O Protocolo One Wire (1-Wire)

Versão 1.0 (05/01/2021)

O protocolo 1-Wire, como o próprio nome diz, especifica uma forma de comunicação usando apenas um fio. Ele foi desenvolvido pela Dallas Semiconductor que foi comprada pela Maxxim. Na verdade, faz uso de dois condutores: um terra e uma linha de dados. Neste caso a alimentação é “roubada” da linha de dados enquanto ela está em nível alto. Alguns dispositivos 1-Wire disponibilizam um terceiro pino para alimentação. para a comunicação com o periférico. Este protocolo foi criado para a comunicação com dispositivos simples e que tenham pouca transação de dados. Sua velocidade máxima é de 16,4 kbits/s (aproximadamente 2 kbytes/s). Exemplos típicos são termômetros, relógios (RTC), coletores de dados, autenticadores etc.

Observações: Este apêndice surgiu da necessidade de se usar o termômetro DS18B20. Então aqui não é feita uma abordagem completa do protocolo, mas sim dos recursos para se usar este termômetro. Ele serve como introdução para se iniciar o acesso aos outros dispositivos 1-Wire. Além disso, os exemplos aqui apresentados privilegiam a clareza e não o desempenho. Eles foram escritos para o LaunchPad com o MSP430 F5529, operando com seu relógio típico que é de 1.048.576 Hz.

O.0. Quero Usar um Dispositivo 1-Wire e Não Pretendo Ler Todo esse Apêndice

Escrever

O.1. Protocolo 1-Wire

O conceito do protocolo 1-Wire é semelhante ao do I2C, ao fazer uso da sinalização via coletor ou dreno aberto para conseguir a comunicação bidirecional. A Figura O.1 apresenta uma típica linha 1-Wire. Quando os dispositivos estão inativos, seus transistores de saída estão cortados e, neste caso, a linha vai para nível alto, graças ao resistor de pull-up. Note que, mesmo no estado inativo, os dispositivos continuam a monitorar o barramento. Qualquer um deles ao "ligar" seu transistor de saída faz uma sinalização para o outro dispositivo, pois a linha vai para nível baixo. O resumo da sinalização é a seguinte:

- 1-Wire = alto: todos os dispositivos estão com seus transistores cortados e
- 1-Wire = baixo: pelo menos um dispositivo está com seu transistor conduzindo.

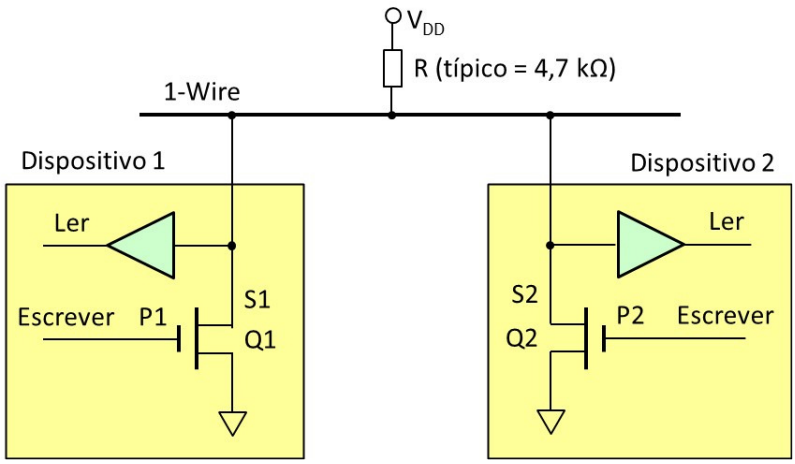


Figura O.1 Conexão de dois dispositivos 1-Wire.

A um barramento 1-Wire podem se conectar diversos dispositivos. Um deles, obrigatoriamente, será o mestre e, por conveniência, vamos chamar os demais de escravos. Os escravos somente respondem quando o mestre os comanda. O leitor já deve estar desconfiando de que a única forma de se fazer alguma sinalização usando um único condutor é por pulsos em nível baixo. É isto mesmo, são especificados pulsos para a inicialização (ressete) e pulsos para a transferência de dados, como será visto no próximo tópico.

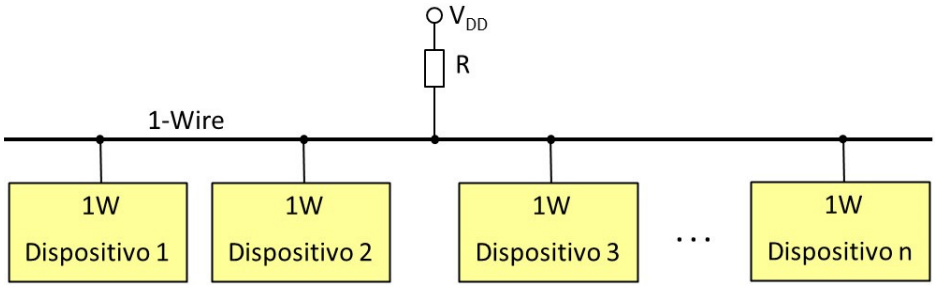


Figura O.2. Típico barramento 1-Wire.

Deve-se notar que a única ação que um disposto realiza sobre o barramento 1-Wire é a de puxar a linha para nível baixo. Daí surgem dois estados para os dispositivos:

- Dispositivo transmitindo (TX): ele puxando a linha para nível baixo, ou seja, seu transistor de saída está conduzindo.
- Dispositivo recebendo (RX): ele está com seu transistor de saída cortado e, neste caso, consegue perceber quando algum outro dispositivo puxa a linha para baixo.

Antes de seguirmos adiante, vamos apresentar como é feita a alimentação do dispositivo 1-Wire usando apenas a linha de dados. Na Figura O.3, note que enquanto a linha está em nível alto, o capacitor se carrega em direção a V_{DD}. Se a linha ficar um bom tempo em nível alto, podemos dizer que a tensão sobre o capacitor é igual a V_{DD} (subtraído da queda de tensão sobre o capacitor). O diodo impede que o capacitor se descarregue quando a linha vai para nível baixo. Enquanto a linha a linha está em nível baixo, o capacitor sustenta a alimentação do dispositivo 1-Wire. Mas ele se descarrega nesta operação, então a linha não pode ficar muito tempo em nível baixo. Por outro lado, tais dispositivos devem se caracterizar por um baixo consumo. Tais dispositivos são preparados para entre em estado de ressete assim que surge a alimentação.

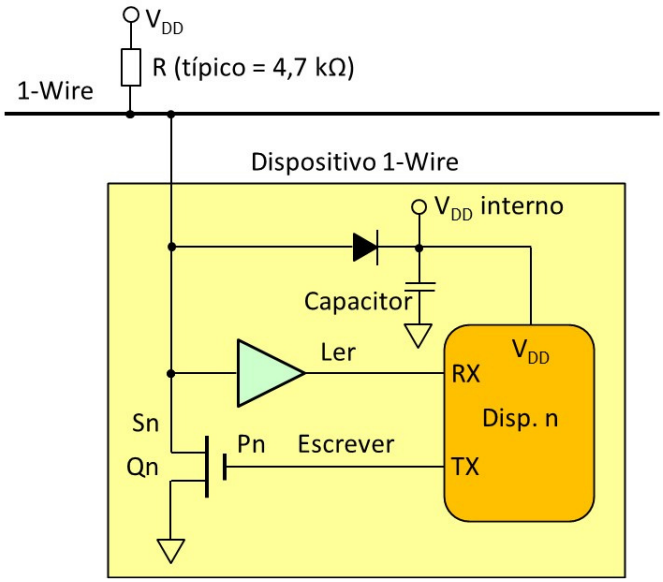


Figura O.3. Alimentação parasitária de um dispositivo 1-Wire.

O.2. Sinalização com o Protocolo 1-Wire

Como já foi adiantado, toda a sinalização no protocolo 1-Wire é feita por pulsos com larguras mínimas pré-definidas. Vamos descrevê-las sob a óptica do mestre. São cinco sinalizações, pois estamos separando a leitura e a escrita.

- Ressete (somente gerado pelo mestre);
- Escrita de bit igual a zero;
- Escrita de bit igual a um;
- Leitura de bit igual a zero e
- Leitura de bit igual a um

O.2.1. Ressete e Pulso e Presença

Todo acesso 1-Wire é iniciado com uma sequência de inicialização que consiste em um pulso de ressete gerado pelo mestre e de um sinal de presença, gerado pelos dispositivos 1-Wire presentes no barramento. A Figura O.4 ilustra a temporização desta sequência de inicialização, onde a sigla V_{PU} indica a tensão conseguida com o resistor de pull-up. A linha preta indica a ação do resistor de pull-up, a linha vermelha a ação do mestre puxando a linha para baixo e a linha azul a ação do periférico, também puxando a linha para baixo.

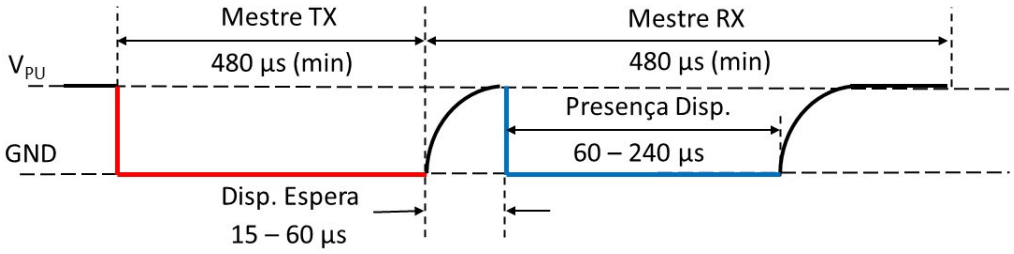


Figura O.4. Diagrama de tempo da sequência de inicialização do barramento 1-Wire

A sequência de inicialização começa com o mestre gerando o pulso de ressete. Para tanto, ele abaixa a linha (modo TX) por um tempo mínimo de 480 µs. Decorrido o tempo mínimo o mestre libera a linha, ou seja, entra no modo recepção (RX). Por ação do resistor de pull-up, a linha vai para nível alto. Depois de um tempo que varia de 15 a 60 µs, os dispositivos presentes no barramento ligam seus transistores de saída (Modo TX) e com isso a linha vai para nível baixo. Assim devem permanecer por um período de 60 a 240 µs. Depois disso, liberam a linha. Caso não existam dispositivos presentes ou prontos para se comunicar, a linha permanece em nível alto.

O mestre que gerou o pulso de ressete monitora a linha para ver se ela vai para nível baixo, o que indica a presença de dispositivos prontos para se comunicar. Caso o mestre detecte o pulso de presença é porque existe pelo menos um dispositivo pronto para se comunicar. Cada dispositivo 1-Wire possui um endereço de 64 bits previamente programado na sua fabricação. Com um procedimento especial, que será visto adiante, o mestre pode determinar o endereço de cada um dos dispositivos presentes no barramento 1-Wire.

O.2.2. Escrita de Zero ou Um

A janela de tempo (mínima) para o mestre escrever o bit 1 ou o bit 0 no barramento é de 60 µs. Como pode ser visto na Figura O.5, para o mestre enviar um zero, ele mantém a linha em nível baixo durante um período de 60 a 120 µs. Para enviar o bit um, ele abaixa a linha por um período maior que 1 µs e depois a libera; ela então vai para nível alto graças ao resistor de pull-up. Entre quaisquer dois bits enviados, deve existir uma separação mínima de 1 µs. Podemos calcular a velocidade, pois se é consumido 61 µs para cada bit, a máxima taxa de transmissão é de 1/61 MHz, o que resulta em 16,4 kbits/s.

É interessante olharmos com mais atenção as "janelas dos bits". Entre duas dessas janelas deve haver a separação mínima de 1 µs, porém não há limite máximo. Isto significa que o mestre pode pausar o envio de bits quando quiser. Para o mestre escrever 0, a linha deve ficar em nível baixo por, pelo menos, 60 µs e, por outro lado, nunca deve ultrapassar o limite de 120 µs. Se a linha ficar muito tempo em nível baixo, pode se confundir com o sinal de ressete (se bem que o pulso de ressete é 480 µs d). Para enviar 0, o mestre deve colocar a linha em nível baixo por 1 µs e, recomendamos que não ultrapasse 15 µs. Na verdade, na escrita de 1, o final da janele de bit se confunde com a linha inativa.

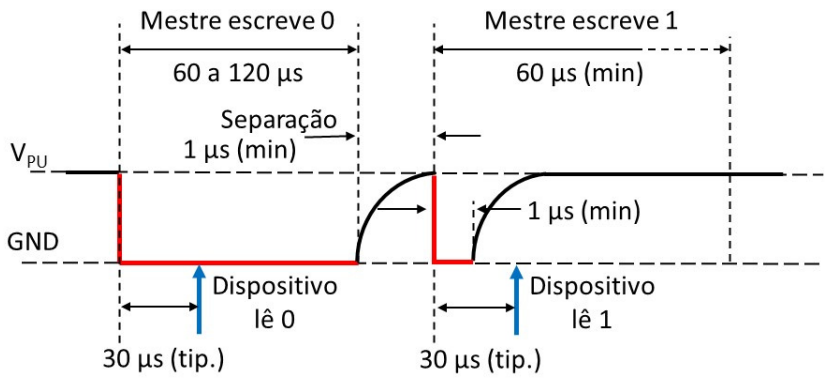


Figura O.5. Diagrama de tempo para os casos em que o mestre escreve 0 e depois escreve 1 no barramento 1-Wire.

A Figura O.5 usa setas verticais para indicar o instante em que o dispositivo lê o barramento. Este instante está marcado como típico, pois pode variar para dispositivos diferentes. Por isso, é importante consultar o manual do dispositivo 1-Wire que se pretende usar. Para o caso particular do DS18B20, que é tomado como referência para este apêndice, a leitura acontece dentro de uma janela de 45 µs, cujo início está µs após o flanco de descida que marca o início da janela de um 1 bit. A Figura O.6 esclarece melhor esta situação.

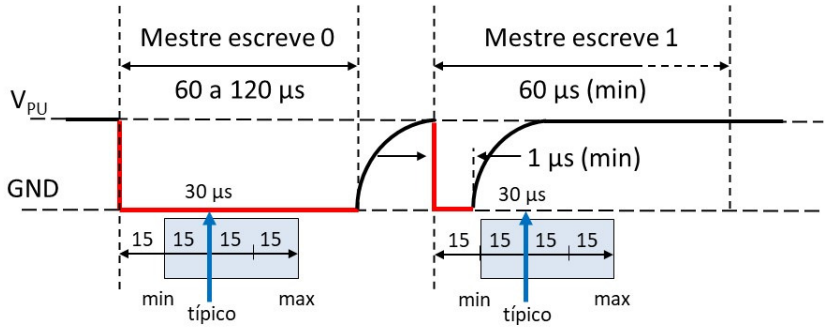


Figura O.6. Diagrama de tempo indicando a janela de leitura do DS18B20 e o valor típico de 30 µs.

O.2.3. Leitura de Zero ou Um

Vamos agora para o problema inverso que é o caso de o mestre ler os bits que são enviados pelo dispositivo 1-Wire. Neste caso, o mestre comanda o instante em que o escravo deve enviar cada bit. Este comando é caracterizado por um pulso em nível baixo, que deve ter a duração mínima de 1 µs. Em seguida o mestre libera a linha e o escravo, então, escreve seu bit. Durante esta escrita do escravo, o mestre amostra a linha para identificar o valor do bit. A Figura O.7 apresenta um diagrama de tempo que tomou como referência o DS18B20.

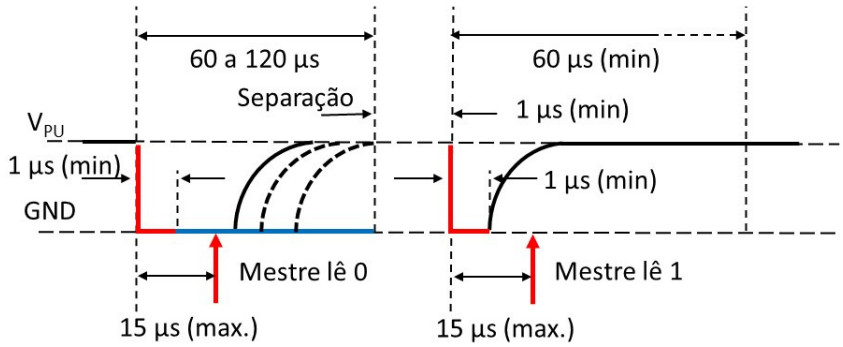


Figura O.7. Diagrama de tempo para os casos em que o mestre lê o bit enviado pelo escravo presente no barramento 1-Wire.

Nesta figura deve ser notado que o mestre precisa fazer a leitura dentro dos primeiros 15 µs após o início da janela. No caso particular do DS18B20, após este intervalo não há garantia de que o bit ainda esteja presente no barramento.

Com isto terminamos a parte de sinalização do barramento 1-Wire e vamos agora fazer um breve ensaio de como gerar estes sinais usando o MSP430.

O.3. Geração de Sinais 1-Wire com o MSP430

Como já dissemos, este tópico é escrito considerando a disponibilidade da LaunchPad que em sua configuração inicial tem o MCLK e o SMCLK em 1.048.576 Hz. Os exemplos e medições são feitas com termômetro DS18B20. Será usado o pino P2.5 para construir o barramento.

Iniciamos então com a configuração deste pino. São duas exigências, pois o pino precisa ir para nível baixo para gerar o bit 0 e entrar em alta impedância para gerar o bit 1. Uma solução bem simples é mostrada na Tabela O.1, onde se vê que basta colocar P2OUT.5 = 0 e P2REN.5 = 0 para que o controle seja feito apenas pelo o registrador de direção.

Tabela O.1. Controle de P2.5 para acionar o barramento 1-Wire

P2DIR.5	P2OUT.5	P2REN.5	Estado do pino
1	0	0	P2.5 = 0
0	0	0	P2.5 = Hi Z

O próximo item é sobre a temporização dos sinais no barramento 1-Wire. Precisamos gerar sinais com intervalos definidos e a primeira ideia que nos vêm à cabeça é usar um dos timers para esta tarefa, porém existe uma dificuldade. Na configuração que está sendo usada, o relógio mais elevado é de 1.048.576 Hz, ou seja, aproximadamente 1 MHz. Será difícil usar tal relógio para gerar intervalos de alguns microssegundos. O relógio da CPU também está perto de 1 MHz, o que significa que cada instrução assembly consome, pelo menos, 1 µs. Não devemos usar interrupções porque o desvio para uma ISR consome, no mínimo, 6 µs.

Tudo isso é colocado para justificar que vamos gerar os intervalos por laços de programas. Se aumentássemos a frequência do MCLK e SMCLK para 8 MHz ou 16 MHz, então poderíamos usar o recurso de um timer. Os exercícios propostos sugerem desafios neste sentido.

O.3.1. Experimentação na Geração de Pulsos com o MSP430

Se é para gerar atrasos por software, poderíamos lançar mão da função `__delay_cycles()`. Porém não faremos isto. Queremos deixar tudo mais interessante e gerar nossos próprios atrasos. Vamos iniciar experimentando algumas alternativas para gerar atrasos, fazermos medições e decidiremos pela melhor. Isso vai nos dar uma primeira noção de quanto tempo cada linha em C consome.

Experimento 1:
O programa abaixo fica num laço infinito, alternando o pino P2.5 entre os dois estados possíveis. É esperado que tempo em que o pino fica em nível alto seja maior porque ele vai incluir o fechamento do laço.

```
#include <msp430.h>
void main(void){
      WDTCTL = WDTPW | WDTHOLD;     // stop watchdog timer
    P2DIR &= ~BIT5;      //P2.5 = entrada
    P2OUT &= ~BIT5;      //P2.5 = 0
    P2REN &= ~BIT5;      //Desabilita Pull/up/down
    while(1){
        P2DIR |=  BIT5; //P2.5 = Zero
        P2DIR &= ~BIT5; //P2.5 = Hi-Z
    }
}
```

Ao rodarmos este programa, obtemos o resultado apresentado na Figura O.8. Considerando que o MCLK é igual a 1.048.576 Hz, facilmente calculamos que o 4,7 µs corresponde a $5T_{MCLK}$ e que 7,6 µs corresponde a 8 T_{MCLK}. Em suma, cada instrução que opera com P2DIR consome 5 T_{MCLK} e o fechamento do laço consome 3 T_{MCLK}. Percebemos então que o menor pulso que conseguimos gerar por software é de, aproximadamente, 5 µs.

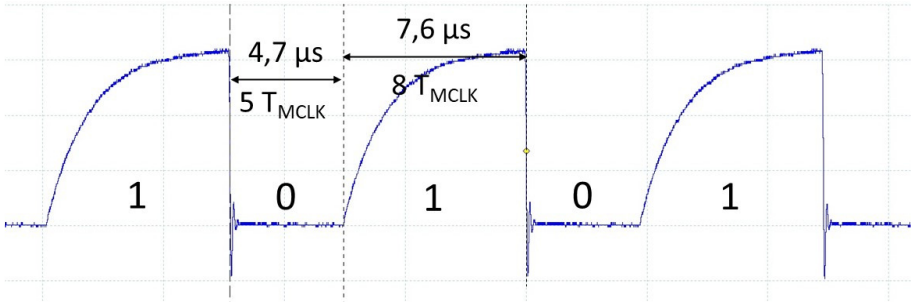


Figura O.8. Diagrama de tempos resultante do Experimento 1.

Experimento 2:

Surge então a ideia de usarmos diretamente o assembly. A listagem abaixo apresenta esta sugestão construindo o laço usando o recurso de assembly inline. Ao rodarmos o programa, vamos constatar que o resultado é exatamente o mesmo da Figura O.8. Em outras palavras, o compilador do CCS é bastante eficiente e neste caso nada ganhamos a usar o assembly.

```
#include <msp430.h>
void main(void){
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    P2DIR &= ~BIT5;     //P2.5 = entrada
    P2OUT &= ~BIT5;     //P2.5 = 0
    P2REN &= ~BIT5;     //Desabilita Pull/up/down
    asm("LB: BIS.B  #0x32,&0x205"); //BIS.B  #BIT5,&P2DIR, P2.5 = 0
    asm("    BIC.B  #0x32,&0x205"); //BIS.B  #BIT5,&P2DIR, P2.5 = Hi-Z
    asm("    JMP    LB");           //Fechar o laço
}
```

Experimento 3:

No Experimento 2, as linhas `P2DIR |= BIT5;` e `P2DIR &= ~BIT5;` não deixam claro o que está sendo realizado. Seria melhor usarmos funções como nomes mais claros, como é sugerido na listagem abaixo, onde:

`ow_pin_low()` → função para colocar o pino em nível baixo e

`ow_pin_hiz()` → função para colocar o pino em alta impedância (Hi-Z).

```
#include <msp430.h>

void ow_pin_low(void) {P2DIR |=  BIT5; }   //P2.5 = Zero
void ow_pin_hiz(void) {P2DIR &= ~BIT5; }   //P2.5 = Hi-Z

void main(void){
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    P2DIR &= ~BIT5;     //P2.5 = entrada
    P2OUT &= ~BIT5;     //P2.5 = 0
    P2REN &= ~BIT5;     //Desabilita Pull/up/down
    while(1){
        ow_pin_low(); //P2.5 = Zero
        ow_pin_hiz(); //P2.5 = Hi-Z
    }
}
```

Ao rodarmos o programa acima vemos o custo de se empregar funções. Foram adicionados perto de 10 T_{MCLK} na largura de cada pulso. Tal situação vai dificultar ainda mais precisão dos atrasos que precisamos gerar. Porém, existe uma alternativa interessante que é apresentada no Experimento 4.

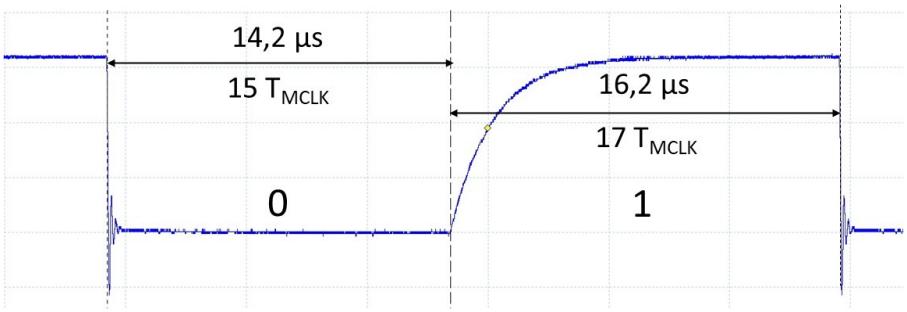


Figura O.9. Diagrama de tempos resultante do Experimento 3.

Experimento 4:
Em C existe o recurso de se declarar uma função do tipo `inline`. Uma função deste tipo, na verdade, não é usada como uma função. Ela se parece mais com uma Macro e ao invés de colocar uma chamada, o compilador substitui a linha que a chama pela função completa. Ao rodarmos o programa listado abaixo, vamos obter exatamente o resultado do Experimento 2, mas com a vantagem de que o programa ficou mais legível.

O recurso `inline` deve ser usado com parcimônia e só faz sentido quando a função é relativamente simples.

```
#include <msp430.h>

inline void ow_pin_low(void) {P2DIR |=  BIT5; }   //P2.5 = Zero
inline void ow_pin_hiz(void) {P2DIR &= ~BIT5; }   //P2.5 = Hi-Z

void main(void){
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    P2DIR &= ~BIT5;     //P2.5 = entrada
    P2OUT &= ~BIT5;     //P2.5 = 0
    P2REN &= ~BIT5;     //Desabilita Pull/up/down
    while(1){
        ow_pin_low(); //P2.5 = Zero
        ow_pin_hiz(); //P2.5 = Hi-Z
    }
}
```

O.3.2. Sinalização 1-Wire com o MSP430

Vamos agora trabalhar na geração dos pulsos e das ações necessários para que o MSP430 possa controlar um barramento 1-Wire. As medições aqui apresentadas foram realizadas com o emprego do termômetro DS18B20. Iniciamos com a sinalização de inicialização.

Experimento 5: Geração do Sinal de Inicialização

A sinalização da inicialização do barramento 1-Wire está apresentada na Figura O.4. Ela envolve períodos longos o suficiente para usarmos um timer, entretanto, como já dissemos anteriormente, vamos fazer tudo com atrasos de software. É preciso gerar um pulso em nível baixo de, pelo menos, 480 µs e depois "liberar" o barramento por mais outros 480 µs para que os periféricos sinalizem suas presenças puxando a linha para nível baixo. O código proposto para este experimente está listado abaixo.

```
// Exp5.c
#include <msp430.h>

#define TRUE  1
#define FALSE 0

// Funções para leds e osciloscópio
inline void SCOPE(void)  {  P2OUT |=  BIT4;  }  //P2.4=1
inline void scope(void)  {  P2OUT &= ~BIT4;  }  //P2.4=0
inline void Scope(void)  {  P2OUT ^=  BIT4;  }  //P2.4=invertido
inline void led_VM(void) {  P1OUT |=  BIT0;  }  //VM=aceso
inline void led_vm(void) {  P1OUT &= ~BIT0;  }  //VM=apagado
inline void led_Vm(void) {  P1OUT ^=  BIT0;  }  //VM=invertido
inline void led_VD(void) {  P4OUT |=  BIT7;  }  //VD=aceso
inline void led_vd(void) {  P4OUT &= ~BIT7;  }  //VD=apagado
inline void led_Vd(void) {  P4OUT ^=  BIT7;  }  //VD=invertido

// Constantes para os atrasos
#define OW_RST1 88
#define OW_RST2 10

// Funções para 1-Wire
char ow_rst(void);
inline char ow_pin_rd (void) {return (P2IN&BIT5)>>5;}   //Ler P2.5
inline void ow_pin_low(void) {P2DIR |=  BIT5; }         //P2.5 = Zero
inline void ow_pin_hiz(void) {P2DIR &= ~BIT5; }         //P2.5 = Hi-Z
void gpio_config(void);                                 //Configurar GPIO

void main(void){
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    gpio_config();
    while(1){
        if (ow_rst() == TRUE) { led_vm(); led_VD();    }   //OK
        else                  { led_VM(); led_vd();    }   //OK
        __delay_cycles(1000);
    }
}

// Gerar Reset e ler resposta do perinférico
// TRUE --> OK FALSE --> Não OK
char ow_rst(void){
```

```
  char i,aux;
  SCOPE();
  ow_pin_low();
  for (i=0; i<OW_RST1; i++) __no_operation();
  ow_pin_hiz();
  for (i=0; i<OW_RST2; i++) __no_operation();
  aux=ow_pin_rd();
  Scope();  //Sinalização para o osciloscópio
  Scope();  //Sinalização para o osciloscópio
  for (i=0; i<(OW_RST1-OW_RST2); i++) __no_operation();
  scope();
  return aux^BIT0;  //1 = presente
}

// Inicialização do GPIO
// P2.5 = 1-Wire
// P2.4 = Auxiliar para Osciloscópio
void gpio_config(){
    P2DIR |= BIT5;      //P2.5 = saída
    P2OUT &= ~BIT5;     //P2.5 = 0
    P2REN &= ~BIT5;     //Desabilita Pull/up/down

    P2DIR |=  BIT4; P2OUT &= ~BIT4; //Scope
    P1DIR |= BIT0;  P1OUT &= ~BIT0; //Led VM
    P4DIR |= BIT7;  P4OUT &= ~BIT7; //Led VD
}
```

O código recém apresentado está um pouco longo porque traz várias funções para facilitar a operação, além de um sinal especial (P2.4) para facilitar a sincronização com o osciloscópio e permitir a medição da largura da sinalização realizada. Das funções apresentadas, apenas duas são importantes para a sinalização:

- `char ow_rd_pin(void)` → que retorna o estado do barramento 1-Wire, que neste caso está construído com o pino P2.5) e
- `char ow_rst(void)` → que faz a inicialização do barramento e retorna TRUE caso algum dispositivo responda com o sinal de presença.

A função `ow_rst()` é simples, ela coloca a linha em nível baixo, e depois libera a linha e faz a leitura do pulso de presença. O tempo em que linha fica em nível baixo é ditado pela constante `OW_RST1`. Após liberar a linha, a função espera um intervalo ditado pela constante `OW_RST2`, faz a leitura do pulso de presença e depois ainda aguarda um período proporcional a `OW_RST1 – OW_RST2` e retorna falso ou verdadeiro em função do valor do pulso de presença. O uso dessas constantes vai facilitar a adaptação para o caso de se alterar o relógio da CPU (MCLK). Todos os atrasos foram gerados com laços de programa executando a função `__no_operation()`, que corresponde à instrução `NOP` do assembly.

A Figura O.10 apresenta a imagem colhida com o osciloscópio onde se pode comprovar a temporização da sinalização. O estado do barramento 1-Wire é dado pela linha azul. A linha vermelha é um sinal gerado pela função para facilitar a sincronização e indicar o momento da leitura do pulso de presença. Intencionalmente, pulsos estão um pouco acima do mínimo especificado.

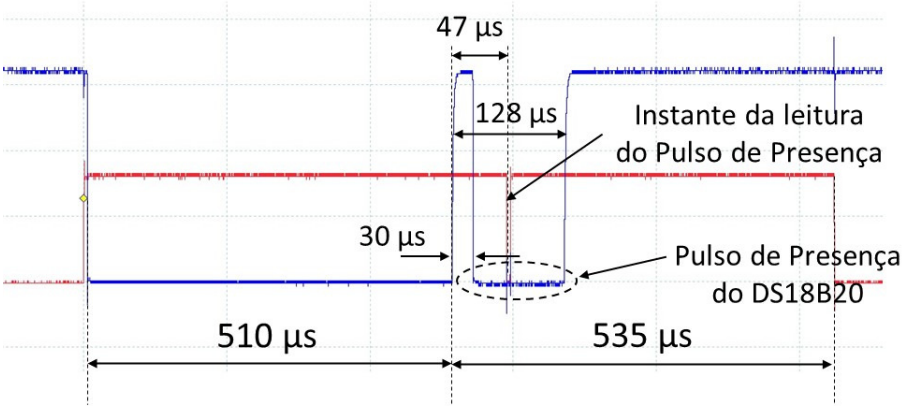


Figura O.10. Sinal de inicialização gerado pelo código proposto.

Experimento 6: Escrita de 0 e 1
Para verificar a sinalização de 0 e 1 foi preparado o Experimento 6, cuja listagem está apresentada abaixo. A listagem abaixo não compila diretamente, pois alguns trechos, por serem idênticos aos apresentados na listagem anterior, estão omitidos. Entretanto, com facilidade o leitor pode compor o experimento completo.

De importante, temos as duas funções `ow_wr_0()` e `ow_wr_1()`, que são bem simples e dispensam maiores explicações. Na entrada e na saída dessas funções foram colocadas marcações para permitir a sincronização com o osciloscópio e saber, quando se retorna da função que escreve 1, já que seu final se confunde com a linha inativa. Na Figura O.11, essa marcação é dada pela linha vermelha. É claro que existe um atraso causa pela chamada das funções `SCOPE()` e `scope()`.

```
// Exp6.c
#include <msp430.h>

// Constantes para os atrasos
#define OW_RST1 88
#define OW_RST2 10
#define OW_WR   10

// Funções para 1-Wire
void ow_wr_1(void);
void ow_wr_0(void);
inline void ow_pin_low(void) {P2DIR |=  BIT5; }         //P2.5 = Zero
```

```
inline void ow_pin_hiz(void) {P2DIR &= ~BIT5; }          //P2.5 = Hi-Z

void main(void){
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    gpio_config();
    while(1){
        ow_wr_0();
        ow_wr_1();
        __delay_cycles(1000); //Facilitar sincr com osciloscópio
    }
}

// Escrever 1 no barramento One Wire
void ow_wr_1(void){
  char i;
  SCOPE();
  ow_pin_low();
  ow_pin_hiz();
  for (i=0; i<OW_WR; i++)    __no_operation();
  scope();
}

// Escrever 0 no barramento One Wire
void ow_wr_0(void){
  char i;
  SCOPE();
  ow_pin_low();
  for (i=0; i<OW_WR; i++)    __no_operation();
  ow_pin_hiz();
  __no_operation();
  scope();
}
```

Analisando a Figura O.11, podemos conferir a temporização do sinal gerado. A largura do pulso está, intencionalmente, acima do mínimo de 60 µs exigido pelo protocolo. A duração do pulso é ditada pela constante `OW_WR`. Em seu lugar poderia ser usada a constante `OW_RST2`, mas preferimos manter as constantes separadas para facilitar a tarefa do leitor, caso tenha que fazer alguma adaptação do código. A escrita do bit 0 está bem clara, marcada pela linha azul e teve a duração de 66 µs. Já para saber a duração da escrita do 1, como seu final se confunde com a linha inativa, precisamos fazer uso da linha vermelha do osciloscópio. A linha vermelha é gerada pelas funções denominadas scope e como o atraso de uma função deste tipo é de, aproximadamente, 5 µs, podemos também dizer que o pulso 1 também durou 66 µs. Na figura está também marcado o instante em que o DS18B20, tipicamente, deve fazer a leitura do bit. Para outros dispositivos 1-Wire pode ser que se faça necessário um pequeno ajuste.

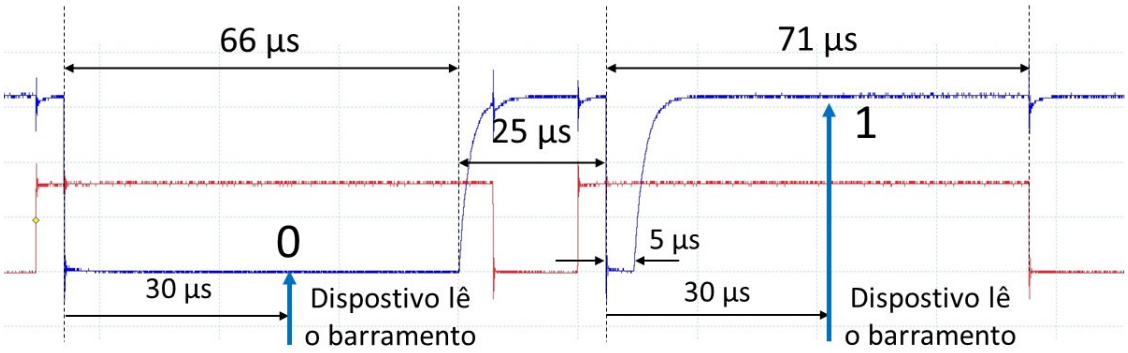


Figura O.11. Resultado da sinalização de 0 e 1 realizada pelo MSP430.

Experimento 7: Leitura dos bits 0 e 1

Este experimento deve ser um pouco mais cuidadoso, pois é necessário adequar o instante de leitura ao intervalo em que o dispositivo disponibiliza seu dado. A listagem abaixo faz sucessivas leituras do dispositivo 1-Wire para que se possa conferir a temporização. Para que o dispositivo 1-Wire gere alguns dados, é necessário o envio de diversos comandos para solicitar esses dados. Tais comandos serão explicados mais adiante. A listagem apresenta apenas as funções essenciais. Tudo o que foi truncado pode ser obtido nas listagens anteriores.

```
// Exp7.c
#include <msp430.h>

//Comandos para o DS18B20
#define DS_SKIP_ROM     0xCC  //Pulas enderaçamento
#define DS_CONVERT_T    0x44  //Iniciar conversão de temperatura
#define DS_RD_SCRATCH  0xBE   //Ler todo scratchpad (9 bytes)

// Constantes para os atrasos
#define OW_RST1 88
#define OW_RST2 10
#define OW_WR   10
#define OW_RD    7

// Funções para 1-Wire
char ow_rd_blk(char *vt, char qtd);
char ow_rd_byte(void);
char ow_rd_bit(void);
void ow_wr_byte(char dt);
```

```
void main(void){
    volatile char vetor[9];
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    gpio_config();
    while(1){
        if (ow_rst() == FALSE){     //Reset
            led_VM();               //Erro
            while(TRUE);            //Trava execução
        }
        else{
            ow_wr_byte(DS_SKIP_ROM);      //Ignorar endereçamento
            ow_wr_byte(DS_CONVERT_T);     //ComandoConverter temperatura
            __delay_cycles(1000000);      //Esperar 1 segundo
            ow_rst();                     //Reset
            ow_wr_byte(DS_SKIP_ROM);      //Ignorar endereçamento
            ow_wr_byte(DS_RD_SCRATCH);    //Comando Ler Scratchpad
            ow_rd_blk(vetor,9);           //Ler bytes do Scratchpad
        }
    }
}

// Ler um bloco de bytes pelo 1-Wire
char ow_rd_blk(char *vt, char qtd){
  char i;
  for (i=0; i<qtd; i++){
    vt[i]=ow_rd_byte();
  }
  return qtd;
}

// Ler um byte pelo 1-Wire
char ow_rd_byte(void){
  char i,dt=0;
  for (i=0; i<8; i++){
    dt=dt>>1;
    if ( ow_rd_bit() == BIT0)  dt |= 0x80;      //Ler 1-Wire
  }
  return dt;
}

// Ler 1 bit no barramento 1-Wire
char ow_rd_bit(void){
    char x,i;
    SCOPE();
    ow_pin_low();
    ow_pin_hiz();
    x=ow_pin_rd();  //Ler barramento 1-Wire
    Scope();        //Marcar momento da leitura
    Scope();        //Marcar momento da leitura
    for (i=0; i<OW_RD; i++)    __no_operation();
```

```
    scope();
    return x;
}

// Enviar um byte pelo 1-Wire
void ow_wr_byte(char dt){
  char i;
  for (i=0; i<8; i++){
    if ((dt&BIT0)==0) ow_wr_0();
    else              ow_wr_1();
    dt = dt>>1;
  }
}
```

A Figura O.12 apresenta a captura da tela do osciloscópio enquanto o MSP executava os ciclos de leitura do barramento 1-Wire. A temporização de referência é apresentada na Figura O.7. A janela para que o dispositivo envie um bit está em 73 µs, obedecendo ao valor mínimo que é de 60 µs. Devido à presença das funções scope(), na verdade, a janela deve estar perto de 68 µs. O instante de leitura acontece aproximadamente 19 µs após o flanco de descida que caracteriza o início da janela (na verdade, deve estar em 15 µs). Com todos esses experimentos, construímos o básico necessário para nos comunicarmos com barramento 1-Wire. Vamos agora apresentar um estudo resumido do termômetro DS18B20.

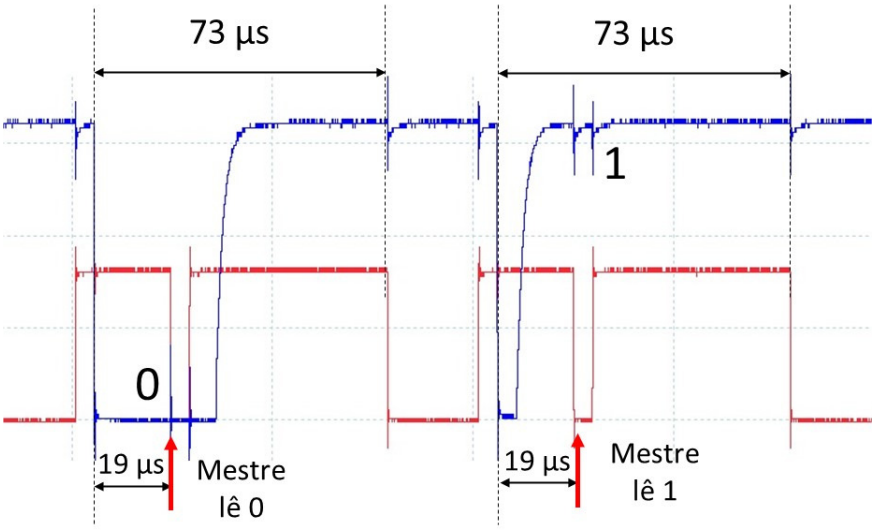


Figura O.12. Temporização da operação de leitura de 0 e 1, realizada pelo MSP430.

O.4. O Termômetro DS18B20

O DS18B20 é um termômetro digital barato e de baixíssimo consumo de energia, que fornece medidas de temperatura em graus Celsius, com resolução de 9 até 12 bits. Sua faixa de operação é de 3,0 até 5,5 V e o consumo de corrente em Standby é de 750 ηA. Além disso, oferece uma memória não volátil para registrar alarmes de temperatura abaixo da mínima ou acima da máxima especificada.

Em tempos de Covid, quando tanto se fala em transporte e armazenagem de vacina seguindo limites de temperatura, cada caixa de vacina pode incluir um desses termômetros, com os limites de temperatura previamente programados. Ao chegar no destino, basta consultar o termômetro para verificar se as restrições de temperatura foram obedecidas.

O.4.1. Descrição do DS18B20

Aqui é apresentado um modelo muito simplificado do dispositivo DS18B20. Este modelo é perfeitamente funcional para o que se propõe, porém, se for analisado a fundo, ele pode diferenciar de alguns detalhes do manual. Como mostrado na Figura O.13, além da interface 1-Wire e do controlador, este dispositivo possui os seguintes elementos:

- ROM de 64 bits com um código serial único e
- Registrador Scratchpad com 9 posições (9 bytes);
- Um sensor de temperatura;
- Uma memória EEPROM de 3 bytes e
- Um controlador com acesso ao barramento 1-Wire.

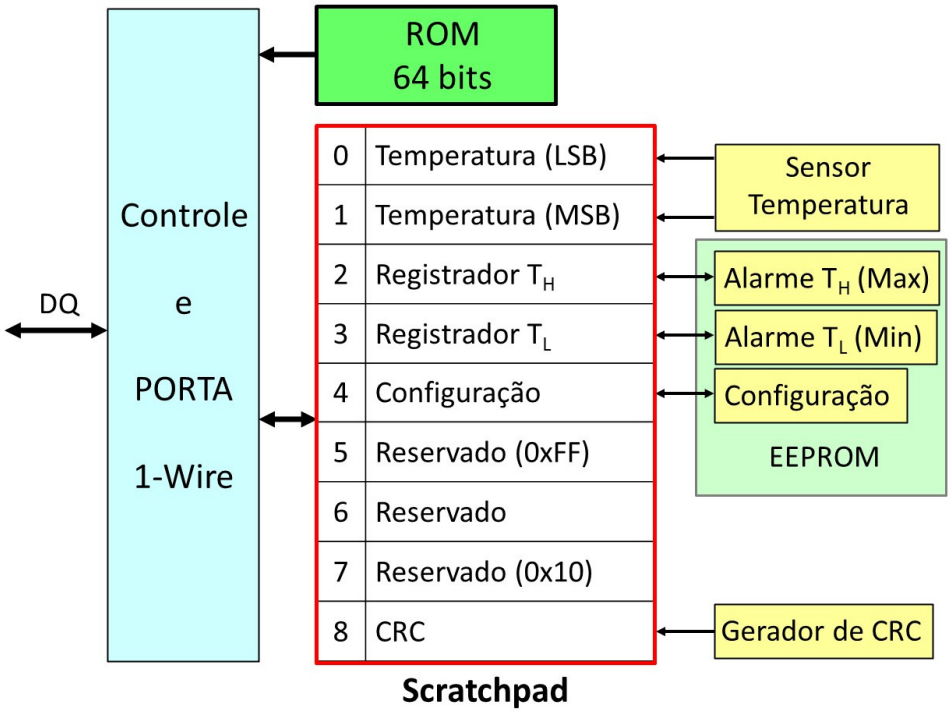


Figura O.13. Diagrama simplificado do DS18B20.

A memória ROM armazena um código de 64 bits que é único para cada dispositivo. Esse código permite que vários dispositivos compartilhem o mesmo barramento. Toda operação com o DS18B20 é feita com a intermediação do Scratchpad. Quando se solicita uma conversão de temperatura, o resultado estará disponível nas posições 0 e 1 desta scratchpad. As posições 2, 3 e 4 podem ser preenchidas com os 3 bytes da EEPROM. Por outro lado, o usuário pode escrever nas posições 2, 3 e 4 do scratchpad e solicitar que sejam gravados na EEPROM. Quando se solicita os alarmes de temperatura, eles são escritos nas posições 2 e 3 da scratchpad. Quando não usam os alarmes de temperatura, essas duas posições podem ser usadas como memória extra. O registrador de configuração, tanto para leitura, como para escrita, é acessado através da posição 4 deste Scratchpad. A posição 8 contém o CRC que é usado para garantir a integridade de cada leitura do Scratchpad.

O.4.2. Comandos para Operação do DS18B20

Aqui faremos um breve resumo dos comandos disponíveis para a operação do DS18B20. O manual faz a distinção entre comandos de ROM e comandos de operação (funções). Entretanto, aqui vamos apresentar todos juntos, sendo que na coluna da esquerda, junto sinalizamos junto com o código, o identificador “ROM” ou “Função”.

Tabela O.2. Lista dos comandos para operação do DS18B20

Código	Descrição
0xF0 ROM	**Search Rom** Permite que o mestre descubra o código (64 bits) de cada dispositivo conectado ao barramento. Assim, determina quantos dispositivos estão presentes e pode acessar individualmente cada um deles. Ver detalhes no ER xx.xx.
0x33 ROM	**Read ROM** Só pode ser usado quando se tem um único dispositivo presente no barramento. O dispositivo responde com seu código (64 bits) único.
0x55 ROM	**Match ROM** Este comando, seguido pelo código (64 bits) permite que o mestre se comunique com um particular dispositivo presente no barramento.
0xCC ROM	**Skip ROM** Permite que o mestre comande todos os dispositivos presentes no barramento. Por exemplo, comandar para iniciar a conversão de temperatura. No caso de o mestre precisar ler o periférico, este comando só funciona se um único dispositivo estiver presente.
0xEC ROM	**Alarm Search** Este comando se parece com o Search ROM, porém só participarão os dispositivos que estiverem com o alarme acionado.
0x44 Função	**Convert T** Dispara a conversão de temperatura.
0x4E Função	**Write Scratchpad** Permite ao mestre escrever 3 bytes no Scratchpad. As escritas acontecem em sequência nos endereços 2, 3 e 4.
0xBE Função	**Read Scratchpad** Faz a leitura dos 9 bytes do Scratchpad, do endereço 0 até o 8. O último byte é o CRC.
0x48 Função	**Copy Scratchpad** Copia os endereços 2, 3 e 4 do Scratchpad para a EEPROM.
0xB8 Função	**Recall E^2** Copia as 3 posições da EEPROM para o Scratchpad.
0xB4 Função	**Read Power Supply** Após este comando, o mestre deve ler o barramento pois os dispositivos que estiverem usando “alimentação parasita” irão puxar a linha para baixo.

Todo acesso ao DS18B20 é iniciado com um Ressete. Logo a seguir estão alguns exemplos.

- [Reset] [Comando ROM] [Função] [...].

- [Reset] [Skip ROM] [Convert T]

Iniciar a conversão de temperatura quando se tem um único dispositivo:

- [Reset] [Skip ROM] [Read Scratchpad] [byte 0] [byte 1] … [byte 8]

Ler a temperatura quando se tem um único dispositivo:

- [Reset] [Skip ROM] [Write Scratchpad] [byte 2] [byte 3] [byte 4] [Copy Scratchpad]

Gravar uma determinada configuração (bytes 2, 3 e 4).

- [Reset] [Match ROM] [byte 0] [byte 1] … [byte 7] [Convert T]

Iniciar a conversão de temperatura usando o código único (64 bits) do dispositivo.

- [Reset] [Match ROM] [byte 0] … [byte 7] [Read Scratchpad] [byte 0] … [byte 8]

Ler a temperatura usando o código único (64 bits) do dispositivo.

- [Reset] [Search ROM] [sequência para buscar o código de um dispositivo]

Determinar o código único de um dispositivo. Esta sequência precisa ser repetida até que todos os dispositivos presentes sejam identificados. Veja ER x.xx.

O.4.2. Detalhes para Operação do DS18B20

Neste tópico vamos abordar os detalhes que ainda faltam para a operação do DS18B20. O primeiro ponto é o Registrador de Configuração, que tem 8 bits, dois quais apenas dois podem ser usados, como mostrado na Figura O.14. Este registrador permite configurar a resolução do termômetro, que pesa diretamente no tempo de conversão, como mostrado na Tabela O.3.

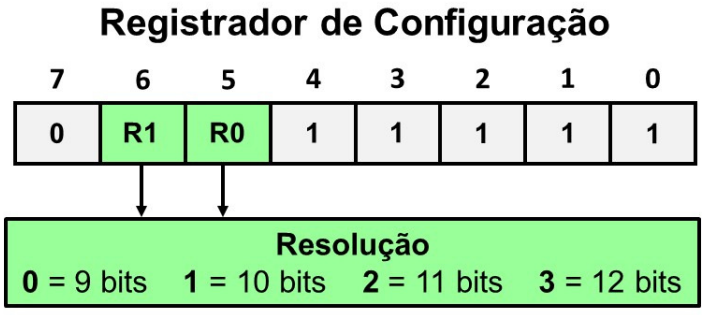


Figura O.14. O Registador de Configuração do DS18B20.

Tabela O.3. Configuração da resolução do termômetro do DS18B20

R1 R0	Resolução	Tempo de Conversão (máximo)
0 0	9 bits	93,75 ms
0 1	10 bits	187,5 ms
1 0	11 bits	375 ms
1 1	12 bits	750 ms

O resultado da conversão tem o formato fixo de 16 bits com sinal, como mostrado na figura abaixo. O sinal é necessário para as temperaturas negativos. A unidade é sempre graus Celsius. Quando configura para 12 bits, todos os bits trazem informação. Se configurado para 11 bits, o bit mais à direita, o bit 0, deve ser ignorado. Em 10 bits, ignoram-se os bits 0 e 1. Então, é claro que na resolução de 9 bits, ignoram-se os bits 0, 1 e 2.

Byte 1 do Scratchpad								Byte 0 do Scratchpad							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
S	S	S	S	S	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}

Figura O.15. Formato do Registador de Temperatura (S = bit de sinal).

Para converter uma leitura em graus Celsius é muito simples: basta multiplicar por 0,0625, pois 2^{-4} = 0,0625. Se for usada uma resolução menor que 12 bits, devem-se zerar os bits que devem ser ignorados. A tabela abaixo apresenta alguns exemplos. Vale lembrar que a faixa de operação deste termômetro é de -55 ºC até +125 ºC.

Tabela O.4. Exemplos de interpretação de leituras do termômetro do DS18B20

Binário	Hexa	Decimal	Temperatura
0000 0111 1101 0000	0x07D0	2.000	+125,0000
0000 0101 0101 0101	0x0555	1.365	+85,3125
0000 0000 0000 0001	0x0001	1	+0,0625
1111 1111 1111 1111	0xFFFF	-1	-0,0625
1111 1111 1111 1000	0xFFF8	-8	-0,5000
1111 1110 0110 1111	0xFE6F	401	-25,0625
1111 1100 1001 0000	0xFC90	880	-55,0000

Toda vez que o Scratchpad é lido, junto com os 8 bytes, é entregue um byte adicional com o CRC. A finalidade é permitir a verificação da integridade dos dados. O mesmo

acontece quando se realiza a operação Search ROM, que busca pelos códigos únicos de todos os dispositivos presentes, neste caso, são apenas 8 bytes, sendo que o oitavo é o CRC. Não é finalidade deste apêndice estudar CRC, por isso, vamos apenas apresentar a função polinomial e o esquema do gerador usando LFSR (Linear Feedback Shift Register). O polinômio está indicado a seguir e o gerador recursivo desenhado logo abaixo.

$$CRC = X^8 + X^5 + X^4 + 1$$

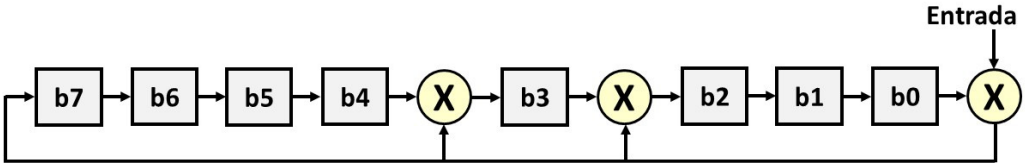


Figura O.16. Diagrama do gerador de CRC (X = operação XOR).

Para finalizar este tópico, devemos apresentar do código de 64 bits que é único para cada dispositivo. Ele é composto por 64 bits (8 bytes) obedecendo a arrumação apresentada abaixo. Os 8 bits mais à direita identificam a família do dispositivo. Para o caso do DS18B20 este código é igual a 0x28. Os bits de 8 até 55 trazem um código particular dentro da família. Como são 48 temos um total de 256 Tera códigos (2^{48}). Os 8 bits mais à esquerda fornecem o CRC para verificar a integridade do código.

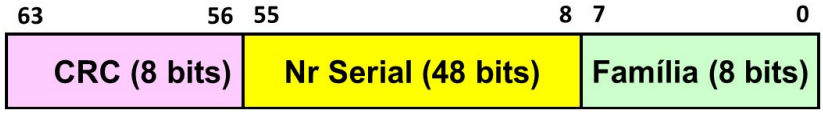


Figura O.17. Detalhamento do código único de 64 bits gravado em ROM.

O.5. Exercícios Resolvidos

Vamos agora, com o auxílio de diversos Exercícios Resolvidos, praticar o emprego do DS18B20.

ER O.1. Este exercício é o mais básico de todos. Ele pede para usar a inicialização do barramento para detectar a presença de algum dispositivo 1-Wire. A conexão sugerida para este exercício e para os próximos está apresentada na Figura O.18.

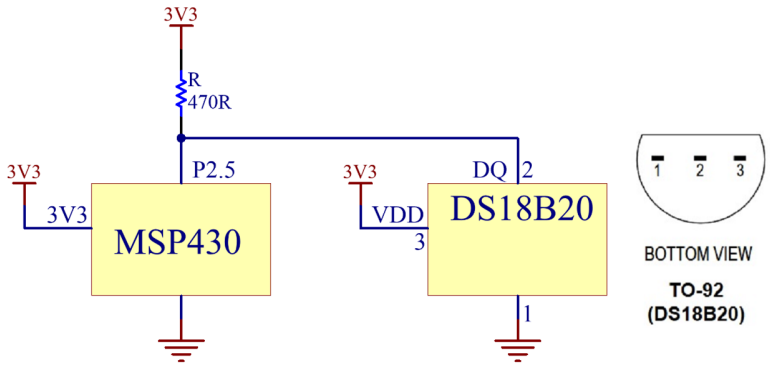


Figura O.18. Sugestão de conexão do DS18B20 ao MSP430.

Solução:
Como descrito no item O.2.1, Figura O.4, após o sinal de ressete, os dispositivos presentes devem colocar sua saída em nível baixo. Isto significa que se houver, pelo menos um dispositivo presente, o mestre vai ler o sinal de presença em nível baixo. A função `char ow_rst(void)` retorna `TRUE` se o sinal de presença foi ativado e `FALSE`, caso negativo.

Assim, este programa é muito simples, ele fica num laço infinito, fazendo a inicialização a cada 0,5 segundo e sinalizando nos leds a presença ou ausência de dispositivos 1-Wire. Mesmo com o programa rodando, o leitor pode conectar ou desconectar seu dispositivo 1-Wire para conferir a sinalização nos leds.

Listagem da solução do ER O.01

```
// ER O.01
// Enviar reset e ler sinal de presença
// P2.5 = 1-Wire
// Led VD = OK  e  Led VM = NOK

#include <msp430.h>

#define TRUE  1
#define FALSE 0

// Protótipo de Funções para os leds e auxiliar do Osciloscópio
inline void SCOPE(void)  {  P2OUT |=  BIT4;  }  //P2.4=1
inline void scope(void)  {  P2OUT &= ~BIT4;  }  //P2.4=0
inline void Scope(void)  {  P2OUT ^=  BIT4;  }  //P2.4=invertido
inline void led_VM(void) {  P1OUT |=  BIT0;  }  //VM=aceso
inline void led_vm(void) {  P1OUT &= ~BIT0;  }  //VM=apagado
inline void led_Vm(void) {  P1OUT ^=  BIT0;  }  //VM=invertido
inline void led_VD(void) {  P4OUT |=  BIT7;  }  //VD=aceso
inline void led_vd(void) {  P4OUT &= ~BIT7;  }  //VD=apagado
```

```
inline void led_Vd(void) {  P4OUT ^=  BIT7;  }  //VD=invertido

// Constantes para os atrasos
#define OW_RST1 88
#define OW_RST2 10
#define OW_WR   10
#define OW_RD    7

// Protótipo das funções para 1-Wire
char ow_rst(void);
inline char ow_pin_rd (void) {return (P2IN&BIT5)>>5;} //Ler P2.5
inline void ow_pin_low(void) {P2DIR |=  BIT5; }       //P2.5 = Zero
inline void ow_pin_hiz(void) {P2DIR &= ~BIT5; }       //P2.5 = Hi-Z
void ow_config(void);                                 //Configurar P2.5
void gpio_config(void);                               //Configurar Leds

void main(void){
    volatile char vetor[9];
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    gpio_config();
    ow_config();
    while(TRUE){
        if (ow_rst() == FALSE)  {led_VM();  led_vd();   }
        else                    {led_vm();  led_VD();   }
        __delay_cycles(500000);      //Esperar 0,5 segundo
    }
}

// Gerar Reset e ler resposta do periférico
// TRUE --> OK FALSE --> Não OK
char ow_rst(void){
  char i,aux;
  ow_pin_low();
  for (i=0; i<OW_RST1; i++) __no_operation();
  ow_pin_hiz();
  for (i=0; i<OW_RST2; i++) __no_operation();
  aux=ow_pin_rd();
  for (i=0; i<(OW_RST1-OW_RST2); i++) __no_operation();
  return aux^BIT0;  //1 = presente
}

// Configurar P2.5 para operar com 1-Wire
void ow_config(void){
    P2DIR |= BIT5;      //P2.5 = saída
    P2OUT &= ~BIT5;     //P2.5 = 0
    P2REN &= ~BIT5;     //Desabilita Pull/up/down
}

// Inicialização Leds e Auxiliar para Osciloscópio
void gpio_config(){
```

```
    P1DIR |= BIT0;  P1OUT &= ~BIT0; //Led VM
    P4DIR |= BIT7;  P4OUT &= ~BIT7; //Led VD
    P2DIR |= BIT4;  P2OUT &= ~BIT4; //Scope (P2.4)
}
```

ER O.2. Neste exercício pede para fazer a leitura do scratchpad num vetor de 9 posições e conferir o CRC.

Solução:
Este exercício apresenta ao leitor as funções básicas para se operar com o barramento 1-Wire. Como ainda não vimos como endereçamento dos dispositivos, este exercício só funciona corretamente com a presença de um único dispositivo 1-Wire. A sequência de comandos que vamos usar está listada abaixo.

- [Reset] [Skip ROM] [Read Scratchpad] [byte 0] [byte 1] … [byte 8]

Para que possamos realizar essas operações, além das funções do exercício anterior, precisaremos das funções que estão listadas abaixo. Elas serão usadas na solução dos demais exercícios.

`void ow_wr_0 (void)` → Escreve 0 no barramento 1-Wire.

`void ow_wr_1 (void)` → Escreve 1 no barramento 1-Wire.

`void ow_wr_byte (char dt)` → Escreve o byte `dt` no barramento 1-Wire.

`char ow_rd_bit (void)` → Lê um bit do barramento 1-Wire. Retorna o bit lido.

`char ow_rd_byte (void)` → Lê um byte do barramento 1-Wire. Retorna o byte lido.

`char ow_rd_blk (char *vt, char qtd)` → Recebe no vetor `vt` a quantidade `qtd` de bytes lidos no barramento 1-Wire. Retorna a quantidade lida.

`char ow_crc (cahr *vt, char qtd)` → Faz verificação de redundância cíclica dos `qtd` bytes disponíveis no vetor `vt`. Retorna `TRUE` em caso de sucesso e `FALSE` em caso de falha.

Com o uso das funções acima listadas, o programa ficou simples. Note o laço principal onde é realizado o reset do barramento e em seguida as operações para a leitura da memória. A sinalização de erro é dada pelo led vermelho. São duas possibilidades: nenhuma resposta de presença durante o reset do barramento 1-Wire ou falha na conferência do CRC. Após uma espera de 0,5 s, o ciclo se repete. Para que o leitor possa verificar o resultado da leitura, é interessante colocar um *breakpoint* na linha de espera (`__delay_cycles`).

Se o seu DS18B20 acabou de ser energizado, espera-se na variável vetor os seguintes valores, em hexadecimal: 50, 05, 4B, 46, 7F, FF, 0C,10, 1C.

Solução do ER O.2

```
// ER O.02
// Ler a Scratchpad (SRAM)
// P2.5 = 1-Wire

#include <msp430.h>

#define TRUE  1
#define FALSE 0

// Protótipo de Funções para os leds e auxiliar do Osciloscópio
... copiar do ER O.1

// DS18B20 - Comandos
#define DS_SKIP_ROM     0xCC  //Ignorar código único
#define DS_RD_SCRATCH   0xBE  //Ler scratchpad incluindo CRC (9 bytes)

// Constantes para os atrasos
... copiar do ER O.1

// Protótipo das funções para 1-Wire
char ow_crc(char *vt, char qtd);
char ow_rd_blk(char *vt, char qtd);
char ow_rd_byte(void);
char ow_rd_bit(void);
void ow_wr_byte(char dt);
void ow_wr_1(void);
void ow_wr_0(void);
char ow_rst(void);
inline char ow_pin_rd (void) {return (P2IN&BIT5)>>5;} //Ler P2.5
inline void ow_pin_low(void) {P2DIR |=  BIT5; }       //P2.5 = Zero
inline void ow_pin_hiz(void) {P2DIR &= ~BIT5; }       //P2.5 = Hi-Z
void ow_config(void);                                 //Configurar P2.5
void gpio_config(void);                               //Configurar Leds

void main(void){
    char i,vetor[9];
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    gpio_config();
    ow_config();
    while(TRUE){
        for (i=0; i<9; i++) vetor[i]=0;
        led_vm();
        led_vd();
        if (ow_rst() == FALSE){
            led_VM();
```

```
            led_vd();
        }
        else{
            ow_wr_byte(DS_SKIP_ROM);
            ow_wr_byte(DS_RD_SCRATCH);
            ow_rd_blk(vetor,9);
            if (ow_crc(vetor, 9) == TRUE){
                led_vm();
                led_VD();
            }
            else{
                led_VM();
                led_vd();
            }
        }
        __delay_cycles(500000);      //Esperar 0,5 segundo
    }
}

// Calcula CRC para uma certa quantidade de bytes
char ow_crc(char *vt, char qtd) {
  char i,j,crc = 0;
  for (j=0; j<qtd; j++) {
    crc = crc ^ vt[j];
    for (i=0; i<8; i++) crc = crc>>1 ^ ((crc & 1) ? 0x8c : 0);
  }
  if (crc == 0) return TRUE;
  else          return FALSE;
}

// Ler um bloco de bytes pelo 1-Wire
char ow_rd_blk(char *vt, char qtd){
  char i;
  for (i=0; i<qtd; i++){
    vt[i]=ow_rd_byte();
  }
  return qtd;
}

// Ler um byte pelo 1-Wire
char ow_rd_byte(void){
  char i,dt=0;
  for (i=0; i<8; i++){
    dt=dt>>1;
    if ( ow_rd_bit() == BIT0)  dt |= 0x80;
  }
  return dt;
}

// Ler 1 bit no barramento 1-Wire
```

```
char ow_rd_bit(void){
    char x,i;
    ow_pin_low();
    ow_pin_hiz();
    x=ow_pin_rd();  //Ler barramento
    for (i=0; i<OW_RD; i++)    __no_operation();
    return x;
}

// Enviar um byte pelo 1-Wire
void ow_wr_byte(char dt){
  char i;
  for (i=0; i<8; i++){
    if ((dt&BIT0)==0) ow_wr_0();
    else              ow_wr_1();
    dt = dt>>1;
  }
}

// Escrever 1 no barramento 1-Wire
void ow_wr_1(void){
  char i;
  ow_pin_low();
  ow_pin_hiz();
  for (i=0; i<OW_WR; i++)    __no_operation();
}

// Escrever 0 no barramento 1-Wire
void ow_wr_0(void){
  char i;
  ow_pin_low();
  for (i=0; i<OW_WR; i++)    __no_operation();
  ow_pin_hiz();
  __no_operation();
}

char ow_rst(void)       {;}         // copiar do ER O.1
void ow_config(void)    {;}         // copiar do ER O.1
void gpio_config(void)  {;}         // copiar do ER O.1
```

ER O.3. Este exercício pede para apresentar a temperatura em graus Celsius numa variável do tipo float.

Solução:
Nesta solução vamos empregar as funções básicas apresentadas no exercício anterior. Como ainda não vimos como endereçamento dos dispositivos, este exercício só funciona

corretamente com a presença de um único dispositivo 1-Wire. A sequência de comandos que vamos usar está listada abaixo. A primeira linha pede para iniciar a conversão de temperatura. A segunda linha faz a leitura da conversão.

- [Reset] [Skip ROM] [Convert T]
- [Reset] [Skip ROM] [Read Scratchpad] [byte 0] [byte 1] … [byte 8]

Ao ser ligado, o DS18B20 inicia na resolução de 12 bits, isto significa que cada conversão demora 750 ms. Por essa razão foi usado o `__delay_cycles(1.000.000)` para criar um atraso aproximado de 1 segundo. A listagem abaixo apresenta a solução. Para que ela não ficasse longa, foram omitidos vários trechos que eram idênticos à solução anterior. A variável `temp` foi declara `volatile` para evitar a otimização do compilador. Uma boa ideia é colocar um breakpoint logo no começo do laço infinito para pode observar os resultados da temperatura.

Solução do ER O.3

```
// ER O.03
// Ler a temperatura
// P2.5 = 1-Wire

#include <msp430.h>

// .. Copiar demais linhas do exercício anterior

// DS18B20 - Comandos
#define DS_SKIP_ROM     0xCC  //Ignorar código único
#define DS_CONVERT_T    0x44  //Iniciar conversão de temperatura
#define DS_RD_SCRATCH   0xBE  //Ler scratchpad incluindo CRC (9 bytes)

void main(void){
    char i,vetor[9];
    int aux;
    volatile float temp;
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    gpio_config();
    ow_config();
    while(TRUE){
        led_vm();
        led_vd();
        if (ow_rst() == FALSE){
            led_VM();
            led_vd();
        }
        else{
            ow_wr_byte(DS_SKIP_ROM);
            ow_wr_byte(DS_CONVERT_T);
            __delay_cycles(1000000);
```

```
            ow_rst();
            ow_wr_byte(DS_SKIP_ROM);
            ow_wr_byte(DS_RD_SCRATCH);
            ow_rd_blk(vetor,9);
            if (ow_crc(vetor, 9) == TRUE){
                led_vm();
                led_VD();
                aux=(vetor[1]<<8)+vetor[0];
                temp=0.0625*aux;
            }
            else{
                led_VM();
                led_vd();
            }
        }
    }
}

// As funções abaixo devem ser copiadas da solução do ER O.2
char ow_crc(char *vt, char qtd)     { ... }
char ow_rd_blk(char *vt, char qtd)  { ... }
char ow_rd_byte(void)               { ... }
char ow_rd_bit(void)                { ... }
void ow_wr_byte(char dt)            { ... }
void ow_wr_1(void)                  { ... }
void ow_wr_0(void)                  { ... }
char ow_rst(void)                   { ... }
void ow_config(void)                { ... }
void gpio_config()                  { ... }
```

ER O.4. Este exercício pede ler o Código Único de 64 bits quando se tem um único dispositivo presente no barramento.

Solução:
A solução deste exercício só funciona se um único dispositivo estiver presente no barramento. Ele faz a leitura do código único que pode ser usado para individualizar os dispositivos. Conectando-os um a um ao barramento, o leitor pode descobrir o código de cada um deles. De posse desses códigos, é possível trabalhar com vários dispositivos. A sequência de comandos é apresenta abaixo.

- [Reset] [Read ROM] [byte 0] [byte 1] … [byte 7]

No caso do DS18B20 usado neste exercício, o resultado foi a sequência hexadecimal:

28 FF 37 77 74 16 04 E5. Note que no vetor os 8 bytes estão na ordem inversa. Isso não importa, porque a mesma ordem será usada para enviar os códigos pelo barramento 1-Wire.

0	1	2	3	4	5	6	7
Família	Número Serial, 48 bits						CRC
28	FF	37	77	74	16	04	E5

Figura O.19. Um resultado da leitura do Código da ROM.

Abaixo está a listagem da solução deste exercício. Ela dispensa maiores explicações. É muito semelhante aos exercícios anteriores.

Solução do ER O.4

```
// ER O.04
// Ler o código da ROM
// P2.5 = 1-Wire

#include <msp430.h>

// .. Copiar demais linhas do exercício anterior

// DS18B20 - Comandos
#define DS_RD_ROM       0x33  //Ler código único

void main(void){
    char i,vetor[8];
    int aux;
    volatile float temp;
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    gpio_config();
    ow_config();
    while(TRUE){
        led_vm();
        led_vd();
        if (ow_rst() == FALSE){
            led_VM();
            led_vd();
        }
        else{
            ow_wr_byte(DS_RD_ROM);
            ow_rd_blk(vetor,8);
            if (ow_crc(vetor, 8) == TRUE){
                led_vm();
                led_VD();
```

```
            }
            else{
                led_VM();
                led_vd();
            }
        }
        __delay_cycles(500000);
    }
}

// As funções abaixo devem ser copiadas da solução do ER O.2
char ow_crc(char *vt, char qtd)      { ... }
char ow_rd_blk(char *vt, char qtd)   { ... }
char ow_rd_byte(void)                { ... }
char ow_rd_bit(void)                 { ... }
void ow_wr_byte(char dt)             { ... }
void ow_wr_1(void)                   { ... }
void ow_wr_0(void)                   { ... }
char ow_rst(void)                    { ... }
void ow_config(void)                 { ... }
void gpio_config()                   { ... }
```

ER O.5. Este exercício pede se usar 4 dispositivos DS18B20 para medir a temperatura, considerando que conhecemos seus respectivos códigos de ROM. A figura abaixo apresenta a forma de se fazer a conexão.

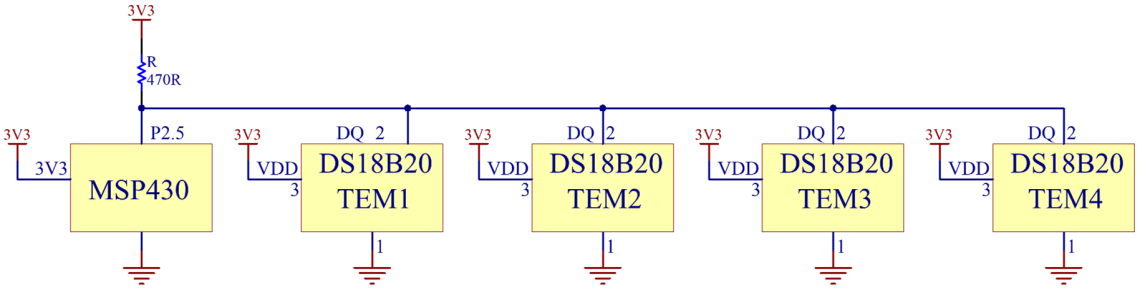


Figura O.20. Esquema para conexão simultânea de 4 DS18B20.

Solução:
Este exercício é interessante porque permite que o leitor faça controles sofisticas de temperatura usando vários termômetros conectados ao mesmo barramento. Abaixo está a lista com os códigos ROM dos 4 termômetros usados neste exemplo. Esses códigos foram obtidos usando o recurso da solução anterior.

Termômetro 1: 28 4C 45 6A 0B 00 00 67
Termômetro 2: 28 2E B7 66 0B 00 00 FD

Termômetro 3: 28 5E 37 68 0B 00 00 BB
Termômetro 4: 28 FF 37 77 74 16 04 E5

É importante chamar a atenção para função `ow_wr_blk()` que permite o envio de uma certa quantidade de bytes pelo barramento 1-Wire. O leitor deve examinar a listagem abaixo e identificar o laço infinito. Note que o comando de conversão é enviado simultaneamente para os 4 termômetros, pois foi precedido pelo comando "Skip ROM". Depois de uma espera de 1 segundo os termômetros são lidos individualmente usando os respectivos códigos de ROM. Se o led vermelho piscar, é porque que há algum erro acontecendo. Usando breakpoint, o erro pode ser facilmente identificado. A sugestão é para que o leitor coloque um breakpoint sobre o `while (TRUE)` para que possa examinar as temperaturas a cada laço de medição. Ao colocar o dedo sobre um dos termômetros, será possível notar a variação de temperatura.

Solução do ER O.5

```
// ER O.05
// Operar com 4 termômetros
// P2.5 = 1-Wire

#include <msp430.h>

// .. Copiar demais linhas do exercício anterior

// DS18B20 - Comandos
#define DS_SEARCH_ROM   0xF0  //Busca pelos códigos únicos (ROM)
#define DS_RD_ROM       0x33  //Ler código único
#define DS_MATCH_ROM    0x55  //Selecionar disp com código único
#define DS_SKIP_ROM     0xCC  //Ignorar código único
#define DS_ALARM_SEARCH 0xEC  //Buscar pelos dispositivos alarmados
#define DS_CONVERT_T    0x44  //Iniciar conversão de temperatura
#define DS_RD_SCRATCH   0xBE  //Ler scratchpad incluindo CRC (9 bytes)
#define DS_WR_SCRATCH   0x4E  //Escrever bytes 2(TH),3(TL)e 4(Config)
#define DS_COPY_SCRATCH 0x48  //Copiar bytes 2, 3 e 4 para a EEPROM
#define DS_RECALL_EE    0xB8  //Transf bytes 2,3, 4 da EEPROM --> scratch
#define DS_RD_POWER     0xB4  //DS18B20 transmite status de potência

// Protótipo da nova função
void ow_wr_byte(char dt);

// Códigos dos 4 termômetros
char term_cod[4][8] = {{0x28, 0x4C, 0x45, 0x6A, 0x0B, 0x00, 0x00, 0x67},
                       {0x28, 0x2E, 0xB7, 0x66, 0x0B, 0x00, 0x00, 0xFD},
                       {0x28, 0x5E, 0x37, 0x68, 0x0B, 0x00, 0x00, 0xBB},
                       {0x28, 0xFF, 0x37, 0x77, 0x74, 0x16, 0x04, 0xE5}};

void main(void){
    char i,vetor[9];
```

```
    int aux;
    volatile float temp[4];
    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
    gpio_config();
    ow_config();
    while(TRUE){
        led_vm();
        led_vd();
        if (ow_rst() == FALSE){
            led_VM();
            led_vd();
        }
        else{
            ow_wr_byte(DS_SKIP_ROM);
            ow_wr_byte(DS_CONVERT_T);   //Comandar os 4 termômetros
            __delay_cycles(1000000);
            for (i=0; i<4; i++){
                ow_rst();
                ow_wr_byte(DS_MATCH_ROM);
                ow_wr_blk(&term_cod[i][0],8);
                ow_wr_byte(DS_RD_SCRATCH);
                ow_rd_blk(vetor,9);
                if (ow_crc(vetor, 9) == TRUE){
                    led_vm();
                    led_VD();
                    aux=(vetor[1]<<8)+vetor[0];
                    temp[i]=0.0625*aux;
                }
                else{
                    led_VM();
                    led_vd();
                }
            }
        }
    }
}

// Escrever um bloco de bytes pelo 1-Wire
char ow_wr_blk(char *vt, char qtd){
  char i;
  for (i=0; i<qtd; i++){
    ow_wr_byte(vt[i]);
  }
  return qtd;
}

// As demais funções devem ser copiadas da solução do ER O.2
```

ER O.6. Este exercício pede para usar o mesmo esquema de 4 termômetros do exercício anterior para comprovar o algoritmo para identificação dos códigos de ROM, previsto pelo protocolo quando se tem vários dispositivos presentes no barramento 1-Wire.

Solução:
O protocolo 1-Wire prevê uma forma de se determinar os códigos de ROM dos dispositivos, mesmo quando vários estão presentes no barramento. Isto é conseguido usando a função "Search ROM". Após este comando, seguindo a cadência ditada pelo mestre, os escravos vão apresentando, um a um, os 64 bits que compõe seus diferentes códigos, partindo com o bit menos significativo. O mestre vai indicando os dispositivos que devem abandonar a busca, até que reste apenas um só.

O problema de identificação se resume na varredura de uma árvore binária formada pelos diversos códigos dos dispositivos presentes. Para facilitar a compreensão, vamos partir com o caso prático visto no exercício anterior, onde trabalhamos com 4 dispositivos cujos códigos estão listados na tabela abaixo. Note que, por código, se entende a união do número que indica a família, mais o número serial e o CRC. A busca começa pelo bit 0 do byte 0 e termina com o bit 7 do byte 7.

Tabela O.5. Detalhamento dos Códigos de ROM dos 4 dispositivos que serão usados como exemplo neste exercício

	Código gravado em ROM							
	CRC	**Número Serial**						**Família**
	Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
T1	67	00	00	0B	6A	45	4C	28
T2	FD	00	00	0B	66	B7	2E	28
T3	BB	00	00	0B	68	37	5E	28
T4	E5	04	16	74	77	37	FF	28

Iniciamos com a explicação de como os dispositivos presentes no barramento respondem ao comando "Search ROM". Após este comando, todos os dispositivos respondem enviando o valor do primeiro bit de seu código e em seguida o complemento deste bit. Em seguida, o mestre escreve um determinado bit (0 ou 1) no barramento e com isso instrui para que continuem na busca somente aqueles dispositivos que têm o bit de código igual ao bit que o mestre escreveu. Assim, alguns dispositivos vão abandonando a busca. Isto é repetido 64 vezes, uma vez para cada bit de código. No final, apenas um dispositivo responderá à busca e seu código está descoberto.

Num determinado momento, se os diversos dispositivos presentes no barramento 1-Wire enviam um bit de código e depois o complemento deste bit, temos 4 possibilidades,

listadas na tabela abaixo. Então, o mestre faz a leitura desses dois bits para tomar a decisão de quais dispositivos devem permanecer na busca.

Tabela O.6. Possíveis resultados na busca por um bit de código

bit	$\overline{bit}$	Significado
0	0	Choque de bits, dispositivos com endereços conflitantes.
0	1	Todos os dispositivos têm o mesmo bit de código (bit = 0).
1	0	Todos os dispositivos têm o mesmo bit de código (bit = 1).
1	1	Não existem dispositivos participando da busca.

Apresentamos abaixo o algoritmo de busca, mas de uma forma muito simplificada. Ele passa a primeira ideia do mecanismo. Maia adiante ele é detalhado.

Algoritmo simplificado

1) O mestre envia os comandos de Reset, Search ROM e inicializa contador `cont` = 1.

2) Mestre lê em sequência dois bits: o bit de código e o complemento deste bit de código enviados pelos dispositivos que estão respondendo à busca. Um novo bit de código a cada passada (para `cont` = 1, 2, ..., 64).

3) Mestre analisa os dois bits recebidos e decide se a busca deve prosseguir com os dispositivos que têm este bit de código igual a 0 ou igual a 1.

4) Mestre escreve este bit no barramento indicando que os dispositivos com o bit de código diferente devem abandonar a busca.

5) Mestre incrementar o contador `cont`.

6) Se `cont` < 65, repetir a busca, senão já se tem o código de um dispositivo.

Todo o processo é repetido para o próximo código.

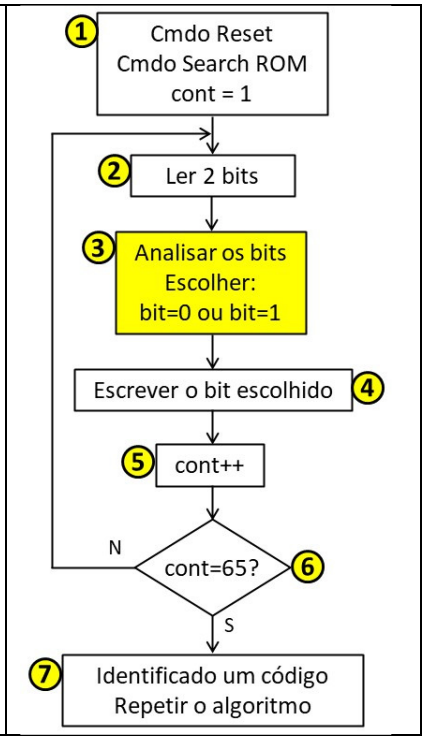


Figura O.21. Fluxograma de um trecho do algoritmo de busca de código de ROM e junto com uma pequena explicação.

Bem, com a explicação acima, tem-se uma pequena ideia de como a busca acontece. A parte mais importante está no trecho de número 3. Como escolher qual o sentido da busca? Quais dispositivos devem continuar a responder à operação de busca? Considerando os 4 dispositivos listados na Tabela O.6, vemos que todos eles, como são termômetros, têm o número de família igual a 0x28. Isto significa que não haverá choque

nos primeiros 8 bits de código. Os choques (ou conflitos) começam a surgir a partir do nono bit. Note que vamos numerar de 1 até 64 os bits que compõe o código (e não de 0 até 63, como usualmente fazemos). Isso vai facilitar o algoritmo.

Vamos particularizar nossa explicação para o byte 1 do código, que é quando surgem os choques e se faz a diferenciação. Temos então 4 bytes: 0x4C, 0x2E, 0x5E e 0xFF. O leitor pode notar que, neste caso, estes bytes farão a individualização dos dispositivos presentes. A tabela abaixo mostra de forma clara que os primeiros 8 bits são idênticos e que a diferenciação acontece nos bits de 9 até 16. Logo abaixo está a figura que ilustra a árvore binária para estas 4 alternativas.

Tabela O.7. Detalhamento dos bits de 1 até 16 dos 4 códigos. Note que os bits são escritos a partir do menos significativo.

Byte 0								**Byte 1**								...	
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	...	
0	0	0	1	0	1	0	0	0	0	1	1	0	0	1	0	···	28 **4C** ...
0	0	0	1	0	1	0	0	0	1	1	1	0	1	0	0	···	28 **2E** ...
0	0	0	1	0	1	0	0	0	1	1	1	1	0	1	0	···	28 **5E** ...
0	0	0	1	0	1	0	0	1	1	1	1	1	1	1	1	···	28 **FF** ...

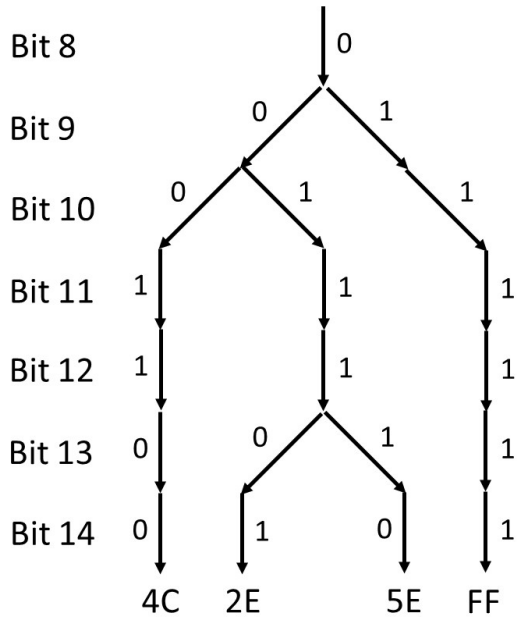
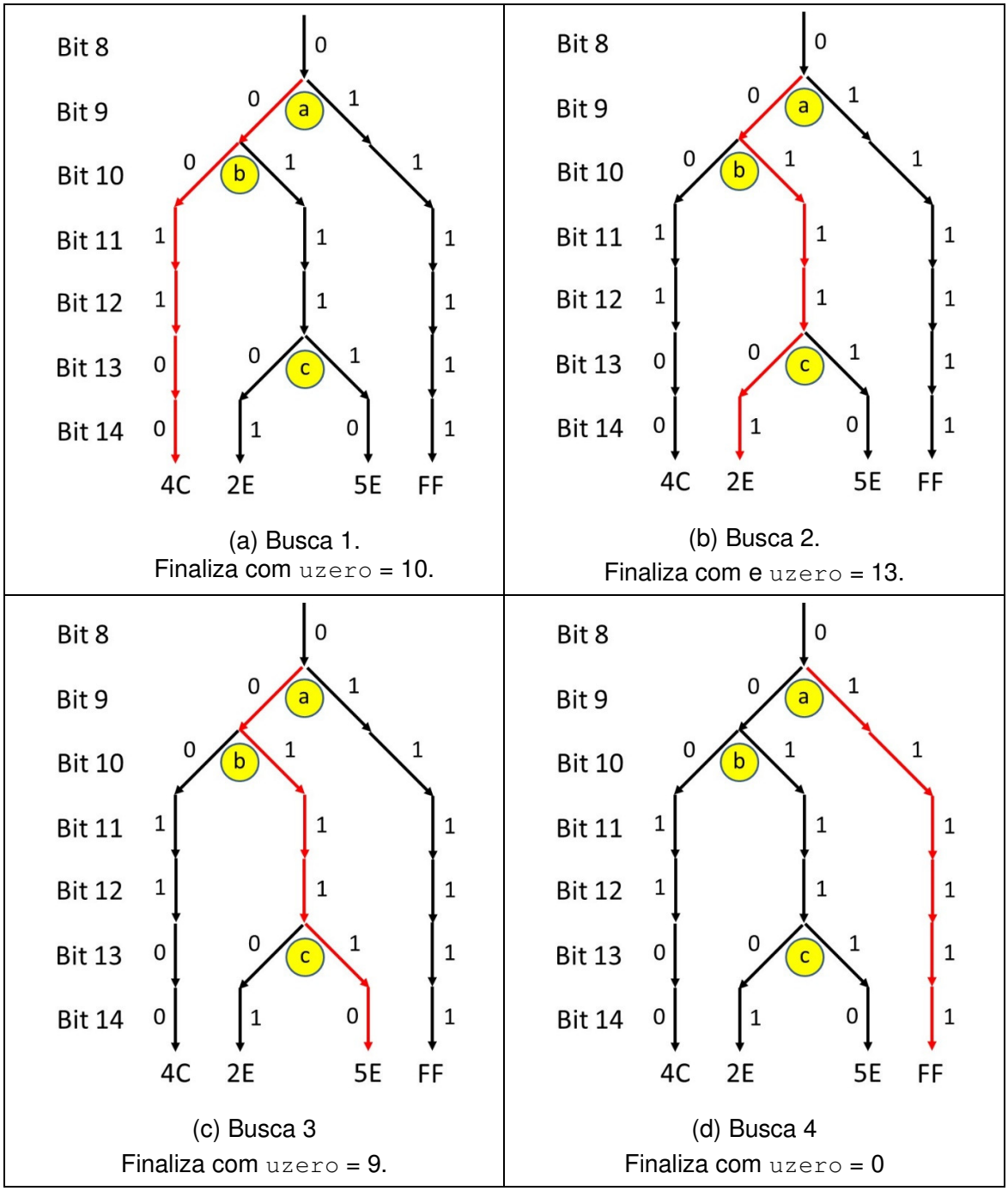


Figura O.22. Árvore binária a ser varrida para individualizar os códigos dos 4 dispositivos.

A Figura O.23 apresenta os caminhos realizados para a identificação de cada disposito. No caso deste exemplo, temos 3 conflitos (choques), denominados de **a** (bit 9), **b** (bit 10) e **c** (bit 13), marcados na figura com círculos amarelos. Primeiro tenta-se varrer a árvore optando pelos bits iguais a zero. Depois, em cada conflito se faz o outro percurso, que é com o bit igual a 1.



(a) Busca 1.
Finaliza com `uzero` = 10.

(b) Busca 2.
Finaliza com e `uzero` = 13.

(c) Busca 3
Finaliza com `uzero` = 9.

(d) Busca 4
Finaliza com `uzero` = 0

Figura O.23. Detalhamento da busca pelos códigos dos 4 dispositivos e o valor da variável `uzero` ao final de cada varredura.

O algoritmo será explicado tomando como base a Figura O.23. Temos quatro variáveis importantes:

- `pbit` → (1, ..., 64) indica a posição do bit que está sendo testado;
- `uxq` → (1, ..., 64) posição do último conflito (choque) de bits e
- `uzero` → (1, ..., 64) posição da última decisão pelo bit 0.
- `rom` → vetor com 8 bytes (64 bits) para armazenar uma varredura.

A Tabela O.8 apresenta detalhes das decisões tomadas em cada conflito. Antes de cada varredura, copiamos em `uxq` a variável `uzero` (`uxq` = `uzero`). Abaixo temos, explicadas passo a passo, as decisões apresentadas na tabelas. O código de cada dispositivo vai sendo formado num vetor denominado `rom`. Depois da primeira varredura, esse vetor `rom` tem o último caminho realizado e, algumas vezes, é consultado durante a próxima varredura.

- 1ª Varredura: Parte com `uxq` = 0 e `uzero` = 0.
  - Conflito **a** (`pbit`=9): temos `pbit` > `uxq`, então bit=0.
  - Conflito **b** (`pbit`=10): temos `pbit` > `uxq`, então bit=0.
  - Finaliza com `uzero` = 10.
- 2ª Varredura: Parte com `uxq` = 10 e `uzero` = 0.
  - Conflito **a** (`pbit`=9): temos `pbit` < `uxq`, então bit=0 (última decisão, `rom`).
  - Conflito **b** (`pbit`=10): temos `pbit` = `uxq`, então bit = 1.
  - Conflito **c** (`pbit`=13): temos `pbit` > `uxq`, então bit = 0.
  - Finaliza com `uzero` = 13.
- 3ª Varredura: Parte com `uxq` = 13 e `uzero` = 0.
  - Conflito **a** (`pbit`=9): temos `pbit` < `uxq`, então bit=0 (última decisão, `rom`)
  - Conflito **b** (`pbit`=10): temos `pbit` < `uxq`, então bit= (última decisão, `rom`)
  - Conflito **c** (`pbit`=13): temos `pbit` = `uxq`, então bit=1.
  - Finaliza com `uzero` = 9.
- 4ª Varredura: Parte com `uxq` = 9 e `uzero` = 0.
  - Conflito **a** (`pbit`=9): temos `pbit` = `uxq`, então bit=1.
  - Finaliza com `uzero` = 0, o que caracteriza o fim.

Tabela O.8. Detalhamento dos bits de 1 até 16 dos códigos.

Varredura	Inicial	a (pbit = 9)	b (pbit = 10)	c (pbit = 13)	Final
1ª	uxq = 0 uzero = 0	pbit > uxq bit = 0	pbit > uxq bit = 0	-	uzero = 10

		uzero = 9	uzero = 10		
2ª	uxq = 10 uzero = 0	pbit < uxq bit = 0 (rom) uzero = 9	pbit = uxq bit = 1	pbit > uxq bit = 0 uzero = 13	uzero = 13
3ª	uxq = 13 uzero = 0	pbit < uxq bit = 0 (rom) uzero = 9	pbit < uxq bit = 1 (rom)	pbit = uxq bit = 1	uzero = 9
4ª	uxq = 9 uzero = 0	pbit = uxq bit = 1	-	-	uzero = 0 (FIM)

Apresentamos a seguir o fluxograma de todo o algoritmo de busca pelos códigos de ROM. Ele está dividido em duas partes. A primeira parte, apresentada na Figura O.24 trata da parte de mais alto nível. Nesta figura está a proposta da seguinte função que retorna a quantidade de códigos descobertos:

```
char ow_adr_search ( char mat[][8], char qtd )
```

- `char mat [][8]` → é o ponteiro para a matriz que vai receber os códigos e
- `char qtd` → indica até quantos códigos podem ser identificados, ela é importante para que o chamador possa definir o tamanho da variável `mat`.

A Figura O.24 então apresenta os passos para contar os códigos descobertos e gravá-los na matriz indicada. Cada busca em particular vai gravando os bits na variável `rom`. Assim, `rom` sempre tem o último caminho percorrido (exceto pela primeira busca). Ao final de cada varredura, o vetor `rom` é copiado para a matriz `mat`. Nesta figura, o bloco "Lógica de bd", marcado em amarelo resume todo a lógica que se usa para decidir qual o valor do bit a ser escolhido (`bd`), ou seja, define a direção da busca.

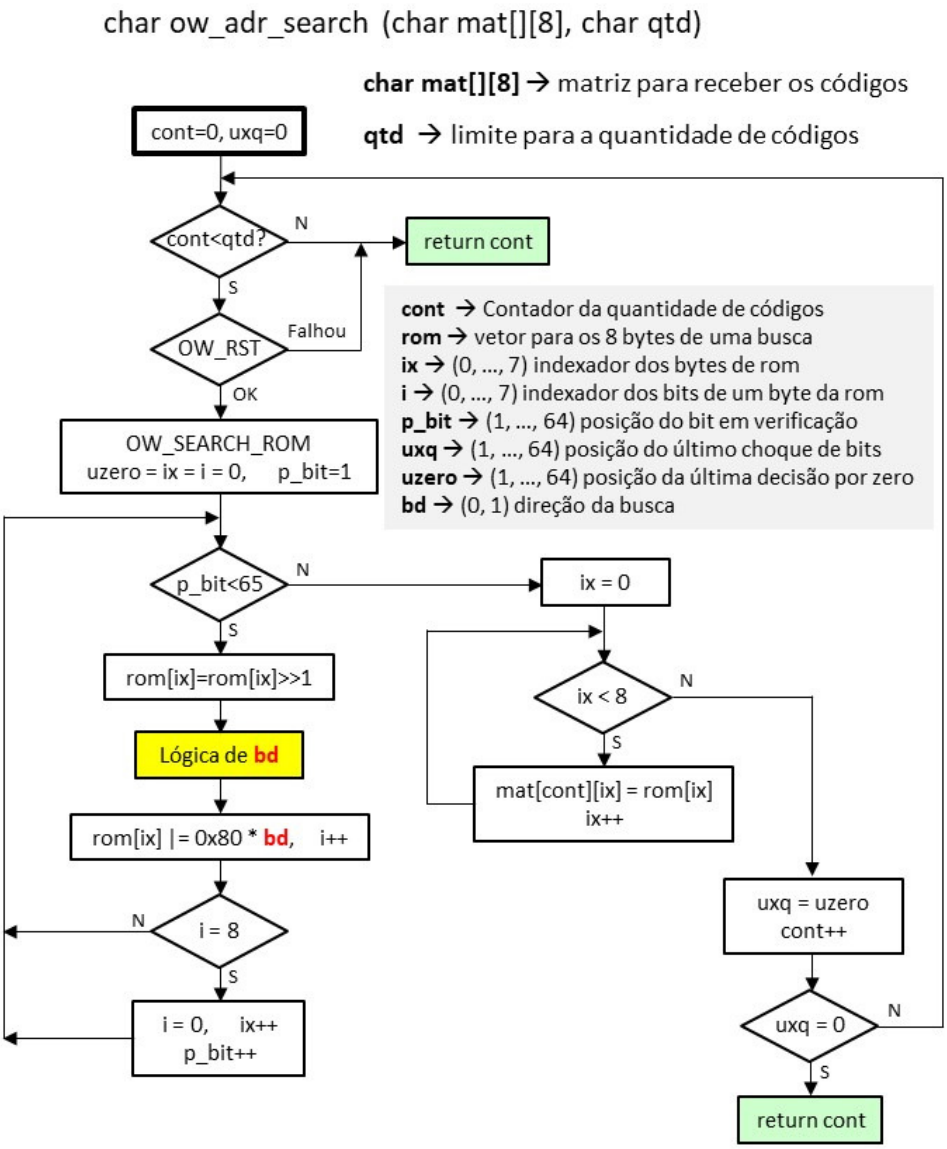


Figura O.24. Fluxograma de busca de códigos de ROM para dispositivos 1-Wire. O bloco "Lógica de bd" é apresentado com detalhes na próxima figura.

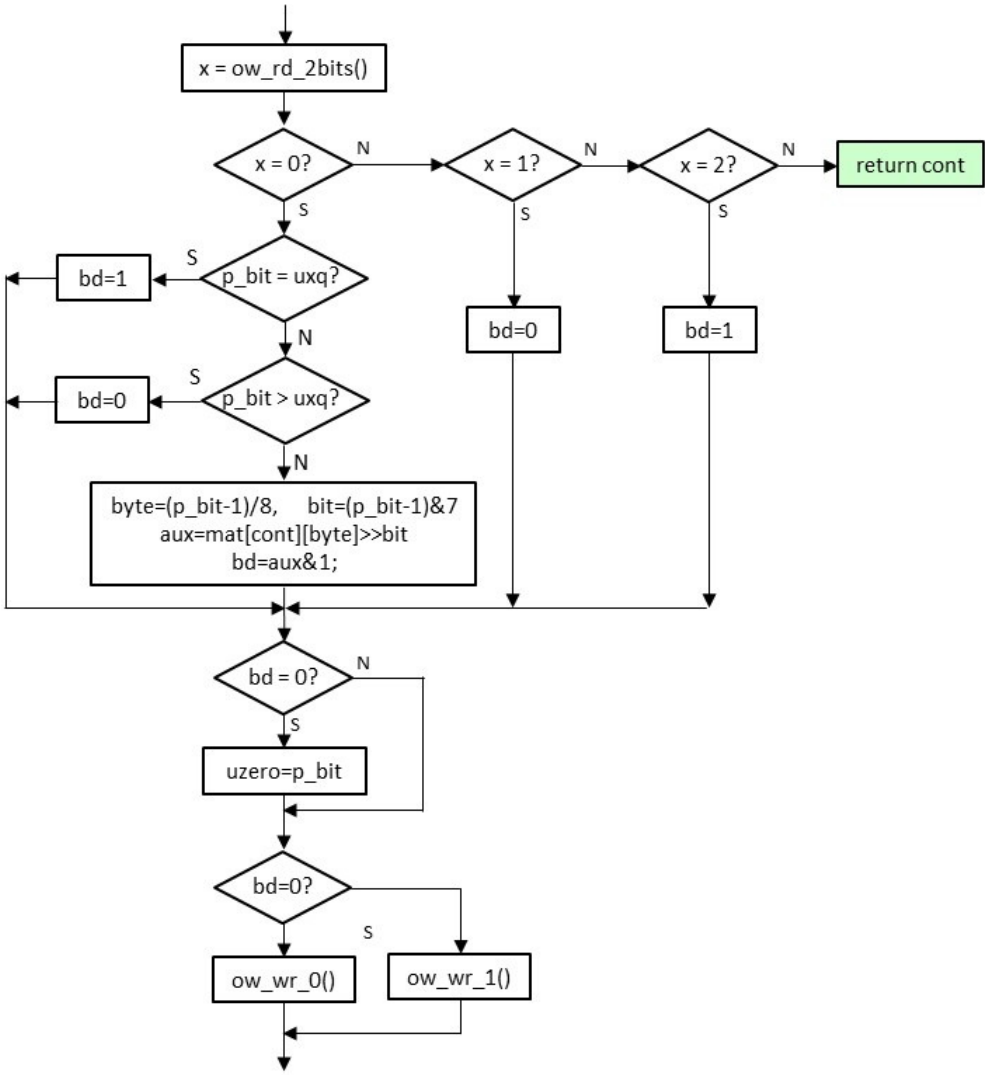


Figura O.25. Esta figura apresenta o trecho que no fluxograma anterior foi denominado de "Lógica de bd". Nele está implementado o protocolo de busca.

Apresentamos a seguir a listagem solução para o exercício proposto. O trecho importante é a função ow_adr_search().

```
// ER O.06
// Algoritmo Search ROM
// P2.5 = 1-Wire

#include <msp430.h>

#define TRUE  1
#define FALSE 0

// Protótipo de Funções para os leds e auxiliar do Osciloscópio
```

```
inline void SCOPE(void)  {  P2OUT |=  BIT4;  }  //P2.4=1
inline void scope(void)  {  P2OUT &= ~BIT4;  }  //P2.4=0
inline void Scope(void)  {  P2OUT ^=  BIT4;  }  //P2.4=invertido
inline void led_VM(void) {  P1OUT |=  BIT0;  }  //VM=aceso
inline void led_vm(void) {  P1OUT &= ~BIT0;  }  //VM=apagado
inline void led_Vm(void) {  P1OUT ^=  BIT0;  }  //VM=invertido
inline void led_VD(void) {  P4OUT |=  BIT7;  }  //VD=aceso
inline void led_vd(void) {  P4OUT &= ~BIT7;  }  //VD=apagado
inline void led_Vd(void) {  P4OUT ^=  BIT7;  }  //VD=invertido

// DS18B20 - Comandos
#define DS_SEARCH_ROM   0xF0  //Busca pelos códigos únicos (ROM)

// Constantes para os atrasos
#define OW_RST1 88
#define OW_RST2 10
#define OW_WR   10
#define OW_RD    7

// Protótipo das funções para 1-Wire
char ow_adr_search(char vetor[][8], char qtd);
char ow_rd_2bits(void);
char ow_crc(char *vt, char qtd);
char ow_rd_blk(char *vt, char qtd);
char ow_rd_byte(void);
char ow_rd_bit(void);
char ow_wr_blk(char *vt, char qtd);
void ow_wr_byte(char dt);
void ow_wr_1(void);
void ow_wr_0(void);
char ow_rst(void);
inline char ow_pin_rd (void) {return (P2IN&BIT5)>>5;} //Ler P2.5
inline void ow_pin_low(void) {P2DIR |=  BIT5; }       //P2.5 = Zero
inline void ow_pin_hiz(void) {P2DIR &= ~BIT5; }       //P2.5 = Hi-Z
void ow_config(void);                                 //Configurar P2.5
void gpio_config(void);                               //Configurar Leds

// Códigos dos 4 termômetros
char term_cod[4][8] = { {0x28, 0x4C, 0x45, 0x6A, 0x0B, 0x00, 0x00, 0x67},
                        {0x28, 0x2E, 0xB7, 0x66, 0x0B, 0x00, 0x00, 0xFD},
                        {0x28, 0x5E, 0x37, 0x68, 0x0B, 0x00, 0x00, 0xBB},
                        {0x28, 0xFF, 0x37, 0x77, 0x74, 0x16, 0x04, 0xE5}
};

void main(void){
    char ix,i,qtd;
    volatile float temp[4];
    char ow_adr[6][8];      //Matriz para guardar até 6 códigos

    WDTCTL = WDTPW | WDTHOLD;   // stop watchdog timer
```

```
    gpio_config();
    ow_config();
    while(TRUE){
        led_vm();
        led_vd();
        for(ix=0; ix<6; ix++)
            for (i=0; i<8; i++)
                ow_adr[ix][i]=0;    //Zerar matriz (capricho)

        qtd=ow_adr_search(ow_adr,6);
        if (qtd == 0)   led_VM();
        else            led_VD();

        for (i=0; i<qtd; i++){
            if (ow_crc(&ow_adr[i][0], 8) == FALSE)
                led_VM();
        }
        __delay_cycles(1000000);
    }
}


// Buscar pelos endereços dos dispositivos
// Algoritmo Search ROM
char ow_adr_search(char mat[][8], char qtd){
    unsigned char cont;     //contador de endereços encontrados
    unsigned char ix;       //indexador contador de bytes (0, ..., 7)
    unsigned char i;        //contador de bits dentro do byte (0, ..., 7)
    unsigned char pos_bit;  //Posição do bit atual (1, ..., 64)
    unsigned char uxq;      //Último choque (1, ..., 64)
    unsigned char uzero;     //Última decisão por zero em caso de choque
(1, ..., 64)
    unsigned char bd;       //Direção da  busca (bit = 0 ou 1)


    unsigned char aux;          //auxiliar
    unsigned char pbyte,pbit;        //auxiliar

    unsigned char x;        //auxiliar
    //unsigned char fim;      //LastDevice_Flag
    //unsigned  char  uxq_fam;    //LastFamilyDiscrepancy,  último  choque
dentro da família (1, ..., 8)
    unsigned char rom[8];   //ROM_NO, formar endereço atual




    for (ix=0; ix<8; ix++) rom[ix]=0;
    cont=0;         //(1, ..., qtd) contador de endereços
    uxq=0;          //(1, ..., 64) último choque
```

```
    //fim=0;          //(0 ou 1)  indicar que terminou
    while(cont<qtd){

        if (ow_rst()==FALSE)        return cont;
        ow_wr_byte(DS_SEARCH_ROM);
        uzero=0;    //(1, ..., 64) posição da última decisão por zero
        ix=i=0;
        for (pos_bit=1; pos_bit<65; pos_bit++){
            rom[ix]=rom[ix]>>1;
            x=ow_rd_2bits();

            if (x==0){      //Choque
                if (pos_bit==uxq)
                    bd=1;
                else if (pos_bit>uxq)
                    bd=0;
                else{
//                if (pos_bit<uxq){
                    aux=pos_bit-1;
                    pbyte=(aux)/8;
                    pbit=(aux&7);
                    //bd=(rom[pbyte]>>pbit)&BIT0;
                    bd=(mat[cont-1][pbyte]>>pbit)&BIT0;
                }
                if (bd==0)  uzero=pos_bit;
            }
            else if (x==1)  bd=0;
            else if (x==2)  bd=1;
            else            return cont;

            rom[ix]|=0x80*bd;
            if (bd==0)  ow_wr_0();
            else        ow_wr_1();
            i++;
            if (i==8){
                ix++;
                i=0;
            }
        }
        for  (ix=0;  ix<8;  ix++)    mat[cont][ix]=rom[ix];       //Guardar
endereço
        uxq=uzero;
        cont++;
        if (uxq==0)  break;
    }
    return cont;
}

// ler 2 bits para a ROM Search
char ow_rd_2bits(void){
```

```
    char x;
    x=2*ow_rd_bit();
    x+=ow_rd_bit();
    return x;
}

// Calcula CRC para uma certa quantidade de bytes
char ow_crc(char *vt, char qtd) {
  char i,j,crc = 0;
  for (j=0; j<qtd; j++) {
    crc = crc ^ vt[j];
    for (i=0; i<8; i++) crc = crc>>1 ^ ((crc & 1) ? 0x8c : 0);
  }
  if (crc == 0) return TRUE;
  else          return FALSE;
}

// Escrever um bloco de bytes pelo 1-Wire
char ow_wr_blk(char *vt, char qtd){
  char i;
  for (i=0; i<qtd; i++){
    ow_wr_byte(vt[i]);
  }
  return qtd;
}

// Ler um byte pelo 1-Wire


// Ler um bloco de bytes pelo 1-Wire
char ow_rd_blk(char *vt, char qtd){
  char i;
  for (i=0; i<qtd; i++){
    vt[i]=ow_rd_byte();
  }
  return qtd;
}

// Ler um byte pelo 1-Wire
char ow_rd_byte(void){
  char i,dt=0;
  for (i=0; i<8; i++){
    dt=dt>>1;
    if ( ow_rd_bit() == BIT0)  dt |= 0x80;
  }
  return dt;
}

// Ler 1 bit no barramento 1-Wire
char ow_rd_bit(void){
```

```
    char x,i;
    ow_pin_low();
    ow_pin_hiz();
    x=ow_pin_rd();  //Ler barramento
    for (i=0; i<OW_RD; i++)    __no_operation();
    return x;
}

// Enviar um byte pelo 1-Wire
void ow_wr_byte(char dt){
  char i;
  for (i=0; i<8; i++){
    if ((dt&BIT0)==0) ow_wr_0();
    else              ow_wr_1();
    dt = dt>>1;
  }
}

// Escrever 1 no barramento 1-Wire
void ow_wr_1(void){
  char i;
  ow_pin_low();
  ow_pin_hiz();
  for (i=0; i<OW_WR; i++)    __no_operation();
}

// Escrever 0 no barramento 1-Wire
void ow_wr_0(void){
  char i;
  ow_pin_low();
  for (i=0; i<OW_WR; i++)    __no_operation();
  ow_pin_hiz();
  __no_operation();
}

// Gerar Reset e ler resposta do periférico
// TRUE --> OK FALSE --> Não OK
char ow_rst(void){
  char i,aux;
  ow_pin_low();
  for (i=0; i<OW_RST1; i++) __no_operation();
  ow_pin_hiz();
  for (i=0; i<OW_RST2; i++) __no_operation();
  aux=ow_pin_rd();
  for (i=0; i<(OW_RST1-OW_RST2); i++) __no_operation();
  return aux^BIT0;  //1 = presente
}

// Configurar P2.5 para operar com 1-Wire
void ow_config(void){
```

```
    P2DIR |= BIT5;       //P2.5 = saída
    P2OUT &= ~BIT5;      //P2.5 = 0
    P2REN &= ~BIT5;      //Desabilita Pull/up/down
}

// Inicialização Leds e Auxiliar para Osciloscópio
void gpio_config(){
    P1DIR |= BIT0;  P1OUT &= ~BIT0; //Led VM
    P4DIR |= BIT7;  P4OUT &= ~BIT7; //Led VD
    P2DIR |= BIT4;  P2OUT &= ~BIT4; //Scope (P2.4)
}
```

Dizemos então, a grosso modo, que o relógio é de 1 MHz, ou seja, cada instrução consome, pelo menos, 1 µs.

Todo acesso 1-Wire é iniciado com uma sequência de inicialização que consiste em um Como po

o pção do O conceito do protocolo 1-Wire é semelhante ao do I2C, ao fazer uso da sinalização via
com o

Em princípio, o acesso a um dispositivo 1-Wire tem três ou quatro fases, como listado abaixo.

1) **Inicialização:** que é um comando de ressete geral. Todos os dispositivos fazem sua inicialização e ficam esperando algum comando do mestre.
2) **Comando de ROM:** operam sobre um ROM de 64 bits, que é única para cada dispositivo. Permite que o mestre identifique os dispositivos e os questione sobre alguma condição especial.

3) **Comando de Função:** varia de acordo com o dispositivo. Permite ao mestre comandar o dispositivo a realizar a função para o qual ele foi projetado.
4) **Troca de dados (opcional):** Nesta fase, pode acontecer uma troca de dados entre o mestre e o dispositivo.

Escrever

1) Termometro : 28 4C 45 6A 0B 00 00 67 --> 0010 0000 / 0100 1100
2) Termometro bola: 28 2E B7 66 0B 00 00 FD --> 0010 0000 / 0010 1110
3) Termometro 2bola: 28 5E 37 68 0B 00 00 BB --> 0010 0000 / 0101 1110
4) Termometro cabo: 28 FF 37 77 74 16 04 E5 --> 0010 0000 / 1111 1111

Figura H.1. Foto da calculadora HP-25 de seu Handbook (obtidas em HP Museum).