém todas as possíveis funções de cada pino dos *headers* do microcontrolador. Para utilizar um dos dois módulos que implementam o protocolo I2C, é necessário, desta maneira, procurar os pinos que possuem ao menos uma de suas funções associada a eles. Para o módulo *I2C0*, muitos pinos são disponibilizados, entre eles, `(PTE24, PTE25)`, `(PTB0, PTB1)`, `(PTB2, PTB3)` e `(PTC8, PTC9)`, em que cada tupla representa um par de sinais `(SCL, SDA)`. Para o módulo *I2C1*, apenas 4 pinos estão adaptados: `(PTC1, PTC2)` e `(PTC10, PTC11)`. Conforme sugerido, utilizaremos a tupla `(PTC10, PTC11)`, já que tais pinos ainda não foram usados para nenhuma outra aplicação.

Segundo a tabela *Pin Configurations* de [2], além dos sinais de controle `SCL`, `SDA` e de alimentação, 4 outras entradas são configuráveis. O sinal `WP` impede, caso esteja em nível lógico alto, a escrita no componente. No nosso caso, não utilizaremos esta função e, portanto, ligaremos esta entrada diretamente ao terra `GND`. Os pinos `A0`, `A1` e `A2` determinam, por sua vez, o endereço que o componente, atuando como *slave*, possuirá. Tais sinais são importantes porque, antes do envio dos dados de leitura ou escrita, o *master* transmite a identicação, isto é, o endereço, do *slave* que ele pretende atingir. Quando o pretendido *slave* reconhece seu *id* na mensagem transmitida pelo *master*, ele o encaminha um *acknowledge bit* e a comunicação entre os dois pode ser iniciada. Como teremos somente um escravo, podemos associar a ele a identificação 0, que equivale a ligar `A0`, `A1` e `A2` diretamente ao terra.

O componente AT24C16B é capaz de operar em diversas faixas de alimentação, especificadas no primeiro item da lista *Features* em [2]. Para tensões V_{CC} de alimentação entre 2.7V e 5.5V, por

exemplo, o valor de tensão baixa a ser adotada é 2.7V. A escolha da alimentação foi feita com base na tabela *AC Characteristics,* também presente em [2]. Considerando que as requisições temporais são mais leves para alimentações inferiores a 5.0V, escolhe-se a saída de 3.3V do próprio *kit* para alimentar a memória EEPROM. [2] também especifica que as saídas `SCL` e `SDA` são do tipo *open-drain* e a documentação do módulo I2C de [1] afirma que *"all devices connected to it must have open drain or open collector outputs"* (subseção 38.4.1). Este conceito é similar àquele de *open-collector*, porém se aplica a transistores *MOSFET*. Dessa maneira, será necessário o uso de dois resistores *pull-up,* que além de limitarem a corrente a fim de não danificar as portas do microcontrolador, atribui uma tensão de referência a tais saídas. Foram escolhidas resistências da ordem de 1kΩ, que limitam as correntes ao valor máximo de 3.3mA. A figura 1 abaixo representa a ligação destes resistores *pull-up* com o *chip* e as saídas `PTC10` e `PTC11`.

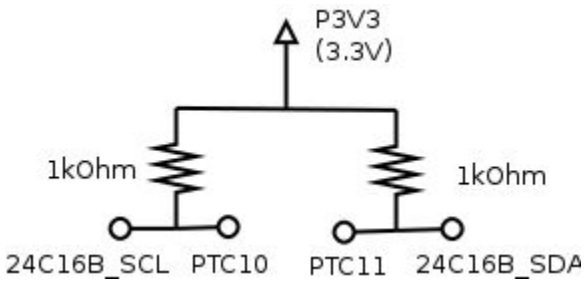


Figura 1: Ligações do *chip* e das portas ao resistores *pull-up*

Síntese dos sinais de Controle (itens 4 e 5)

O protocolo I2C estabelece que, no início da comunicação entre *master* e *slave,* um sinal de *START* deve ser enviado. Da mesma maneira, quando a troca de dados é finalizada, um sinal de *STOP* também deve ser transmitido. O *START* é caracterizado por uma borda de descida do sinal `SDA` quando `SCL` é alto, enquanto que o *STOP* é definido por uma borda de subida de `SDA` quando `SCL` é alto. A figura 2, retirada de [2], exemplifica tais sinais.

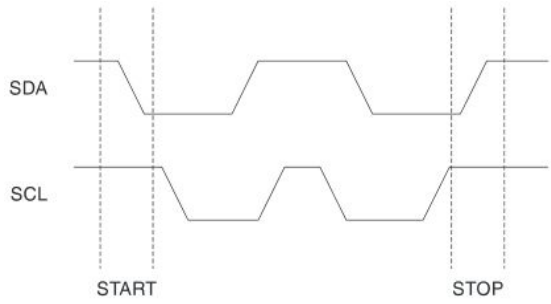


Figura 2: sinais de START e STOP. Retirado de [2].

Após um *START*, o *master* deve enviar o endereço do componente *slave* que realizará uma leitura ou escrita. Tal operação é chamada de *Device Addressing* e é realizada através do envio de uma palavra de 8 *bits* [7:0], sendo que o *bit* mais significativo, isto é o 7, neste caso, vale sempre 1. Os *bits* [6:4] devem refletir o estado dos pinos `A0`, ¬`A1` e `A2` do *slave* alto, nesta ordem. No nosso caso, tais bits sempre serão 010. Em seguida, os *bits* [3:1] representam os três *bits* mais significativos do endereço de memória sobre a qual a operação deve ser realizada. Em termos práticos, tais *bits* especificam um dos 8 blocos de 256 palavras de 8 *bits* contidos na memória. Enfim, o *bit* 0 especifica se a operação é de leitura (1) ou de escrita (0). Tal estrutura está representada na figura 2, também retirada de [2]. Quando o *slave* reconhece seu endereço, ele envia um sinal de *acknowledge,* impondo o nível lógico baixo durante um ciclo de `SCL` no barramento `SDA`.

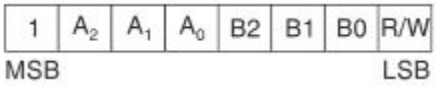
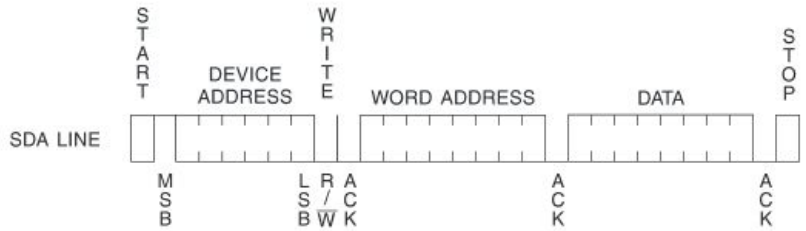
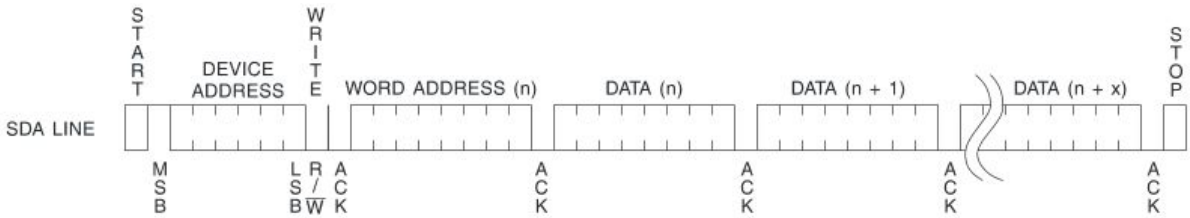


Figura 3: estrutura da palavra de *device addressing*. Retirado de [2].

Para a operação de escrita, dois modos são disponíveis: *byte write* e *page write*. Para o primeiro modo, após o primeiro *byte* utilizado no *device addressing,* o *master* envia uma outra palavra com os 8 *bits* restantes referentes ao resto do endereço da memória e espera um sinal de *acknowledge* do *slave*. Uma vez recebido, o *master* procede e envia o próximo *byte*, contendo os dados a serem escritos. Após receber novamente um *acknowledge*, o *master* envia o sinal de *STOP*, indicando o fim da comunicação. A memória EEPROM, então, desabilita todas as suas entradas e entra em um ciclo de escrita, que pode demorar até 10ms para uma alimentação de 3.3V. O segundo modo é capaz de escrever até 16 palavras de *1 byte* (uma página) de maneira sequencial na memória. O início da transmissão é idêntico àquele do modo *byte write*, porém, após o primeiro *byte* de dados, o microcontrolador não envia um sinal de *STOP*. Em vez disso, ele continua a enviar *bytes* de dados. A memória EEPROM possui um contador interno que é incrementado a cada recebimento de um *byte*. Caso mais de 16 palavras sejam enviadas, o memória sobreescreverá os conteúdos de maneira cíclica, isto é, o 17º *byte* será escrito sobre o 1º e assim por diante. A figura 4 simboliza estes dois modos de operação.



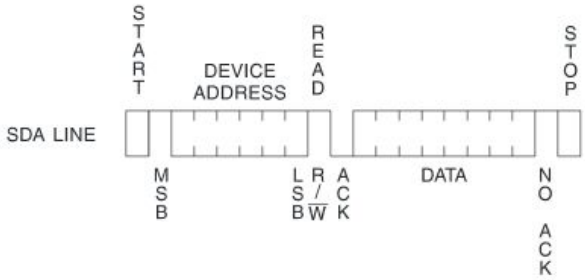
(a) Operação de escrita do modo *byte write.*



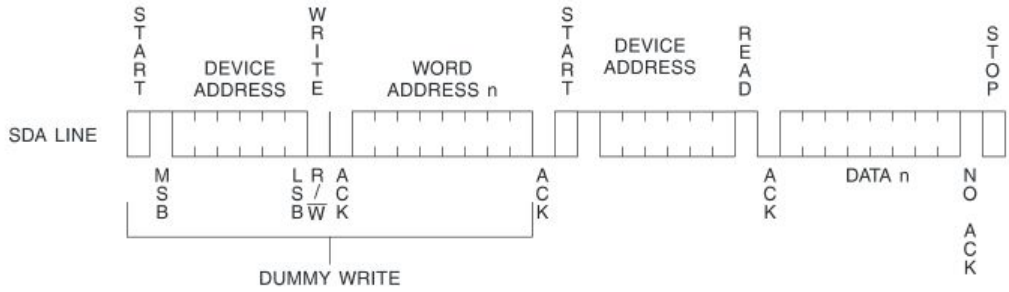
(b) Operação de escrita do modo *page write.*

Figura 4: Modos de escrita na memória EEPROM. Retirado de [2].

Para a operação de leitura, três modos são disponíveis: *current address read, random read* e *sequential read*. No experimento, somente as duas primeiras foram utilizadas, portanto serão as únicas explicadas neste relatório. O mecanismo de *device addressing* é o mesmo explicado anteriormente. O primeiro modo retorna o *byte* contido no endereço armazenado no contador interno da memória, que representa o último acesso realizado por ela em uma operação de escrita ou leitura. Após receber o respectivo *byte*, o microcontrolador envia o sinal de *STOP*. O modo *random read* combina uma operação de escrita, a fim de modificar o endereço do contador interno, com uma leitura do tipo *current address read*. Neste modo, o microcontrolador inicia uma sequência de escrita, chamada de *dummy write,* no endereço de memória desejado, mas a interrompe antes do envio do *byte* referente aos dados e, em seguida, encaminha uma operação de *current address read*. A figura 5 representa os modos *current address read* e *random read*.



(a) Operação de leitura do modo *current address read.*



(b) Operação de leitura do modo *random read.*

Figura 5: Modos de leitura na memória EEPROM. Retirado de [2].

Um *driver* que implementa todos os sinais detalhados acima é disponibilizado para o *Processor Expert*. A sua configuração e utilização serão discutidos na seção **Implementação do Firmware** deste relatório.

Em suma, no nosso experimento, o *device ID* do componente é 0 e o tamanho máximo de um bloco (ou página) é 16 *bytes*.

Compatibilidade Temporal dos Sinais (item 6)

A tabela *AC Characteristics* de [2] fornece várias características temporais que devem ser levadas em consideração para o bom funcionamento do componente. Entre elas, destacam-se, para uma alimentação de 3.3V:

- a frequência máxima admitida, $f_{SCL}^{max} = 100\ kHz$;
- o tempo mínimo necessário de retenção dos dados, *SDA (Data) Hold time*:
$$t_{SDA.hold}^{min} = t_{HD.DAT}^{min} + t_{SD.DAT}^{min} = 200ns$$
- os tempos mínimos de retenção para os sinais *START* e *STOP*, sendo, respectivamente, $t_{HD.STA}^{min} = 4.7\mu s$ e $t_{SU.STO}^{min} = 4.7\mu s$.

A configuração dos divisores de frequência e dos tempos de *hold* do módulo I2C1 é realizada por meio do registrador `I2C1_F`, que contém dois campos: `MULT`, de 2 *bits*, e `ICR`, de 6. Na descrição deste registrador, na seção 38.3.2 de [1], são explicitadas algumas equações capazes de determinar os tempos acima através do valor de `MULT` e de algumas constantes determinadas por `ICR`. Tais constantes podem ser encontradas na tabela 38.41 de [1].

A partir das equações e definindo `MULT` = `0x00` (`mul` = 1) e `ICR` = `0x100010` (a partir da tabela 38.41, `SCL divider = 224`, `SDA hold value = 33`, `SCL hold (start) value = 110` e `SCL hold (stop) value = 113`), obtém-se:

- $f_{SCL} = \frac{20.98MHz}{mul\ x\ SCL\ divider} = \frac{20.98x10^6}{1\ x\ 224} = 93.67kHz < 100\ kHz$;
- $SDA\ hold\ time = \frac{mul * SDA\ hold\ value}{20.98x10^6} = \frac{1 * 33}{20.98x10^6} = 1.573\mu s > 200ns$;
- $SCL\ start\ hold\ time = \frac{mul * SCL\ start\ value}{20.98x10^6} = \frac{1 * 110}{20.98x10^6} = 5.243\mu s > 4.7\mu s$;
- $SCL\ stop\ hold\ time = \frac{mul * SCL\ stop\ value}{20.98x10^6} = \frac{1 * 113}{20.98x10^6} = 5.386\mu s > 4.7\mu s$

As requisições temporais são, portanto, atendidas.

Implementação do Firmware (itens 7, 8 e 9)

Nesta seção, serão detalhados os recursos utilizados na implementação do *firmware*.

Driver 24AA_EEPROM

Um *driver* para comunicação *serial* via protocolo I2C com memórias EEPROM é disponibilizado em [3]. Tal *driver* implementa todos os sinais de controle detalhados conforme seção **Síntese dos sinais de controle** e provê, além de uma interface gráfica de configuração dos registradores do módulo I2C1, alguns métodos para a escrita e leitura de valores da memória, listados abaixo.

- `ReadByte(addr, byte* data)`: realiza a leitura do dado contido no endereço *addr* e o armazena em *data.* A leitura é realizada segundo o modo *random read*, isto é, inicialmente é realizada uma escrita no contador interno e, em seguida, a leitura propriamente dita;
- `WriteByte(addr, byte data)`: realiza a escrita do *byte* `data` na posição `addr`, segundo o modo *byte write;*
- `ReadBlock(addr, byte *data, word dataSize)`: lê `dataSize` *bytes* da memória a partir do endereço `addr` e armazena o resultado em `data`.
- `WriteBlock(addr, byte *data, word dataSize)`: escreve `dataSize` *bytes* contidos no vetor `data` a partir do endereço `addr`.
- `Test()`: realiza leituras e escritas para verificar funcionamento da memória. Retorna `ERR_OK` caso o componente responda corretamente.

Testando a memória

A tabela 1 de [4] sugere um algoritmo a fim de testar todas as posições da memória. Esse algoritmo consiste em, para cada uma das 2048 posições da memória, escrever e ler os valores 0x01, 0x02, 0x04, …, 0x40 e 0x80. De maneira prática, testaremos todos os 8 bits da cada posição da memória. A implementação deste teste é realizada na função `test_memory()`, representada abaixo:

Programa 1: função `test_memory()` que testa todos os bits da memória.

```
int test_memory() {
    int address;
    byte data_byte = 0, data_read;
    for (address = 0; address < 2048; address++) {

      for (data_byte = 1; data_byte != 0; data_byte = data_byte << 1) {

           EE241_WriteByte(address, data_byte);
           EE241_ReadByte(address, &data_read);
```

```
            if (data_byte != data_read)
                return 0; /* Dado lido diferente do escrito */
        }
    }
    /* Todos os dados foram escritos e lidos corretamente */
    return 1;
}
```

Enfim, testamos também a escrita de uma cadeia de caracteres em uma posição aleatória da memória:

Programa 2: testa escrita e leitura de um bloco.

```
char *string_test = “EA076 S2”, buffer_read[10];
memset (buffer_read, 0, 10);

EE241_WriteBlock(0x212, string_test, strlen(string_test));
EE241_ReadBlock(0x212, buffer_read, strlen(string_test));

/* Escreve resultado no LCD */
if (strcmp(buffer_read, string_teste) == 0)
    send_string(“OK”);
else send_string(“Erro”);
```

Implementando uma aplicação com LCD, conversor AD e EEPROM

Uma vez que a memória foi corretamente configurada e testada, um exemplo prático de seu uso pode ser implementado. Neste laboratório, desenvolvemos uma aplicação que:

(i) Espera o usuário apertar a tecla 6 do teclado para iniciar a amostragem do conversor AD, ligado a um sensor de temperatura, com uma periodicidade de 1 segundo. Cada amostra é armazenada em uma posição da memória. Os valores da soma, máximo e mínimo são recalculados a cada nova amostra. As teclas 3, 4, 5 e 8 são desabilitadas até o fim da amostragem;

(ii) A amostragem continua até o usuário apertar a tecla 7. Quando a tecla 7 é apertada, a amostragem é interrompida e o usuário pode pressionar as teclas 3, 4,5, 6 ou 8;

(iii) A tecla 6 reinicia o processo (i);

(iv) A tecla 3 mostra no LCD a média das amostras obtidas, a 4, o valor máximo das amostras e 5, o mínimo.

(v) A tecla 8 mostra no LCD os registros armazenados na memória. A impressão é realizada sequencialmente e ciclicamente a cada ‘*’ apertado, isto é, quando essa tecla é apertada a próxima posição de memória é lida e impressa no LCD. Após a última posição ser lida, o contador é reiniciado e a primeira amostra é novamente lida. O processo é interrompido quando a tecla ‘#’ for pressionada.

O techo de código abaixo representa o processo detalhado em (i). Antes do `while`, ocorre a inicialização de algumas variáveis que armazenarão a soma, valor mínimo, máximo, número de amostras lidas e o endereço da memória em que a amostra deve ser armazenada. A periodicidade de 1 segundo é realizada através da função `wait_n_interruptions`, explicada no experimento 2, que espera a quantidade de interrupções (cada interrupção é gerada a cada 60µs) passada como parâmetro. A função `getDigitalTension()` lê um valor de 16 *bits* do conversor analógico-digital. Para obter os *bytes* relativos à temperatura, foi implementada uma estrutura do tipo `union` chamada

`Temperatura`, com dois campos: um *float* e um vetor de *bytes*. A vantagem deste tipo de estrutura é que os mesmos *bytes* são compartilhados pelos dois campos. Dessa forma, quando modificamos um deles, estamos modificando o outro automaticamente.

Programa 3: trecho do programa que realiza as amostragens.

```
float sum = 0, min = 5000, max = 0;
int count = 0;
EE241_Address initial_address = 0;

/* Amostragem dos valores ateh tecla 7 for apertada. */
while (read_keys() != '7') {

    uint16_t tension = getDigitalTension();
    float tension_f;
    union Temperatura temp;

    /* Transforma a tensao em temperatura */
    tension_f = (float)tension * 3300/65536;
    temp.temp_as_float = (tension_f - 600)/10;

    /* Escreve na próxima posicao de memoria */
    EE241_WriteBlock(initial_address, temp.temp_as_bytes,sizeof(float));
    temp.temp_as_float = 0;
    /* Apenas para conferir resultado. */
    memset(temp.temp_as_bytes, 0, sizeof(float));
    EE241_ReadBlock(initial_address, temp.temp_as_bytes, sizeof(float));

    /* Atualiza contadores */
    sum += temp.temp_as_float;
    initial_address += sizeof(float);
    count++;

    /* Atualiza maximo e minimo */
    if (temp.temp_as_float > max)
      max = temp.temp_as_float;

    if (temp.temp_as_float < min)
      min = temp.temp_as_float;

    /* Espera 1s; */
    wait_n_interruptions(16667);
}
```

Uma vez apertada a tecla 7, a impressão dos resultados é habilitada. O trecho de código relativo às teclas 3, 4, 5 e 6 é apresentado abaixo. As três primeiras teclas utilizam as variáveis `sum`, `min`, `max` e `count` calculadas no trecho anterior. A tecla 6 reinicia a amostragem através da mudança da variável `restart`.

Programa 4: trecho do programa que mostra resultados no LCD.

```
button_pressed = read_keys();

/* Reinicia amostragem */
if (button_pressed == '6')
    restart = 1;
```

```
/* Media, minimo ou maximo */
if (button_pressed == '3' || button_pressed == '4' ||
      button_pressed == '5') {

    float data_print;
    char *text, buffer[6];
    memset(buffer, 0, 6);

    /* Seleciona valor a ser impresso */
    if (button_pressed == '3') {
      data_print = sum/count;
      text = "Media = ";
    } else if (button_pressed == '4') {
      data_print = max;
      text = "Max. = ";
    } else {
      data_print = min;
      text = "Min. = ";
    }

    /* Calcula parte decimal da temperatura */
    float decimal = data_print - (int) data_print;
    /* Imprime resultado com duas casas decimais. */
    sprintf(buffer, "%d.%02d", (int) data_print, (int) (decimal * 100));

    /* Manda para o LCD. */
    send_cmd(0x1, 26);
    send_string(text);
    send_string(buffer);
}
```

Enfim, quando a tecla 8 é pressionada, os valores contidos na memória são impressos sequencialmente e ciclicamente no LCD. A cada tecla '*' pressionada, a tela é atualizada e, após a última posição da memória ser mostrada, o contador é reiniciado para que a primeira posição seja acessada na próxima iteração. O processo é finalizado quando uma tecla '#' for detectada. O trecho abaixo representa tal funcionamento.

Programa 5: trecho do programa que imprime os conteúdos das posições de memória no LCD.

```
/* Imprimir posicoes de memoria */
if (button_pressed == '8') {

/* Imprime primeira posicao. */
union Temperatura temp;
char buffer_data[6], buffer_address[6];

/* Inicializa buffers que serao impressos no LCD */
memset(buffer_data, 0, 6);
memset(buffer_address, 0, 6);

/* Le primeira posicao da memoria */
EE241_Address aux_address = 0x0;
temp.temp_as_float = 0;
```

```
memset(temp.temp_as_bytes, 0, sizeof(float));
EE241_ReadBlock(aux_address, temp.temp_as_bytes, sizeof(float));

/* Prepara buffers para impressao. */
/* Calcula a parte decimal do numero em ponto flutuante */
float decimal = temp.temp_as_float - (int) temp.temp_as_float;
sprintf(buffer_data, "%d.%02d",
                     (int) temp.temp_as_float, (int) (decimal * 100));

sprintf(buffer_address, "0x%x", aux_address);

/* imprime buffers no LCD */
send_cmd(0x1, 26);
send_string(buffer_address);
send_string(" - ");
send_string(buffer_data);

/* Imprime circularmente posicoes ate # for pressionado. */
do {

    button_pressed = read_keys();

    if (button_pressed == '*') {

      /* Avanca na memoria */
      aux_address += sizeof(float);

      if (aux_address >= initial_address)
           aux_address = 0;

      /* Le float da memoria (4bytes). */
      temp.temp_as_float = 0;
      memset(temp.temp_as_bytes, 0, sizeof(float));
      EE241_ReadBlock(aux_address, temp.temp_as_bytes, sizeof(float));

      /* Constroi strings para impressao. */
      float decimal = temp.temp_as_float - (int) temp.temp_as_float;
      sprintf(buffer_data, "%d.%02d",
                       (int) temp.temp_as_float, (int) (decimal * 100));
      sprintf(buffer_address, "0x%x", aux_address);

      /* Imprime posicao atual de memoria. */
      send_cmd(0x1, 26);
      send_string(buffer_address);
      send_string(" - ");
      send_string(buffer_data);

    }

} while (button_pressed != '#');
```

Referências

[1] KL25 Sub-Family Reference Manual – Freescale Semiconductors, disponível em ftp://ftp.dca.fee.unicamp.br/pub/docs/ea871/ARM/KL25P80M48SF0RM.pdf

[2] ATMEL Two-wire Serial EEPROM, disponível em ftp://ftp.dca.fee.unicamp.br/pub/docs/ea076/datasheet/AT24C164.pdf

[3] McuOnEclipse Release on SourceForge, disponível em http://sourceforge.net/projects/mcuoneclipse/files/PEx%20Components/

[4] Software Based Memory Testing, disponível em http://www.esacademy.com/en/library/technical-articles-and-documents/miscellaneous/software-based-memory-testing.html