

# 9

# Inter-IC Bus (I<sup>2</sup>C)

Versão 3.0 09/06/2020

A interface I<sup>2</sup>C (*Inter-IC Bus*) foi desenvolvida pela Philips na década de 80. Ela faz uso de apenas 2 fios e permite a conexão de até 127 dispositivos em uma comunicação relativamente rápida e flexível. Cada dispositivo presente neste barramento tem seu próprio endereço. A qualquer momento, apenas dois dispositivos podem usar o barramento, sendo um o mestre e outro o escravo.

O protocolo especifica três possíveis velocidades:

- *Standard-Mode*: até 100 kb/s;
- *Fast-Mode*: até 400 kb/s e
- *High-Speed-Mode*: até 3400 kb/s.

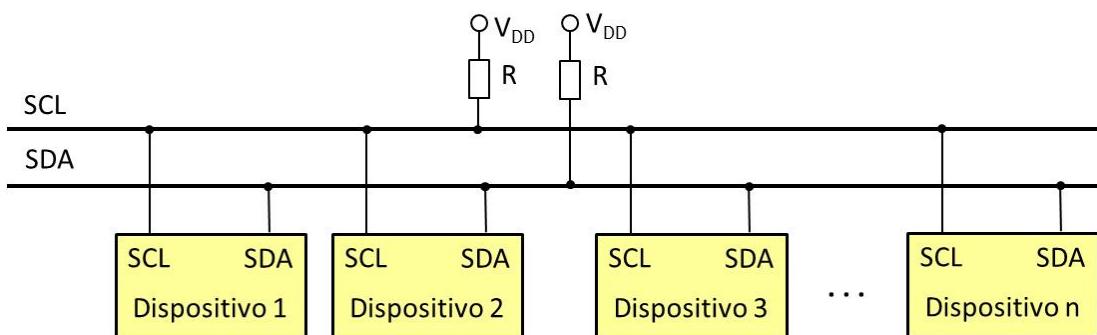
Neste capítulo vamos estudar os recursos que o MSP430 oferece para trabalhar com tal protocolo.

---

## 9.0. Quero Usar a Interface I<sup>2</sup>C e não Pretendo Ler Todo este Capítulo

---

O barramento I<sup>2</sup>C é composto por duas linhas: SDA (*Serial Data*) e SCL (*Serial Clock*), às quais estão conectados todos os dispositivos, como mostrado na figura abaixo. Note a necessidade de dois resistores (R) de *pull-up*, um para cada linha. O valor típico para esses resistores é de 4,7 kΩ.



Um dos dispositivos é o Mestre, responsável por controlar o barramento e os demais são os Escravos. Cada Escravo é identificado por um endereço definido de fábrica. Cada escravo tem, na verdade, dois endereços: um para escrita e outro para leitura. O Mestre usa o endereço de escrita do Escravo para enviar dados ao escravo ou então o endereço de leitura do Escravo para receber dados deste escravo. A leitura ou escrita é definida pelo último *bit* do endereço, vide Figura 9.13.

Toda transação pelo barramento I<sup>2</sup>C é iniciada e terminada pelo Mestre, que endereça um Escravo para enviar ou receber dados. O início de uma transação é marcado pela condição de START e o final pela condição de STOP. Após um START sempre vem o endereço do escravo. O escravo endereçado deve confirmar o endereçamento gerando uma confirmação (ACK) ou uma não-confirmação (NACK) que indica falha no endereçamento. Mais detalhes nas Figuras 9.14 e 9.15.

O dispositivo que recebe os dados, seja ele Mestre ou Escravo, deve gerar um ACK a cada dado recebido. A geração de um NACK é usada para o receptor finalizar a recepção. Assim, o NACK é usado para indicar que se recebeu o último dado. Mais detalhes nas Figuras 9.16 e 9.17. A figura abaixo apresenta uma típica transmissão pelo barramento I<sup>2</sup>C. Note que S = START, P = STOP. Nesta figura, quem terminou a transação foi o transmissor, por isso o último dado foi confirmado com um ACK (e não com um NACK).



O MSP430 possui duas unidades dedicadas para comunicação I<sup>2</sup>C, denominadas de USCI\_B0 e USCI\_B1. Os pinos usados estão na tabela a abaixo, lembre-se de configurar o PxSEL de cada pino.

<b>USCI</b>	<b>SDA</b>	<b>SCL</b>
USCI_B0	P3.0	P3.1
USCI_B1	P4.1	P4.2

A USCI\_Bx (x = 0 ou 1) facilita ao programador a operação do barramento I<sup>2</sup>C. Ela solicita tarefas à USCI e monitora nas *flags* o resultado de cada operação solicitada. É claro que a USCI precisa primeiro ser configurada para depois ser utilizada. A sequência abaixo deve ser obedecida para efetivar a configuração.

1. Ativar o *reset* da USCI\_Bx (UCBx\_CTL1.UCSWRST = 1)
2. Configurar os registradores

3. Configurar as portas
4. Remover o *reset* da USCI\_Bx (UCBx\_CTL1.UCSWRST = 0)
5. Se for o caso, habilitar as interrupções.

Para a operação da USCI\_Bx é preciso seguir os diagramas de tempo apresentados nas figuras listadas abaixo. O que está na metade superior dessas figuras diz respeito ao programador e na metade inferior estão as respostas da USCI.

- Figura 9.23 → Mestre transmissor;
- Figura 9.25 → Mestre receptor;
- Figura 9.27 → Escravo transmissor e
- Figura 9.28 → Escravo receptor.

A seleção do *baud rate* (frequência da linha SCL) é feita no registrador UCBxBRW, de acordo com a fórmula abaixo.

$$\text{Frequência SCL} = \frac{\text{Frequência do relógio selecionado}}{\text{UCBxBRW}}$$

O que se apresentou aqui foi um resumo, recomenda-se a leitura de todo o capítulo para uma melhor compreensão da operação dos recursos I<sup>2</sup>C disponíveis no MSP. Os exercícios resolvidos, apresentados mais adiante, auxiliam nessa compreensão. Ao final deste capítulo está um gabarito para o leitor, com facilidade, configurar os registradores.

## 9.1. Fundamentos do Barramento I<sup>2</sup>C

Antes de iniciar o estudo dos recursos I<sup>2</sup>C oferecidos pelo MSP430, precisamos de alguns conceitos, em especial do emprego de dreno (ou coletor) aberto. Os circuitos digitais fazem uso intenso de transistores, que podem ser de Efeito de Campo (FET) ou Bipolar. O transistor FET é também designado como MOS ou MOSFET, que faz referência à sua tecnologia de fabricação: **Metal-Óxido-Semicondutor**. A Figura 9.1 mostra a representação esquemática de cada tipo desses transistores. Aqui será feito um estudo muito simples de tais dispositivos.

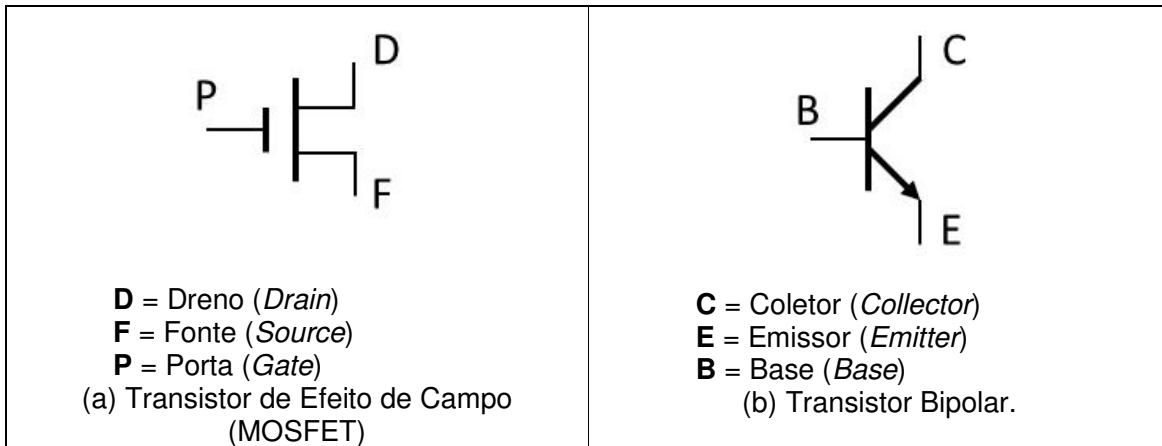


Figura 9.1. Dois tipos de transistores.

Para esta abordagem do barramento I<sup>2</sup>C, vamos apenas estudar o comportamento do transistor como uma chave digital. O controle é feito pela Porta (G) ou pela Base (B), de acordo com o tipo do transistor, conforme mostrado na Tabela 9.1. A explicação aqui apresentada é extremamente simplificada, mas atende à nossa finalidade.

Tabela 9.1. Lógica de condução dos transistores

MOS		Bipolar	
G = nível alto	Conduzindo	B = nível alto	Conduzindo
G = nível baixo	Cortado	B = nível baixo	Cortado

Os transistores são usados para construir a saída de portas lógicas. Vamos aqui exemplificar a porta de 3 estados (*tri-state*), onde a saída pode ir para ZERO, para UM, ou para ALTA IMPEDÂNCIA (Hi-Z). O controle da condução dos transistores Q1 e Q2 permite a geração desses 3 estados, como mostrado na Figura 9.2.

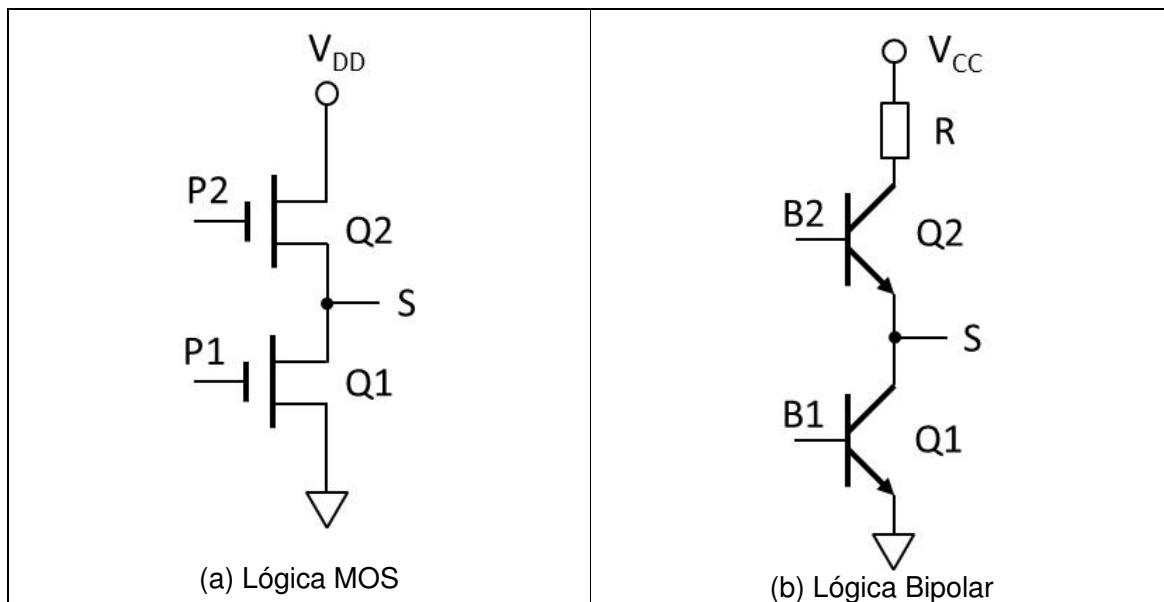


Figura 9.2. Lógica de saída de um circuito digital. O retângulo marcado com a letra “R” representa um resistor. Esta figura está muito simples e serve apenas para ilustrar o conceito.

O estado da saída (S) pode ser controlado com a conveniente aplicação de nível lógico alto (1) ou baixo (0) nas portas (P1 e P2) ou nas bases (B1 e B2) dos transistores. A Tabela 9.2 apresenta todas as possibilidades. Nesta tabela foi usado o “1” para representar o nível alto e o “0” para representar o nível baixo. Se apenas o transistor Q1 está conduzindo, é formado um circuito de baixa impedância para a terra e com isso a saída S vai para zero. Por outro lado, caso apenas o transistor Q2 conduza, forma-se um circuito de baixa impedância para  $V_{CC}$  ( $V_{DD}$ ) e com isso a saída S vai para um. Quando Q1 e Q2 estão cortados, não há circuito de baixa impedância nem para  $V_{CC}$  ( $V_{DD}$ ) e nem para a terra. Nesse caso, diz-se que a saída S está “flutuando”, ou que ela está em um estado de alta impedância (Hi-Z). Evidentemente, a opção de Q1 e Q2 conduzindo não é útil, pois forma-se um circuito de baixa impedância entre a alimentação ( $V_{DD}$  ou  $V_{CC}$ ) e a terra.

Tabela 9.2. Lógica da saída S

Situação	P1 (B1)	P2 (B2)	Q1	Q2	S
1	1	0	Conduz	Cortado	0
2	0	1	Cortado	Conduz	1
3	0	0	Cortado	Cortado	Hi-Z
4	1	1	Conduz	Conduz	Não usar

Vamos agora analisar as transições na saída, ou seja, o que acontece quando ela muda de 0 para 1 ou de 1 para 0. Podemos dizer que quando a saída S muda de 0 para 1, o transistor Q2 injeta corrente para levar a saída para  $V_{CC}$  ( $V_{DD}$ ). Por outro lado, quando a saída S muda de 1 para 0, o transistor Q1 drena corrente para levar a saída para zero. O transistor Q2 recebe o nome de “pull-up ativo”, porque ele é responsável por levar saída S para  $V_{CC}$  ( $V_{DD}$ ). Por ser um transistor, ele pode conduzir muito no início da transição e depois diminuir a condução, por isso é chamado de ativo. É importante que Q2 conduza muito no início da transição para garantir a velocidade e depois que a saída está em nível alto, Q2 pode diminuir sua condução.

Agora vamos fazer o experimento de remover o transistor Q2, como mostrado na Figura 9.3. Agora se diz que a saída é a “Dreno Aberto”, no caso MOS, ou a “Coletor Aberto”, no caso bipolar. O termo “aberto” é porque o dreno ou o coletor de Q1 não está conectado. Este caso leva a duas condições de saída, como mostrado na Tabela 11.3.

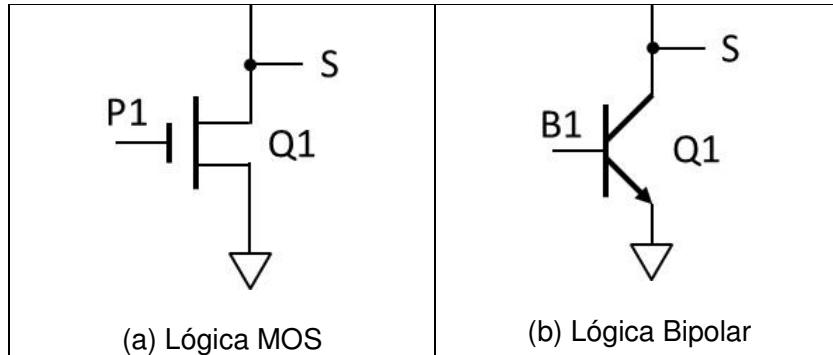


Figura 9.3. A saída a Dreno aberto e a Coletor aberto.

Tabela 9.3. Lógica da saída S, no caso Dreno aberto ou Coletor aberto

Situação	P1 (B1)	Q1	S
1	1	Conduz	0
2	0	Cortado	Hi-Z

Como era esperado, quando Q1 conduz, a saída vai para 0. Entretanto, quando Q1 está cortado, a saída flutua, ou seja, vai para o estado de alta impedância (Hi-Z). Esta situação não parece ter grande utilidade, pois foi criada uma “porta” que só pode ir para zero e nunca para um. Na verdade, este circuito tem diversas utilidades que fogem ao assunto que aqui nos interessa.

A ideia agora colocar um “*pull-up*” na saída, mas um “*pull-up*” passivo, ou seja, um resistor. Assim, chega-se a uma situação muito parecida com a da Figura 9.2 e Tabela 9.1. A Figura 9.4 apresenta este caso. Agora está presente o resistor R, que leva a saída S para V<sub>CC</sub> (V<sub>DD</sub>) quando Q1 estiver cortado. A saída agora funciona de forma semelhante à apresentada na Tabela 9.2, porém a transição da saída S de 0 para 1 é mais lenta, pois é feita com o uso de um *pull-up* passivo (o resistor).

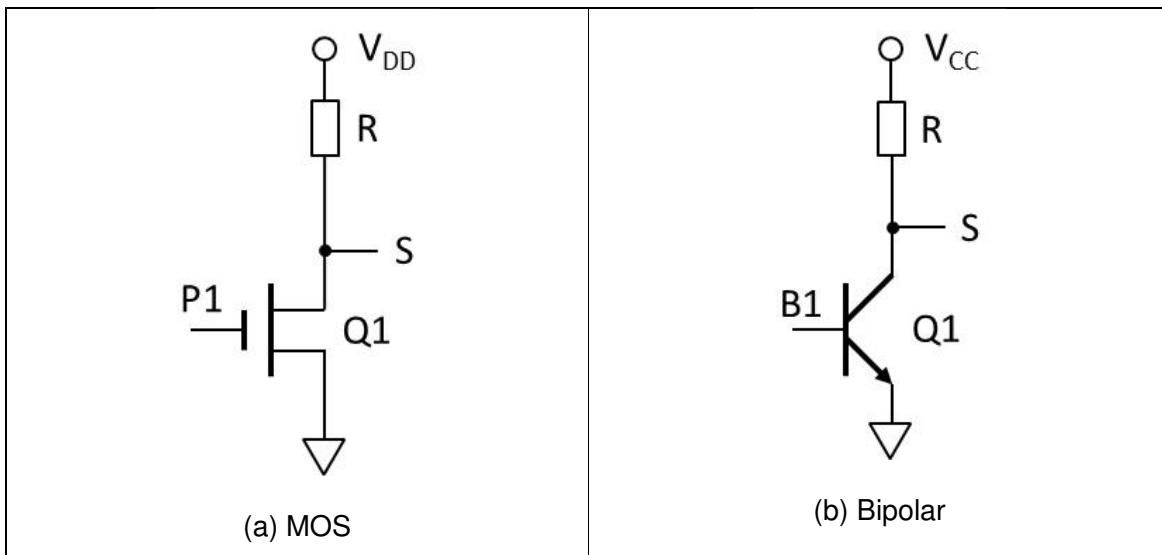


Figura 9.4. Lógica de saída de um circuito digital com saída a coletor aberto e usando um resistor (R) de *pull-up*.

Na maioria dos casos, o valor de R não é crítico. Além disso, não é nosso objetivo indicar como calculá-lo. Entretanto, faremos um pequeno juízo sobre seu valor. Quanto menor for o valor de R, mais rápida será a transição de 0 para 1 na saída S. Por outro lado, quanto menor for o valor de R, maior será o consumo de corrente quando a saída estiver em 0. Note que se Q1 conduz, temos apenas R entre V<sub>CC</sub> (V<sub>DD</sub>) e Terra. Assim, existe uma relação de compromisso no cálculo do valor de R.

A figura 9.5 ilustrada de forma aproximada o tempo de subida para o caso com *pull-up* ativo e para o caso com *pull-up* passivo. Fica claro que saída com *pull-up* passivo é mais lenta. Ela se assemelha à carga de um capacitor. Quanto menor for o valor do resistor, mais rápida é a subida. As siglas significam:

- V<sub>OL</sub> → tensão de saída em nível baixo e
- V<sub>OH</sub> → tensão de saída em nível alto.

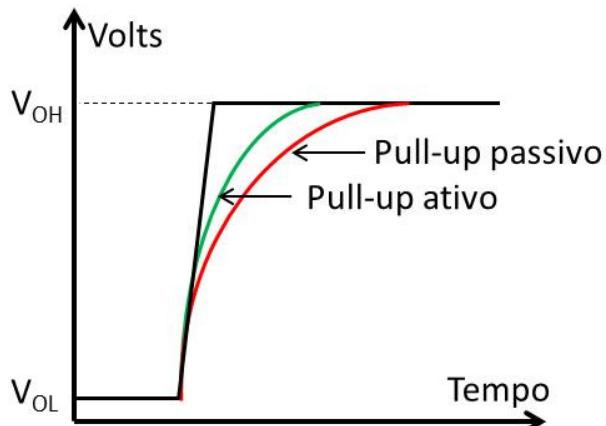


Figura 9.5. Ilustração do tempo de subida (transição de 0 para 1) numa saída digital.

Uma aplicação interessante de dreno (coletor aberto) surge quando conectamos duas saídas a dreno (coletor) aberto, como mostrado na Figura 9.6.a. Foram empregados dois transistores, Q1 e Q2, e apenas um resistor de *pull-up*. Temos que as duas saídas S1 e S2 foram curto-circuitadas, dando origem à saída S. Vamos fazer uma análise de S em função de S1 e S2. A Tabela 9.4 resume as possíveis combinações de S1 e S2.

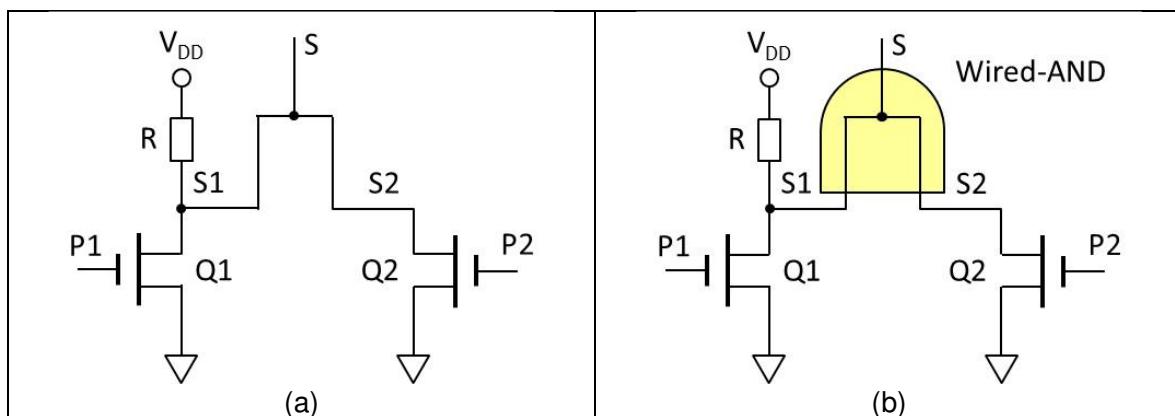


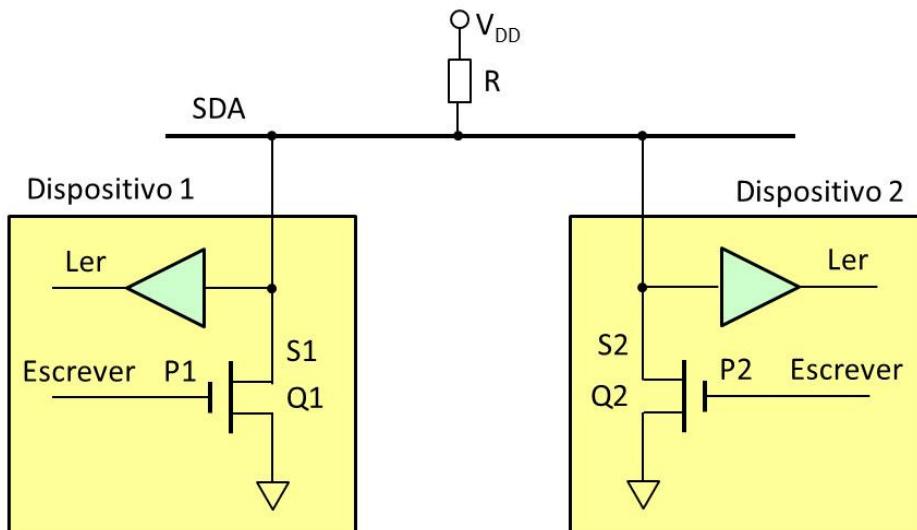
Figura 9.6. Conexão de duas saídas a dreno aberto, dando origem à saída Wired-AND.

Ao examinarmos o circuito da Figura 9.6.a, fica claro que basta que um dos dois transistores conduza, para que a saída S vá para 0. Isto significa que a única opção para termos  $S = 1$  é com os dois transistores cortados. Em outras palavras  $S = 1$  somente quando  $S1 = 1$  e  $S2 = 1$ . Ora, isto é a definição de uma porta AND. Assim, a conexão de S1 com S2 fez surgir uma porta AND, como mostrado na Figura 9.6.b, daí seu nome *Wired-AND*.

Tabela 9.4. Lógica da saída S em função de S1 e S2

Situação	Q1	Q2	S1	S2	S
1	Conduz	Conduz	0	0	0
2	Conduz	Cortado	0	1	0
3	Cortado	Conduz	1	0	0
4	Cortado	Cortado	1	1	1

O mesmo acontece se usarmos vários transistores: a saída só vai para 1 quando todos os transistores estão cortados. Isto permite uma maneira fácil de conseguir comunicação bidirecional que foi aproveitada pelo protocolo I<sup>2</sup>C. Em tal tipo de comunicação, uma determinada linha ora se comporta como saída, ora se comporta como entrada. A Figura 9.7 apresenta o caso típico de conexão de dois dispositivos I<sup>2</sup>C. É importante deixar claro que vamos explicar o funcionamento apenas com dois dispositivos, entretanto, o protocolo permite até 127 dispositivos. O valor do resistor R, tipicamente, está entre 2 kΩ a 10 kΩ.

Figura 9.7. Conexão de dois dispositivos I<sup>2</sup>C.

A Figura 9.7 exemplifica a conexão de dois dispositivos à linha SDA do barramento I<sup>2</sup>C. O pequeno triângulo representa o *buffer* de leitura. Por exemplo, o Dispositivo 1, ao controlar seu transistor Q1, pode escrever na linha SDA. Ele deixa o transistor Q1 cortado para escrever 1 e o coloca em condução quando precisa escrever 0. Através de seu *buffer* de leitura, o Dispositivo 2 consegue ler o estado da linha SDA.

Por exemplo, para o Dispositivo 1 receber uma informação do Dispositivo 2, ele precisa cortar seu transistor Q1. Assim, o Dispositivo 2, usando seu transistor Q2 consegue fazer a linha SDA igual a 0 ou igual a 1. O protocolo especifica que, por vez, somente um dispositivo pode colocar dados no barramento. Assim, todos os dispositivos presentes cortam seus transistores, à exceção daquele que vai enviar as informações.

A Figura 9.8 é um pouco mais técnica. Ela apresenta a forma de se medir o Tempo de Subida numa transição de 0 para 1. Os símbolos significam:

- $V_{OL}$  → tensão de saída da porta em nível baixo;
- $V_{IL}$  → tensão para a entrada da porta interpretar um nível baixo (zero);
- $V_{IH}$  → tensão para a entrada da porta interpretar um nível alto (um) e
- $V_{DD}$  → tensão de alimentação do circuito.

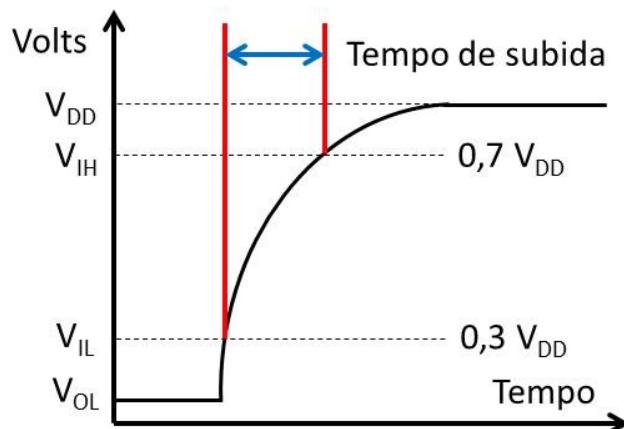


Figura 9.8. Caracterização do Tempo de Subida (Rise Time) numa transição de 0 para 1.

A Tabela 9.5 apresenta detalhes técnicos para as diversas velocidades do protocolo I<sup>2</sup>C. Cada dispositivo que é adicionado ao I<sup>2</sup>C aumenta a capacitância do barramento e em consequência o tempo de subida. Assim, há uma limitação para a capacitância total do barramento e o tempo de subida aceitável. Note também que existem duas velocidades "High-Speed". O filtro de ruído é usado para evitar disparos acidentais. Na velocidade mais baixa, ele não se faz necessário.

Tabela 9.5. Principais parâmetros do protocolo I<sup>2</sup>C

Parâmetros	Standard	Fast	High –Speed	High-Speed
Velocidade (kBits/seg)	0 a 100	0 a 400	0 a 1700	0 a 3400
Máxima capacitância (pF)	400	400	400	100
Tempo de subida (ns)	1000	300	160	80

Filtro ruído ( $\eta$ s)	-	50	10	10
--------------------------	---	----	----	----

Observação: a porta I<sup>2</sup>C disponível no MSP430 não opera na velocidade High-Speed.

A construção de um barramento para comunicação I<sup>2</sup>C é muito simples. Todo dispositivo I<sup>2</sup>C tem dois pinos: SDA (Serial Data) e SCL (Serial Clock), que devem ser conectados, como mostrado na Figura 9.9. Note a necessidade de dois resistores de *pull-up*, um para cada linha. Tipicamente, esses resistores têm valor entre 2 k $\Omega$  e 10 k $\Omega$ . O valor típico é de 4,7 k $\Omega$ . No caso do MSP430, muitas vezes, o *pull-up* interno das portas é suficiente, o que dispensaria o *pull-up* externo.

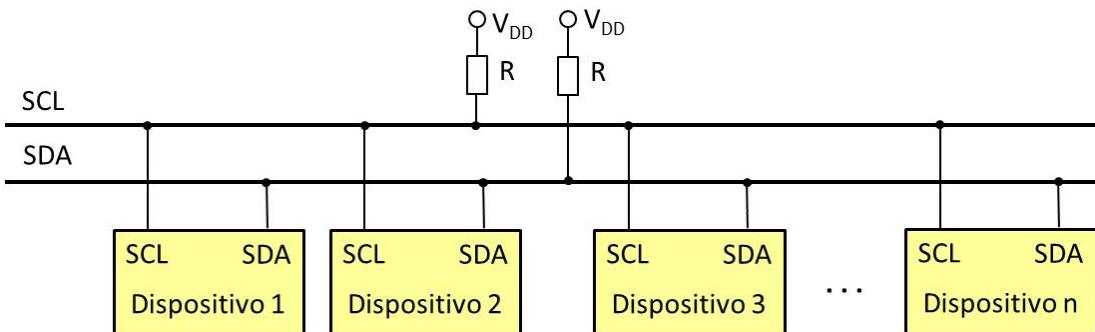


Figura 9.9. Composição típica de um barramento I<sup>2</sup>C.

### 9.1.1. Resumo

O barramento I<sup>2</sup>C é construído com apenas dois condutores, o que lhe dá uma grande facilidade para uso. Por outro lado, como será visto adiante, é preciso de uma sinalização inteligente, para se conseguir enviar e receber informações por apenas esses dois condutores. As duas linhas trabalham com coletor ou dreno aberto, daí a necessidade de resistores de *pull-up*. Como operam em coletor ou dreno aberto, cada linha (SDA ou SCL) só vai para nível alto se todos os dispositivos a ela conectados mantiverem suas respectivas saídas em nível alto. Se uma das linhas está em nível alto, um dispositivo qualquer pode levá-la para nível baixo. Por outro lado, se uma das linhas está em nível baixo, ela só volta para nível alto quando todos os dispositivos a ela conectados estiverem com suas saídas em nível alto.

## 9.2. Protocolo I<sup>2</sup>C

O barramento I<sup>2</sup>C foi apresentado na Figura 9.9 e opera com dreno ou coletor aberto. Novamente, enfatizamos que:

Ricardo Zelenovsky e Daniel Café

- A linha (SCL ou SDA) vai para nível baixo quando um dos dispositivos a ela conectado faz sua saída igual a zero.
- A linha (SCL ou SDA) só vai para nível alto quando todos os dispositivos a ela conectados estão com suas saídas em um.

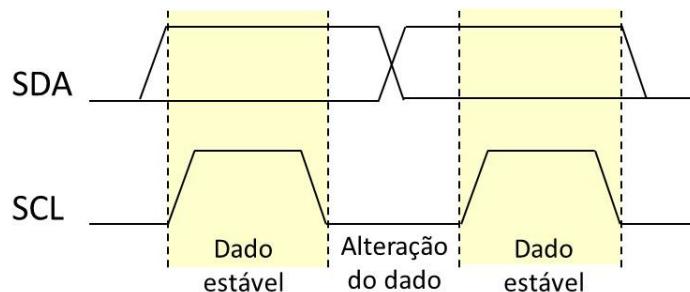
Para se entender o protocolo, é necessário definir uma pequena terminologia, que está apresentada na Tabela 9.6.

*Tabela 9.6. Terminologia para o Barramento I<sup>2</sup>C*

Termo	Descrição
Mestre	É o dispositivo que inicia e termina a transmissão. Ele é responsável por gerar o relógio SCL.
Escravo	Dispositivo endereçado pelo Mestre.
Transmissor	Dispositivo que coloca dados no barramento.
Receptor	Dispositivo que lê os dados do barramento.

### 9.2.1. Transferência de Bits

A linha SDA é destinada a transferir os *bits* serialmente, enquanto que a linha SCL indica o instante em que existe um *bit* válido na linha SDA. Durante uma transferência, a indicação de que existe um *bit* válido na linha SDA é feita com a linha SCL em nível alto. Enquanto a linha SCL está em nível alto, a linha SDA deve permanecer estável. Em outras palavras, a linha SDA só pode ser alterada com a linha SCL em nível baixo. Isto está mostrado na Figura 9.10. Existem duas exceções: na geração da condição de START e da condição de STOP a linha SDA pode mudar mesmo com a linha SCL em nível alto.



*Figura 9.10. Validação do dado na linha SDA.*

### 9.2.2. Condição de START e STOP

O Mestre é o responsável por iniciar e terminar toda transferência de dados. O Mestre inicia uma transferência com uma condição de START e a termina com uma condição STOP. No intervalo entre um START e um STOP, o barramento é considerado ocupado e nenhum outro Mestre pode usá-lo. Assim, após um STOP, o barramento está livre.

O truque do protocolo I2C para marcar o início e fim de uma transmissão foi o de permitir a alteração da linha SDA enquanto a linha SCL está em nível alto. Como pode ser visto na Figura 9.11, a Condição de START (representada pela letra S) é gerada com uma transição de alto para baixo na linha SDA enquanto a linha SCL está em nível alto. Por sua vez, a Condição de STOP (representada pela letra P) é gerada com uma transição de baixo para alto na linha SDA enquanto a linha SCL está em nível alto.

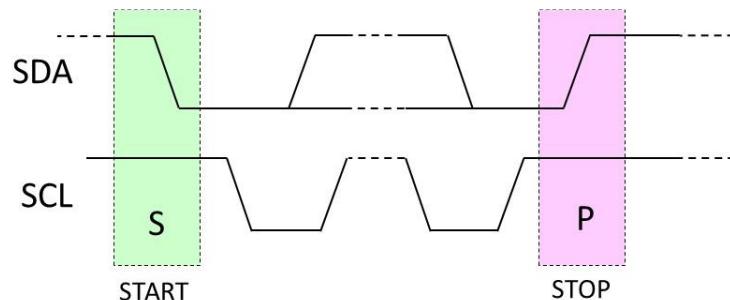


Figura 9.11. Definição das Condições de START (S) e STOP (P).

Toda transação inicia com um START e termina com um STOP. Entre um START e um STOP, o barramento está ocupado. Após o STOP, o barramento está livre. Para o caso em que o Mestre deseja iniciar uma nova transferência sem liberar o barramento ele pode usar a condição de START REPETIDO, ou seja, ao invés de gerar o STOP, ele gera um novo START, como mostrado na Figura 9.12. Para todos os efeitos de comportamento do barramento, o START REPETIDO se parece com um START. Assim, daqui em diante, o termo START também designará o START REPETIDO.

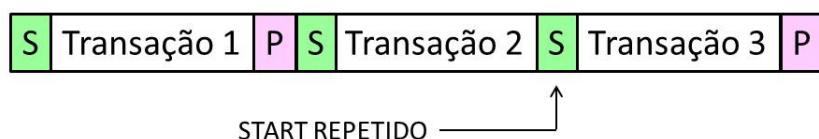


Figura 9.12. Emprego das Condições de START (S), de STOP (P) e de START REPETIDO (S).

### 9.2.3. Formato do Pacote de Endereço

O mestre usa o endereço para indicar qual escravo vai participar da transação que foi iniciada. O pacote de endereço tem 7 bits, mais um bit para especificar se a operação é de leitura ou de escrita, vide Figura 9.13. Isto permite um espaço de 128 endereços ( $128 = 2^7$ ). Pode-se encarar de outra forma: cada dispositivo tem dois endereços, sendo um para leitura e outro para escrita. Por simplicidade vamos adotar a seguinte representação:

- SLA+R → Endereço para leitura (*Slave Address plus Read*) e
- SLA+W → Endereço para escrita (*Slave Address plus Write*)

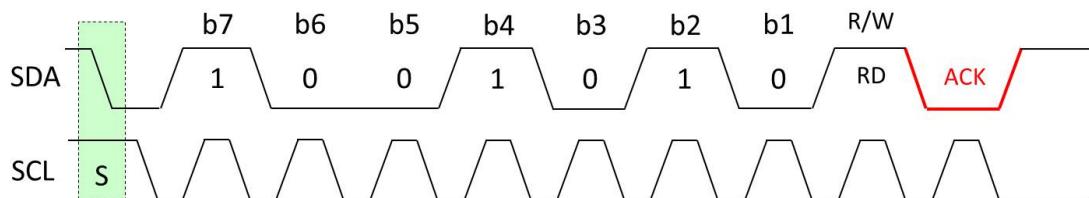
A Figura 9.13 apresenta como exemplo o endereço de 7 bits 0x4A (100 1010 em binário) e a formação do endereço 0x95 (para leitura). Este dispositivo hipotético teria dois endereços: 0x95 para leitura e 0x94 para escrita. É importante notar que cada dispositivo vem de fábrica com um endereço já programado.

b7	b6	b5	b4	b3	b2	b1	R/W
1	0	0	1	0	1	0	1

$R/\bar{W} = 1 \rightarrow$  Leitura  
 $R/\bar{W} = 0 \rightarrow$  Escrita

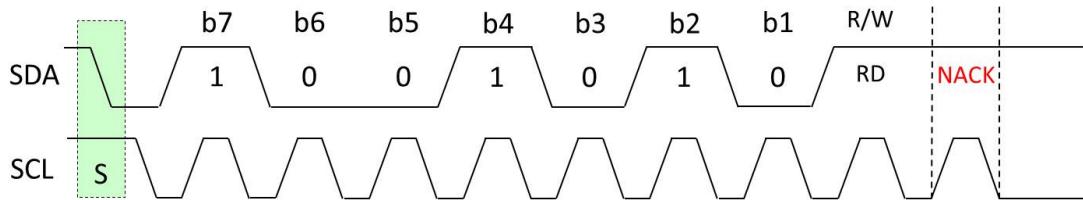
Figura 9.13. Exemplo do endereço 0x95, ou seja, endereçando o dispositivo 0x4A para leitura.

O dispositivo endereçado tem a obrigação de responder ao endereçamento com uma confirmação (ACK = *acknowledge*). Para fazer isto, o dispositivo endereçado coloca a linha SDA em nível baixo por ocasião do último pulso de SCL. A Figura 9.14 mostra o envio do endereço 0x95 pelo barramento e a consequente confirmação (ACK em vermelho) emitida pelo dispositivo endereçado. Note que, de cada byte, o bit mais significativo é enviado primeiro. Todos os sinais da Figura 9.14 foram gerados pelo Mestre do barramento, à exceção do ACK que foi gerado pelo dispositivo endereçado (Escravo). Assim, o Mestre fica sabendo se o dispositivo está pronto operar. Se o dispositivo endereçado está ocupado ou se não pode responder ao endereçamento por alguma razão, ele deve manter a linha SDA em nível alto, o que caracteriza a não confirmação (NACK).



*Figura 9.14. Exemplo do endereço 0x95 enviado pelo barramento I<sup>2</sup>C e a confirmação (ACK) emitida pelo dispositivo endereçado.*

A Figura 9.15 apresenta o caso onde, por alguma razão, o dispositivo endereçado não pôde responder e por isso não enviou a confirmação (NACK). Neste caso, o dispositivo endereçado nada faz, simplesmente deixa a linha SDA em nível alto.



*Figura 9.15. Exemplo do endereço 0x95 colocado no barramento I<sup>2</sup>C e a não confirmação (NACK) emitida pelo dispositivo endereçado.*

Os fabricantes de dispositivos I<sup>2</sup>C têm liberdade para designar os 7 Bits de endereço. Porém o endereço zero (000 0000 em binário) deve ser reservado para a chamada geral. Endereços do tipo 111 1xxx (em binário) devem ser reservados para emprego futuro.

Extensão do SCL: Um ponto interessante para a compatibilização de velocidade é o fato de o escravo poder estender o tempo em que a linha SCL permanece em nível baixo. Graças à lógica de dreno aberto (ou de coletor aberto), se o escravo coloca a linha SCL em nível baixo, ela assim permanece, mesmo que o mestre tente mudá-la para nível alto. O mestre só pode continuar o envio dos dados com a linha SCL em nível alto. Isto também pode ser usado para o escravo, momentaneamente, interromper uma transmissão. Note que esta ação não influi no tempo em que a linha SCL permanece em nível alto. Tal recurso vai ajudar na arbitragem, que será visto na seção 9.5.

#### 9.2.4. Formato do Pacote de Dados

Todo pacote de dados tem 8 bits, mais um bit de confirmação (ACK) enviado pelo receptor. O relógio (SCL) é sempre gerado pelo Mestre e cadencia a colocação dos bits no barramento. Podemos exemplificar duas situações: o escravo recebendo dados enviados pelo mestre e, no caso contrário, o mestre recebendo dados enviados pelo escravo. As Figuras 9.16 e 9.17 apresentam os dois casos, considerando que o dado a ser enviado é o byte 0xB9 (1011 1001 em binário).

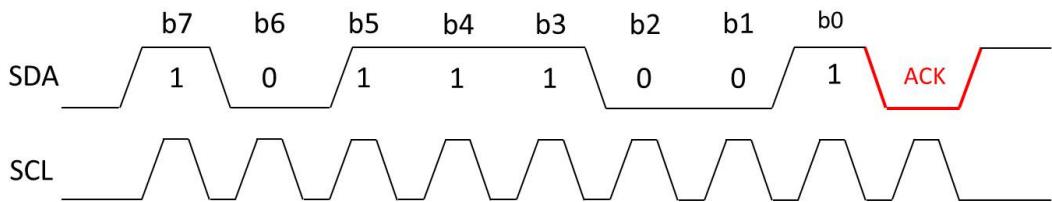


Figura 9.16. Exemplo do byte 0xB9 (1011 1001) sendo transmitido pelo mestre e recebido por um escravo, que gerou a confirmação ACK. Os sinais gerados pelo escravo estão em vermelho e com uma linha mais grossa.

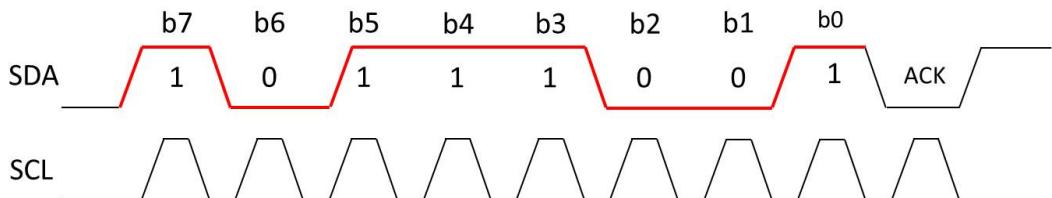


Figura 9.17. Exemplo do byte 0xB9 (1011 1001) sendo transmitido por um escravo e recebido pelo mestre, que gerou a confirmação ACK. Os sinais gerados pelo escravo estão em vermelho e com uma linha mais grossa.

Ao final da transmissão, como mostrado nas figuras acima, o dispositivo que recebe os dados deve gerar a confirmação, levando a linha SDA para nível baixo durante o próximo SCL. Se o receptor deixar a linha SDA em nível alto, significa uma não confirmação (NACK). Em ambos os casos, note que o mestre é o responsável por acionar a linha SCL.

### 9.2.5. Transmissão pelo Barramento I<sup>2</sup>C

Uma transmissão consiste, basicamente, de uma condição de START, um endereço (para leitura ou escrita), um ou mais pacotes de dados e uma condição de STOP. Todas essas etapas estão mostradas na Figura 9.18. O protocolo proíbe uma condição de START seguida por uma condição de STOP, ou seja, uma transmissão sem a fase de dados. Deve ser observado que após um START, vários dados podem ser transmitidos pelo barramento, cada um deles seguido por uma confirmação (ACK). Após a confirmação do último dado, o mestre gera uma condição de STOP.



Figura 9.18. Típica transmissão pelo barramento I<sup>2</sup>C.

Lembrando do que foi dito no item 9.2.2, caso o mestre queira finalizar a transmissão sem liberar o barramento, ao invés da condição de STOP, ele pode gerar um novo START.

### 9.2.6. Arbitragem do Barramento I<sup>2</sup>C

O barramento I<sup>2</sup>C permite a presença de vários mestres. Deve ser lembrando que quando o barramento está ocioso, qualquer mestre pode usar o barramento. O problema surge quando dois ou mais mestres tentam usar ao mesmo tempo o barramento que está ocioso. Para fazer frente a esta situação, deve ser planejada uma forma de garantir que, por vez, somente um mestre use o barramento. A isso se dá o nome de *arbitragem*.

A lógica a dreno (ou coletor) aberto ajuda a resolver esse problema de forma bem simples. É importante lembrar que uma linha I<sup>2</sup>C (SDA ou SCL) só está em nível alto se todos os dispositivos a ela conectados colocarem suas saídas em nível alto. Por outro lado, basta um dispositivo colocar sua saída em nível baixo para que a linha I<sup>2</sup>C (SDA ou SCL) vá para nível baixo. No caso da arbitragem, é importante lembrar da extensão da linha SCL, vista no item 9.2.3 e exemplificado na Figura 9.19.

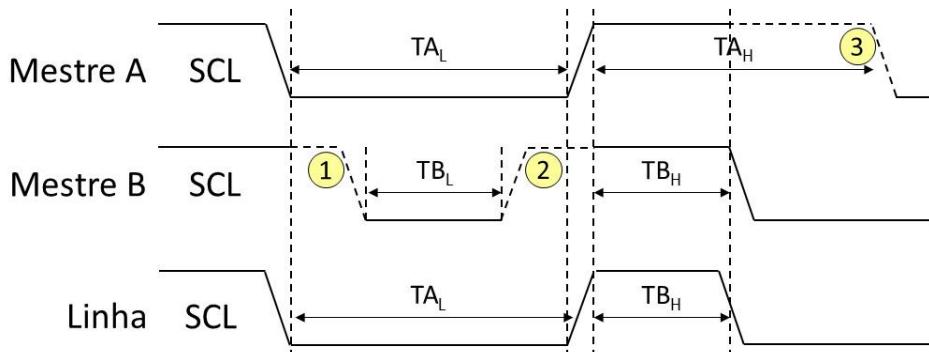


Figura 9.19. Linha SCL: sincronização entre vários mestres.

A explicação a seguir é importante para a arbitragem. Na Figura 9.19 temos dois Mestres, denominados A e B acionando a linha SCL. A figura mostra a intenção de cada mestre e o resultado na linha SCL. O mestre A trabalha com um relógio mais lento que o do mestre B. No início, o mestre A abaixou a linha SCL, já o mestre B pretendia abaixá-la um pouco mais tarde (instante 1). No instante 2, o mestre B quis levantar a linha SCL, mas isso só aconteceu quando o mestre A a levantou. No instante 3, o mestre A pretendia deixar a linha em nível alto por mais algum tempo, mas não pode porque o mestre B a abaixou. Note que o tempo em que a linha SCL ficou em nível baixo foi ditado pelo mestre mais

lento (mestre A) e o tempo em que ela permaneceu em nível alto foi ditado pelo mestre mais rápido (mestre B).

É importante ressaltar que tudo isso que aconteceu não caracteriza condição de erro, mas sim de sincronização. O barramento deve trabalhar em uma das velocidades listada na Tabela 9.5. O fato de um mestre alterar a linha mais cedo ou mais tarde, não gera problema algum.

Abordemos agora a arbitragem. Dois ou mais mestres podem tentar usar o barramento ao mesmo tempo. Eles enviam a condição de START e vamos considerar o pior caso que é quando todos tem sucesso. Após isso eles vão tentar enviar um endereço ou um dado após o endereço. Cada mestre coloca seu dado na linha SDA e a monitora. O mestre que não conseguiu levar a linha SDA para o estado desejado deve sair da disputa. Ao final, somente um mestre ficará presente. O mestre que abandona a arbitragem deve entrar imediatamente em modo escravo, pois ele pode estar sendo endereçado.

Enquanto os mestres que disputam o barramento estiverem enviando os mesmos dados, eles permanecem na disputa. Como é óbvio, um mestre só percebe que deve abandonar a disputa ao falhar em tentar colocar a linha SDA em nível alto. O mestre sempre tem sucesso em colocar a linha SDA em nível baixo. O fato dos mestres gerarem SCL com diferentes períodos não causa problema. Como já vimos, o período em que SCL fica em nível baixo é ditado pelo mestre mais lento e o período em que SCL fica em nível alto é ditado pelo mestre mais rápido.

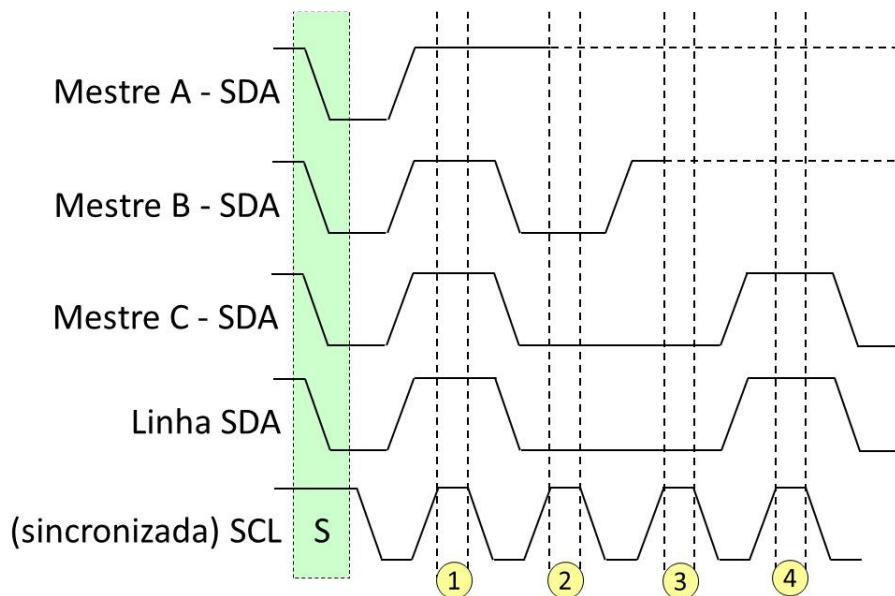


Figura 9.20. Arbitragem do barramento I<sup>2</sup>C envolvendo a disputa de 3 mestres.

A Figura 9.20 apresenta o caso hipotético em que três mestres, denominados de A, B e C tentam ganhar o barramento. No início todos os mestres geram a condição de START, mais ou menos juntos e todos têm sucesso. Após isso, no instante 1, os três mestres enviam o mesmo *bit* 1, cada um colocando sua linha SDA em nível alto. Como os 3 coincidem, todos permanecem na disputa. No instante 2, o mestre A tenta deixar a linha SDA em nível alto, porém os outros dois colocam suas saídas SDA em nível baixo. Com isso, quando a linha SCL vai para nível baixo, o mestre A percebe que não conseguiu controlar a linha SDA e sai da disputa. No instante 3, o mesmo acontece com o mestre B. Assim, do instante 4 em diante o único mestre presente no barramento é o mestre C. Note que um mestre só pode perder a disputa pelo barramento ao tentar colocar a linha SDA em nível alto.

A arbitragem prossegue até ficar um só mestre presente. Essa disputa pode envolver toda a fase de endereçamento e seguir pela fase de envio de dados. O mestre que perde a disputa deve entrar imediatamente no modo escravo, pois a operação pode estar endereçando ele próprio.

A arbitragem é proibida entre:

- Uma condição de START REPETIDO e um *bit* de dado;
- Uma condição de STOP e um *bit* de dado e
- Uma condição de START REPETIDO e um STOP.

É responsabilidade do programa garantir que essas condições proibidas nunca aconteçam. Assim, em uma situação de múltiplos mestres, deve-se usar a mesma composição de endereçamento e pacote de dados, ou seja, todas as transmissões devem conter o mesmo número de pacotes de dados. Do contrário, o resultado da arbitragem será indefinido.

### 9.3. Módulos USCI do MSP430

A arquitetura MSP430 disponibiliza módulos de Interface de Comunicação Serial Universal, denominados de USCI (do inglês *Universal Serial Communication Interface*). Esses módulos atendem a diversos protocolos de comunicação serial. São usadas letras para diferenciar os diferentes módulos. As instâncias de um mesmo módulo são diferenciadas com números. Assim, temos USCI\_A0, USCI\_A1, USCI\_B0 etc.

O módulo USCI\_Ax ( $x = 0, 1, 2, \dots$ ) suporta:

- Modo UART,
- Modo IrDA e
- Modo SPI.

O módulo USCI\_Bx ( $x = 0, 1, 2, \dots$ ) suporta:

- Modo I<sup>2</sup>C e
- Modo SPI.

O MSP430F5529 oferece quatro módulos: USCI\_A0, USCI\_A1, USCI\_B0 e USCI\_B1

## 9.4. Módulos USCI\_Bx do MSP430F5529

Como vimos no tópico anterior, o módulo USCI\_Bx é o único capaz de operar com o protocolo I<sup>2</sup>C, sendo que o MSP430F5529 oferece duas instâncias desse módulo. A tabela abaixo indica os pinos onde as linhas SDA e SCL são disponibilizadas.

*Tabela 9.7. Disponibilidade das linhas SDA e SCL (LaunchPad)*

Módulo	SDA	SCL
USCI_B0	P3.0	P3.1
USCI_B1	P4.1	P4.2

O módulo USCI\_B0 faz uso de P3.0 e P3.1 e, para tanto, é preciso usar o controle do GPIO e fazer P3SEL.0 = 1 e P3SEL.1 = 1. O valor de P3DIR.0 e P3DIR.1 não importa.

O emprego do módulo USCI\_B1 é um pouco mais sofisticado, pois seus sinais são disponibilizados com o mapeamento da porta P4, vide item 3.5 e Tabela 3.4 do Capítulo 3. Isto significa que podemos mapear as linhas SCL e SDA em qualquer bit da porta P4. Porém, a LaunchPad apenas disponibiliza os bits 0, 1, 2 e 3 da porta P4. Para ficarmos coerente as indicações de pinagem presentes na foto (vide Figura 3.22) que acompanha a LaunchPad, vamos dar preferência aos pinos P4.1 e P4.2.

*Tabela 9.8. Código para mapeamento linhas SDA e SCL na Porta P4*

Valor	Mnemônico	Função
14	PM_UCB0SCL	SCL
15	PM_UCB0SDA	SDA

Ao que se constatou até agora, a inicialização da LaunchPad já faz o mapeamento dos pinos P4.1 e P4.2. Entretanto, apresentamos abaixo um trecho de código para mapear os pinos P4.1 (SDA) e P4.2 (SCL)

```

P4DIR |= BIT2 | BIT1;           //P4.1 e P4.2 como saídas
P4SEL |= BIT2 | BIT1;           //P4.1 e P4.2 para função alternativa
MOV #0x02D52, &PMAPKEYID      //Chave para liberar mapeamento
P4MAP1 = PM_UCB1SDA;          //P4.1 = SDA
P4MAP2 = PM_UCB1SCL;          //P4.2 = SCL

```

Listamos a seguir as principais características do módulo USCI\_B.

- Endereçamento de 7 ou 10 bits;
- Chamada geral (*General Call*);
- Suporte para operar em 100 kbps e 400 kbps;
- Frequência de operação programável;
- Recursos para operar em baixo consumo e
- Pode despertar com condição de START.

A seguir faremos um breve estudo do diagrama de blocos da USCI\_Bx. Iniciamos com a seleção do relógio (UCxSCL), que está apresentada na Figura 9.21. Nesta figura, as caixas em verde são configurações que o programador controla. O multiplexador permite que se selecione entre UC1CLK, ACLK ou SMCLK (que está repetido). A entrada UC1CLK corresponde a um pino do processador e permite ao usuário fornecer um relógio externo. No caso do MSP430F5529 esse recurso não está disponível. Então, na verdade, temos apenas duas opções relógio: ACLK e SMCLK. Com o Pré-Escalonador, o usuário divide o relógio escolhido, que passa a se chamar BRCLK, de forma a obter SCL na frequência desejada. Note que para o módulo operar como mestre, é preciso fazer UCMST = 1 e somente nesta condição o sinal de relógio UCxSL é disponibilizado na linha SCL.

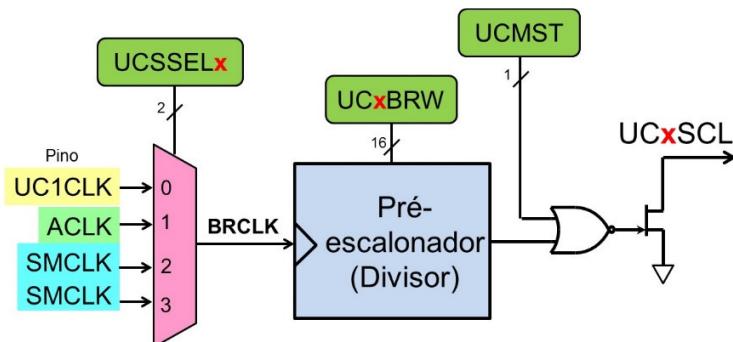


Figura 9.21. Seleção do relógio SCL.

**Exemplo:** Usando SMCLK (1.048.576 Hz) precisamos gerar SCL = 10 kHz. A resposta é fazer UCxBRW = 105. Veja a conta abaixo e note que arredondamos para cima para evitar ultrapassar o valor de 10 kHz.

$$UCxBRW = \frac{1.048.576}{10.000} = 104,8$$

A Figura 9.22 apresenta o restante do módulo USCI\_Bx. Note que a preparação do relógio (UCxSL) já foi feita na figura anterior. Os retângulos verdes indicam opções que o programador pode usar. Nesta figura deve-se notar que cada registrador de deslocamento possui um *buffer* associado. No caso do transmissor, o programador escreve o dado a ser transmitido no UC1TXBUF. A máquina I<sup>2</sup>C transfere o dado para o Registrador de Deslocamento TX e o envia *bit-a-bit* pelo barramento. É de se notar que enquanto se transmite um dado, é possível escrever o novo dado no registrador UC1TXBUF. Algo semelhante acontece na recepção. O Registrador de Deslocamento RX, recebe os *bits* que chegam pelo barramento e quando monta um *byte*, o transfere para o registrador UC1RXBUF. O programador deve ler este dado antes de se completar a recepção de um novo dado.

Existem mais dois registradores. O registrador UCBxI2CSA (*slave address*) que armazena o endereço do escravo quando a USCI opera como mestre, e o registrador UCBxI2COA (*own address*) que é usado para armazenar o próprio endereço do quando a USCI opera como escravo.

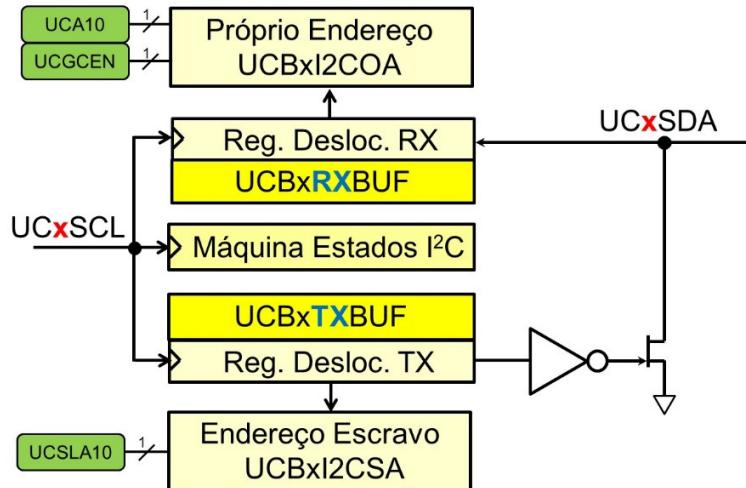


Figura 9.22. Diagrama de blocos do módulo USCI\_Bx.

## 9.5. Inicialização dos Módulos USCI\_Bx

Os parâmetros do módulo USCI\_Bx só podem ser alterados enquanto o *bit* de *reset* da interface (UCSWRST = 1) ativado. Esse *bit* de *reset* é ativado por ocasião do PUC (*Power-Up Clear*), quando se energiza a CPU ou por atuação do programador. Após a

configuração do módulo, o *reset* é removido (`UCSWRST = 0`) e a `UCSBx` está pronta para usar.

A ativação do bit de reset (`UCSWRST = 1`) tem os seguintes resultados:

- A comunicação I<sup>2</sup>C é interrompida e para;
- Linhas SCL e SDA entram em alta impedância;
- Os bits de *status* são zerados (registrador `UCBxI2CSTAT = 0`);
- Os registradores `UCBxIE` e `UCBxIFG` são zerados e
- O demais registradores permanecem inalterados.

É muito importante a sequência abaixo para inicializar ou reconfigurar o módulo USCI:

- 1) Ativar o *reset* (`UCSWRST = 1`);
- 2) Inicializar ou reconfigurar os registradores;
- 3) Configurar as portas;
- 4) Remover o *reset* (`UCSWRST = 0`) e
- 5) Se for o caso, habilitar as interrupções.

## 9.6. Modos de Operação da USCI\_Bx

A seguir faremos um estudo detalhado da interface I<sup>2</sup>C operando nos diversos modos, sempre usando endereçamento em 7 bits. O endereçamento em 10 bits será estudado mais adiante (somente na próxima versão deste livro). São 4 modos, listados a seguir:

- Mestre transmissor;
- Mestre receptor;
- Escravo receptor e
- Escravo transmissor.

As próximas figuras, que ilustram a operação da USCI, seguem as seguintes convenções. Na parte central da figura, com vários retângulos alinhados na horizontal está representada a atividade do barramento I<sup>2</sup>C.

- Os retângulos maiores são as ações da USCI e os retângulos menores as ações do outro dispositivo I<sup>2</sup>C.
- Os retângulos sombreados fazem referência ao mestre e os claros ao escravo.
- Na parte superior estão as atuações do programador e
- Na parte inferior a respostas e informações geradas pela USCI,

### 9.6.1. Modo Mestre

Para configurar o modo I<sup>2</sup>C da USCI é preciso fazer `UCMODEx = 3` e `UCSYNC = 1`. A USCI entra no modo mestre quando se faz o *bit* `UCMST = 1`. Se a USCI mestre for parte de um sistema multi-mestre, é preciso configurar o “endereço próprio” no registrador

UCBxI2COA. O tamanho do endereço é selecionado pelo bit UCA10. O bit UCGCEN permite que se responda à chamada geral. Resumindo tudo na tabela abaixo.

*Tabela 9.9. Possibilidades de configuração da USCI no modo mestre*

Parâmetros	Opções	
UCMODEx	= 3 → modo I <sup>2</sup> C	
UCSYNC	= 1 → habilita modo síncrono	
UCBxI2COA	= endereço próprio para operar no modo multi-mestre	
UCMST	= 0 → modo escravo	= 1 → modo mestre
UCA10	= 0 → endereço em 7 bits	= 1 → endereço em 10 bits
UCGEN	= 0 → sem chamada geral	= 1 → responde a chamada geral

### 9.6.1.1. Modo Mestre Transmissor

Vamos considerar que já foi feita a configuração abaixo:

- UCMODEX = 3 → modo I<sup>2</sup>C
- UCSYNC = 1 → síncrono
- UCMST = 1 → MSP430 é mestre
- UCA10 = 0 → endereço do escravo em 7 bits
- UCGEN = 0 → não responde à chamada geral

Para iniciar a operação, como mostrado na Figura 9.23, é preciso escrever o endereço do escravo no registrador UCBxI2CSA, fazer UCTR = 1, para indicar que é mestre transmissor e ativar UCTXSTT para gerar a condição de START. A USCI verifica a disponibilidade do barramento, gera a condição de START e transmite o endereço do escravo. A flag UCTXIFG vai a 1 quando a condição de START é gerada e o primeiro dado a ser transmitido já pode ser escrito em UCBxTXBUF. Enquanto o dado a ser transmitido não for escrito no UCBxTXBUF, a USCI segura a linha SCL em nível baixo e tudo fica parado. Após a escrita no UCBxTXBUF, a USCI gera mais um pulso (nono pulso) na linha SCL e assim o escravo endereçado pode enviar o ACK ou NACK. Depois disso tudo, o bit UCTXSTT é apagado e o usuário pode verificar a flag UCNACKIFG.

O dado escrito em UCBxTXBUF é então transferido imediatamente para o registrador de deslocamento e enviado pelo barramento. A flag UCTXIFG vai para 1 toda vez que o conteúdo de UCBxTXBUF é copiado para o registrador de deslocamento. Assim, essa flag indica quando se pode escrever um novo dado em UCBxTXBUF. Novamente, se nenhum dado é escrito em UCBxTXBUF, a linha SCL é mantida em nível baixo e tudo fica parado. Após a escrita em UCBxTXBUF, é executado o ciclo de confirmação (ACK) da transmissão do dado anterior.

A ativação do *bit* UCTXSTP gera a condição de STOP logo após a próxima confirmação do escravo (ACK). Se o *bit* UCTXSTP for ativado durante a fase de endereçamento do escravo ou enquanto a USCI espera a escrita de um dado em UCBxTXBUF, uma condição de STOP é gerada, mesmo se nenhum dado foi transferido para o escravo. Para se transmitir um único dado para o escravo, o *bit* UCTXSTP deve ser ativado somente depois de iniciada a transmissão deste dado. A *flag* UCTXIFG serve para indicar se a transmissão do dado foi iniciada.

A ativação do *bit* UCTXSTT serve para gerar uma condição de START REPETIDO. Neste caso o *bit* UCTR (transmissão ou recepção) pode ser alterado e é também alterar o endereço do escravo.

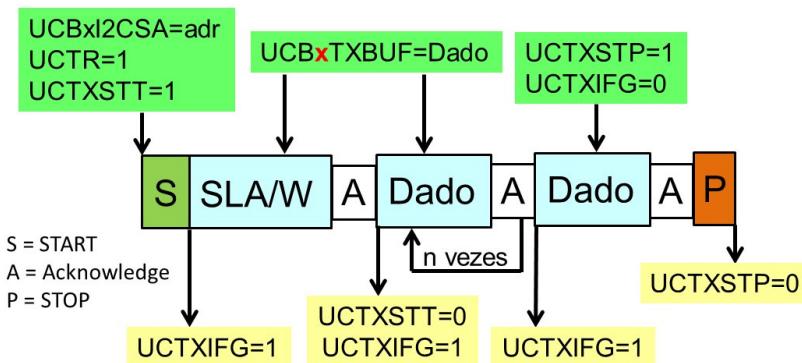


Figura 9.23. Típico diagrama de tempo para o Mestre Transmissor.

Se o escravo não confirma a recepção do endereço ou do dado transmitido, a *flag* UCNACKIFG é ativada, como mostrado na Figura 9.24. O mestre então deve gerar uma condição de STOP ou de START REPETIDO. O dado que porventura estiver no UCBxTXBUF é descartado e junto com o estado do *bit* UCTXSTT. Ou seja, o mestre precisa ativar novamente UCTXSTT.

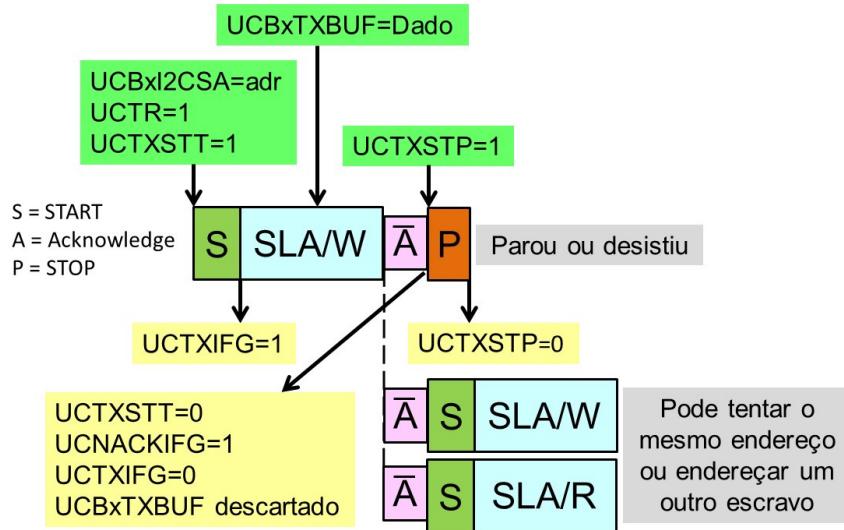


Figura 9.24. Alternativas para quando o escravo não responde ao endereçamento do mestre transmissor.

#### 9.6.1.2. Modo Mestre Receptor

Vamos considerar que já foi feita a configuração abaixo:

- UCMODEX = 3 → modo I<sup>2</sup>C
- UCSYNC = 1 → síncrono
- UCMST = 1 → MSP430 é mestre
- UCA10 = 0 → endereço do escravo em 7 bits
- UCGEN = 0 → não responde à chamada geral

Para iniciar a operação, como mostrado na Figura 9.25, deve-se escrever o endereço do escravo no registrador UCBxI2CSA, fazer UCTR = 0, para indicar que é mestre receptor e ativar UCTXSTT para gerar a condição de START. A USCI verifica a disponibilidade do barramento, gera a condição de START e transmite o endereço do escravo. O bit UCTXSTT é apagado quando o escravo enviar a confirmação do endereçamento (ACK).

Após o escravo confirmar o endereçamento (ACK), a USCI recebe o primeiro dado vindo do escravo. Após a recepção deste dado a flag UCRXIFG é ativada indicando que existe um dado pronto para ser lido e a USCI gera a confirmação (ACK). A leitura do registrador UCBxRXBUF apaga a flag UCRXIFG. Agora, o ciclo de recepção se repete, com UCRXIFG indicando quando tem um dado pronto. O mestre pode finalizá-lo com a geração de uma condição de STOP ou START REPETIDO. Se por algum motivo o registrador UCBxRXBUF demorar para ser lido, o mestre mantém o barramento “preso” após a recepção do último bit, ou seja, não gera a confirmação (ACK).

A ativação do bit UCTXSTP vai gerar uma não confirmação (NAK) após a recepção do corrente dado e em seguida a condição de STOP. Se a USCI estava parada esperando a leitura do registrador UCBxRXBUF, a não confirmação (NAK) e a condição de STOP são geradas imediatamente.

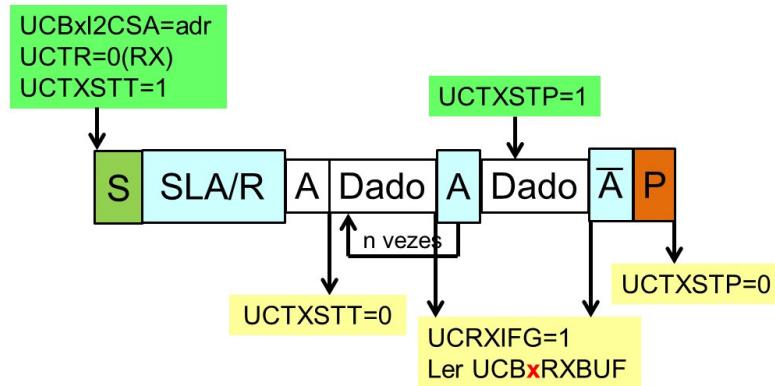
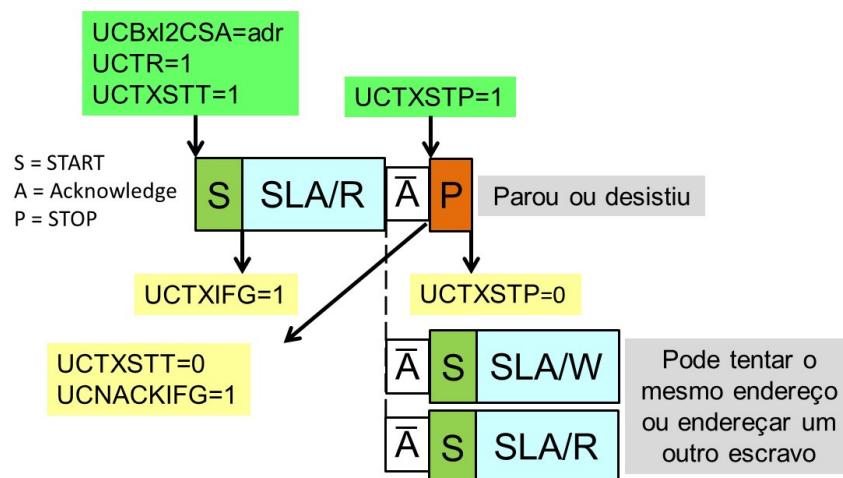


Figura 9.25. Típico diagrama de tempo para o Mestre Receptor.

Se o mestre deseja receber um único byte do escravo, o bit UCTXSTP deve ser ativado enquanto este byte está sendo recebido. Para determinar este momento, é interessante consultar (*polling*) o bit UCTXSTT, pois ele vai a zero quando o escravo confirma o endereçamento (ACK) e inicia a recepção do primeiro dado.

Se o escravo não confirma o endereçamento (NAK), a flag UCNACKIFG é ativada, como mostrado na Figura 9.26. O mestre então deve gerar uma condição de STOP ou de START REPETIDO.



*Figura 9.26. Alternativas para quando o escravo não responde ao endereçamento do mestre transmissor.*

### 9.6.2. Modo Escravo

Para configurar a USCI no modo I<sup>2</sup>C é preciso fazer UCMODEX = 3 e UCSYNC = 1. A USCI entra no modo escravo quando se faz o bit UCMST = 0. Os principais parâmetros estão resumidos na Tabela 9.9, apresentada no tópico anterior. No modo escravo a USCI vai responder ao endereçamento feito por um mestre externo e, em função do endereço recebido, vai transmitir dados ou receber dados.

Inicialmente, a USCI deve ser configurada para o modo recepção (UCTR = 0) para poder receber o endereço enviado pelo mestre. Depois, configurando o *bit* UCTR a USCI deve transmitir ou receber dados em função do modo como foi endereçado. O endereço da USCI escrava deve ser programado no registrador UCBxI2COA. Por enquanto, vamos considerar o endereçamento em 7 *bits* (UCA10 = 0).

Quando uma condição de START é detectada, a USCI recebe o endereço enviado pelo mestre e o compara com o seu próprio endereço que foi previamente armazenado no registrador UCBxI2COA. Se eles forem idênticos, a USCI faz o *bit* UCSTTIFG = 1.

#### 9.6.2.1. Modo Escravo Transmissor

Vamos considerar que foi feita a configuração abaixo:

- UCMODEX = 3 → modo I<sup>2</sup>C
- UCSYNC = 1 → síncrono
- UCMST = 0 → MSP430 é escravo
- UCA10 = 0 → endereço do escravo em 7 *bits*
- UCGEN = 0 → não responde à chamada geral

A USCI entra no modo escravo transmissor quando o endereço enviado pelo mestre coincide com seu próprio endereço, sendo que o último *bit* está em 1 (leitura). O escravo transmissor envia dados serialmente seguindo o relógio SCL gerado pelo mestre. O escravo transmissor não pode gerar o relógio, mas ele pode segurar a linha SCL em nível baixo enquanto aguarda um novo dado a ser transmitido. Com isso o barramento fica “parado”.

Quando a USCI é endereçada como escravo transmissor, os *bits* UCSTTIFG, UCTXIFG e UCTR são ativados automaticamente, como mostrado na Figura 9.27. Eles têm os seguintes significados:

- UCSTTIFG = 1 → START seguido pelo endereço de escravo da USCI;
- UCTR = 1 → endereçamento como escravo transmissor e

- $\text{UCTXIFG} = 1 \rightarrow$  o registrador  $\text{UCBxTXBUF}$  espera pelo dado a ser transmitido.

A partir deste momento, a linha SCL é mantida em nível baixo até o que o dado a ser enviado seja escrito no  $\text{UCBxTXBUF}$ . Quando isso acontece, o escravo gera a confirmação do endereçamento (ACK), apaga o bit  $\text{UCSTTIFG}$  e transmite o dado. Assim que finaliza a transmissão do dado, a flag  $\text{UCTXIFG}$  é novamente ativada, indicando que se espera por um novo dado. Após a confirmação (ACK) pelo mestre receptor, o novo dado escrito em  $\text{UCBxTXBUF}$  é enviado. Se o  $\text{UCBxTXBUF}$  estiver vazio, a linha SCL é mantida em nível baixo e o barramento fica parado.

Se o mestre enviar uma não confirmação (NAK) seguida por uma condição de STOP, a flag  $\text{UCSTPIFG}$  é ativada. Se a não confirmação (NAK) for seguida por uma condição de START REPETIDO, a USCI retorna para a fase de recepção de endereço.

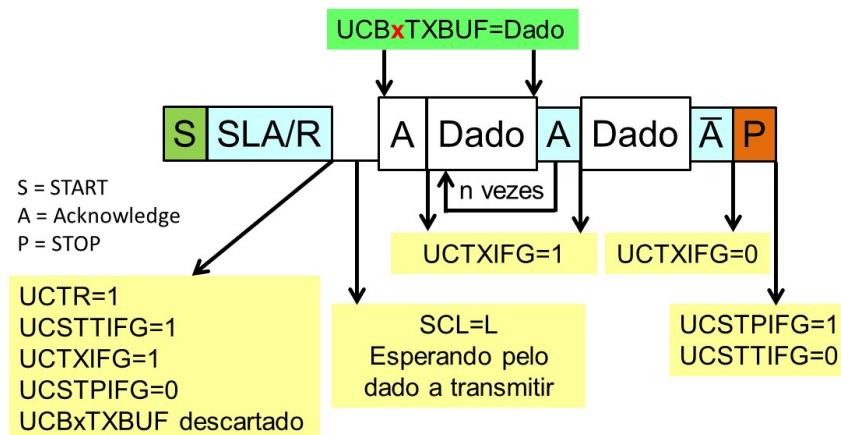


Figura 9.27. Típico diagrama de tempo para a USCI operar como Escravo Transmissor.

### 9.6.2.1. Modo Escravo Receptor

Vamos considerar que foi feita a configuração abaixo:

- $\text{UCMODEX} = 3 \rightarrow$  modo I<sup>2</sup>C
- $\text{UCSYNC} = 1 \rightarrow$  síncrono
- $\text{UCMST} = 0 \rightarrow$  MSP430 é escravo
- $\text{UCA10} = 0 \rightarrow$  endereço do escravo em 7 bits
- $\text{UCGEN} = 0 \rightarrow$  não responde à chamada geral

A USCI entra no modo escravo receptor quando o endereço enviado pelo mestre coincide com seu próprio endereço, sendo que o último bit está em 0 (escrita). O escravo receptor recebe os dados enviados pelo mestre, seguindo o relógio SCL. O escravo receptor não pode gerar o relógio, mas ele pode segurar a linha SCL em nível baixo enquanto aguarda uma intervenção da CPU. Com isso o barramento fica “parado”.

Quando a USCI é endereçada como escravo receptor, o *bit* UCSTTIFG é ativado e o *bit* UCTR é apagado, como mostrado na Figura 9.28. Eles têm os seguintes significados:

- UCSTTIFG = 1 → START seguido pelo endereço de escravo da USCI e
- UCTR = 0 → endereçamento como escravo receptor.

Após a recepção do primeiro dado, a USCI gera automaticamente uma confirmação (ACK) e ativa a *flag* UCRXIFG para indicar que existe um dado pronto em UCBxRXBUF. Em seguida, pode iniciar a recepção de um novo dado. Ao término de uma recepção, se o dado recebido anteriormente ainda não foi lido, a USCI mantém a linha SCL em nível baixo e só gera a confirmação (ACK) após a CPU ler este dado.

O *bit* UCTXNACK é usado para gerar uma não-confirmação (NAK) ao final da recepção do dado atual. A não-confirmação (NAK) é gerada mesmo quando a linha SCL estava sendo mantida em nível baixo esperando a CPU ler UCBxRXBUF. Neste caso, o último dado recebido sobrescreve o que estava em UCBxRXBUF, ou seja, o dado que estava neste registrador é perdido. Uma boa forma de evitar a perda de um dado é ler primeiro o UCBxRXBUF para só depois ativar o bit UCTXNACK.

Quando o mestre gera uma condição de STOP, a *flag* UCSTPIFG é ativada. Se o mestre gera uma condição de START REPETIDO, a USCI retorna para a fase de recepção de endereço.

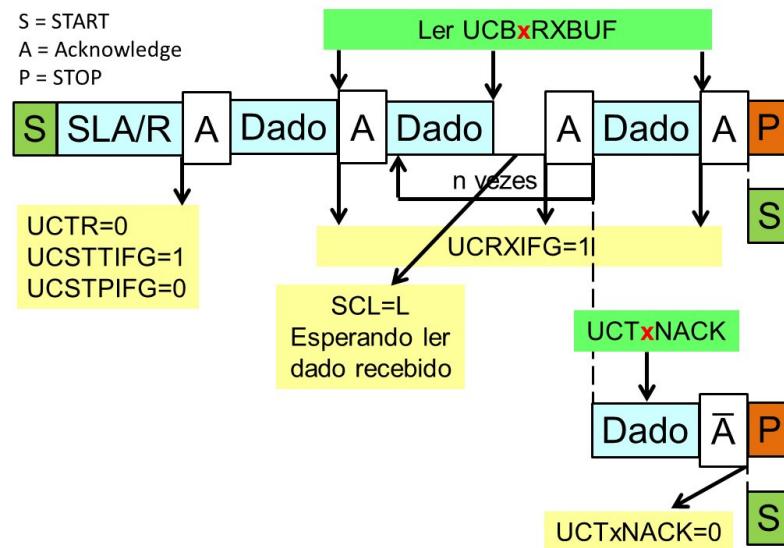


Figura 9.28. Típico diagrama de tempo para a USCI operar como Escravo Receptor.

## 9.7. Interrupções da USCI\_Bx no Modo I<sup>2</sup>C

Cada USCI tem um único vetor para atender às 6 interrupções que estão disponíveis quando opera no Modo I<sup>2</sup>C. Temos uma interrupção para a transmissão, uma para a recepção e 3 para as mudanças de estado, como listado na Tabela 9.1. Note que cada *flag* de interrupção possui uma habilitação em separado. Para que qualquer uma das interrupções possa ocorrer, a *flag* GIE precisa estar em 1.

*Tabela 9.1 - Lista das interrupções disponíveis para operação da USCI no modo I<sup>2</sup>C.*

<b>Flag</b>	<b>Habilitação</b>	<b>Evento</b>
UCRXIFG	UCRXIE	Um dado acabou de ser recebido e está pronto para ser lido no registrador UCBxRXBUF. A leitura deste registrador apaga a <i>flag</i> .
UCTXIFG	UCTXIE	O dado acabou de ser transmitido e o registrador UCBxTXBUF está pronto para aceitar um novo dado. A escrita neste registrador apaga a <i>flag</i> .
UCSTTIFG	USCTTIE	Enquanto a USCI estava no Modo Escravo, uma condição de START com o próprio endereço foi detectada. A <i>flag</i> é apagada quando uma condição de STOP é recebida.
UCSTPIFG	UCSTPIE	Enquanto a USCI estava no Modo Escravo, uma condição de STOP foi detectada. A <i>flag</i> é apagada quando uma condição de START é recebida.
UCNACKIFG	UCNACKIE	A USCI esperava por uma confirmação que não chegou. A <i>flag</i> é apagada quando uma condição de START é recebida.
UCALIFG	UCALIE	Aconteceu a perda de arbitragem. A <i>flag</i> UCMST é zerada e a USCI entra no Modo Escravo.

A USCI\_B0 usa a posição 55 (USCI\_B0\_VECTOR) da Tabela de Interrupções e a USCI\_B1 a posição 45 (USCI\_B1\_VECTOR). O registrador de vetores de interrupção da USCI (UCBxIV) precisa ser consultado para que se saiba qual das 6 *flags* provocou a interrupção. A Tabela 9.2 apresenta uma lista com os possíveis resultados da leitura deste registrador.

*Tabela 9.2. Possíveis resultados da leitura do registrador UCBxIV.*

<b>Prioridade</b>	<b>Valor</b>	<b>Flag</b>	<b>Descrição</b>
	0x0	Nenhum	Nenhuma interrupção pendente
Maior	0x2	UCALIFG	Perda de arbitragem

	0x4	UCNACKIFG	Não-confirmação (NAK)
	0x6	UCSTTIFG	Detectada condição de START
	0x8	UCSTPIFG	Detectada condição de STOP
	0xA	UCRXIFG	Dado recebido
Menor	0xC	UCTXIFG	Buffer de transmissão vazio

Apresentamos a seguir uma sugestão para a função (ISR) que vai atender a essas interrupções. No caso, tomamos como exemplo a USCI\_B0. É claro que o usuário deverá escrever cada uma das funções para responder aos eventos da porta I<sup>2</sup>C.

```
#pragma vector = USCI_B0_VECTOR //Vetor = 55
__interrupt void isr_usci_b0(void){
    int n;
    n = __even_in_range(UCB0IV, 0xC);
    switch(n) {
        case 0x0: break;
        case 0x2: i2c_arb_lost(); break;
        case 0x4: i2c_nack(); break;
        case 0x6: i2c_start(); break;
        case 0x8: i2c_stop(); break;
        case 0xA: i2c_rx(); break;
        case 0xC: i2c_tx(); break;
    }
}
```

## 9.8. Registradores da USCI\_Bx no Modo I<sup>2</sup>C

Nesta seção são descritos os registradores da USCI\_Bx quando opera no modo I<sup>2</sup>C. Alguns podem ser acessados como um registrador de 16 bits ou em duas porções de 8 bits, como mostrado na Tabela 9.3. Outros possuem acesso exclusivo em 8 ou 16 bits. Para o programador em C, isso não faz grande diferença, já que o compilador cuida dos detalhes. Na descrição particular de cada um, para facilitar a ocupação do espaço de cada página, daremos preferência ao acesso usando as porções de 8 bits.

Tabela 9.3. Registradores da USCI\_Bx quando opera no modo I<sup>2</sup>C.

16 bits	8 bits	Acesso	Reset	Ordem
UCBxCTLW0	UCBxCTL1	R/W	0x1	LSB
	UCBxCTL0	R/W	0x1	MSB

UCBxBRW	UCBxBR0	R/W	0	LSB
	UCBxBR1	R/W	0	MSB
-	UCBxSTAT	R/W	0	-
-	UCBxRXBUF	R/W	0	-
-	UCBxTXBUF	R/W	0	-
UCBxI2COA	-	R/W	0	-
UCBxI2CSA	-	R/W	0	-
UCBxICTL	UCBxIE	R/W	0	LSB
	UCBxIFG	R/W	0	MSB
UCBxIV	-	R	0	-

### 9.8.1. UCBxCTL0 – Registrador 0 de Controle (USCI\_Bx Control Register 0)

Este registrador controla o acesso ao modo I<sup>2</sup>C, o tamanho dos campos de endereços e a configuração da USCI em modo Mestre ou Escravo.

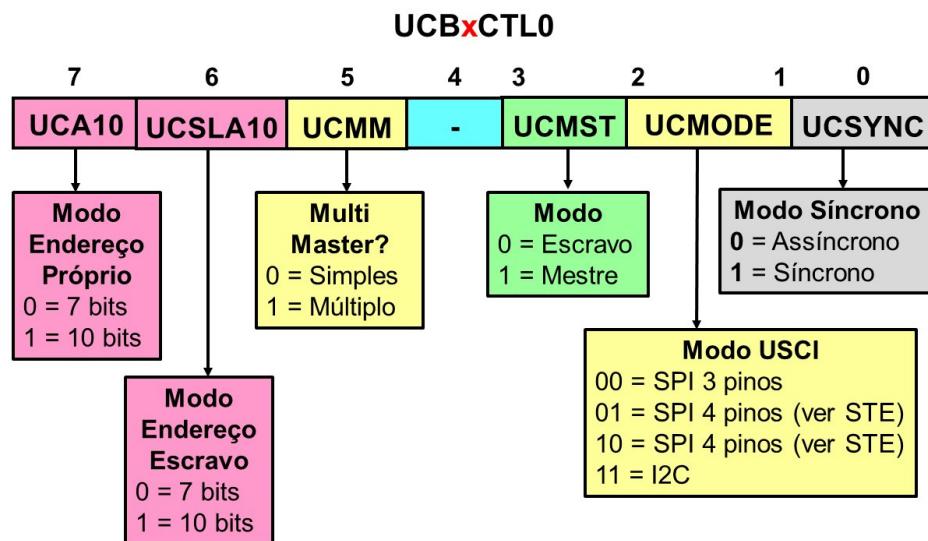


Figura 9.29. Descrição dos bits do registrador UCBxCTL0.

#### (R/W) Bit 7: UCA10 – Tamanho do campo de endereço próprio (Own addressing mode select)

O usuário seleciona o tamanho do campo de endereço da USCI\_Bx quando opera no modo escravo.

UCA10 = 0 → Campo de endereço com 7 bits e

UCA10 = 1 → Campo de endereço com 10 bits.

**(R/W) Bit 6: UCSLA10 – Tamanho do campo de endereço do escravo  
(Slave addressing mode select)**

O usuário seleciona o tamanho do campo de endereço do escravo a ser acessado, quando a USCI\_Bx opera no modo mestre.

UCSLA10 = 0 → Campo de endereço com 7 bits e  
UCSLA10 = 1 → Campo de endereço com 10 bits.

**(R/W) Bit 5: UCMM – Seleção do ambiente multi-mestre  
(Multi-master environment select)**

O usuário define se a USCI\_Bx está ou não operando num ambiente com diversos mestres.

UCMM = 0 → A USCI é o único mestre presente e por isso a unidade que compara endereços é desabilitada.

UCMM = 1 → Vários mestres presentes e a USCI precisa atender ao endereçamento.

**(R) Bit 4: Reservado**

Bit reservado, sua leitura sempre resulta em 0.

**(R/W) Bit 3: UCMST – Seleção do modo mestre  
(Master mode select)**

O programador indica que USCI é a mestre do barramento. Num ambiente multi-mestre, se a USCI perde a arbitragem, este bit é apagado e a USCI passa a operar com escrava.

UCMST = 0 → Modo escravo.

UCMST = 1 → Modo mestre.

**(R/W) Bits 2,1: UCMODE – Modo da USCI  
(USCI mode)**

O programador seleciona se vai operar nos modos SPI ou I<sup>2</sup>C.

UCMODEx = 0x0, 0x1 e 0x2 → Modo SPI

UCMODEx = 0x3 → Modo I<sup>2</sup>C

**(R) Bit 0: UCSYNC – Habilitar modo síncrono  
(Synchronous mode enable)**

Como a USCI\_Bx sempre opera com protocolos síncronos (I<sup>2</sup>C ou SPI), este bit está sempre em 1 e só pode ser lido.

UCSYNC = 0 → Modo assíncrono (não disponível para a USCI\_Bx)

UCSYNC = 1 → Modo síncrono

**9.8.2. UCBxCTL1 – Registrador 1 de Controle  
(USCI\_Bx Control Register 1)**

Este registrador permite a seleção do relógio para gerar a frequência da linha SCL, além controlar as condições de START, STOP e NACK. Muito importante é o *bit* de *reset* da USCI que aqui está disponível.

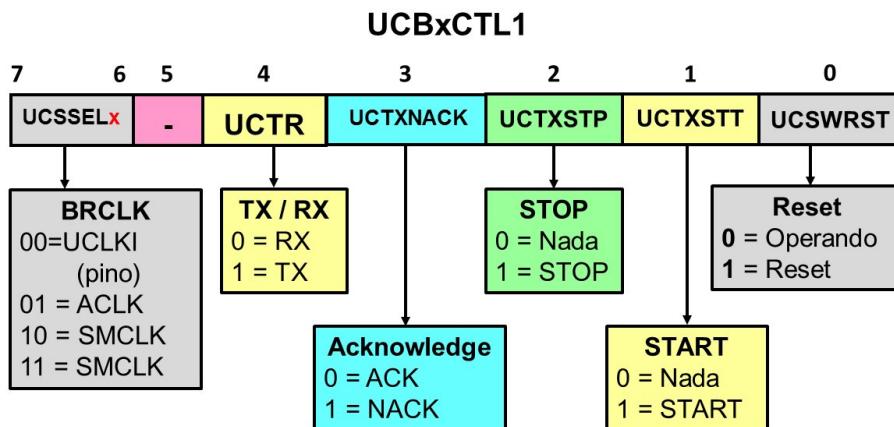


Figura 9.30. Descrição dos bits do registrador UCBxCTL1.

**(R/W) Bits 7,6: UCSSEL<sub>x</sub> – Seleção do relógio para a USCI**

**(USCI clock source select)**

Com esses dois *bits* o programador seleciona o relógio que será usado para gerar a frequência da linha SCL.

UCMODE<sub>x</sub> = 0x0 → UCLKI, relógio por um pino da CPU. Não disponível na F5529.

UCMODE<sub>x</sub> = 0x1 → ACLK

UCMODE<sub>x</sub> = 0x2 → SMCLK

UCMODE<sub>x</sub> = 0x3 → SMCLK

**(R) Bit 5: Reservado**

Bit reservado, sua leitura sempre resulta em 0.

**(R/W) Bit 4: UCTR – Modo transmissor ou receptor**

**(Transmitter or receiver)**

O programador seleciona se a USCI vai transmitir dados ou receber dados, tanto no modo mestre como no modo escravo.

UCTR = 0 → Receptor.

UCTR = 1 → Transmissor.

**(R/W) Bit 3: UCTXNACK – Transmitir um NACK**

**(Transmit a NACK)**

O programador indica que quer gerar um NACK ao final da operação que está em curso.

Este *bit* é apagado após o envio do NACK.

UCTXNACK = 0 → Confirmação normal (ACK).

UCTXNACK = 1 → Não confirmar, gerar NACK.

**(R/W) Bit 2: UCTXSTP – Transmitir condição de STOP**

**(Transmit STOP condition in master mode)**

Quando no modo mestre, o programador indica que quer gerar uma condição de STOP. Este *bit* é ignorado se a USCI estiver no modo escravo. Quando no modo Mestre Receptor, a condição de STOP é precedida pela geração de um NACK. Este *bit* é automaticamente zerado após a condição de STOP ser gerada.

UCTXSTP = 0 → Não gerar a condição de STOP.

UCTXSTP = 1 → Gerar a condição de STOP.

**(R/W) Bit 1: UCTXSTT – Transmitir condição de START**

**(Transmit START condition in master mode)**

Quando no modo mestre, o programador indica que quer gerar uma condição de START. Este *bit* é ignorado se a USCI estiver no modo escravo. Quando no modo Mestre Receptor, a condição de START REPETIDO é precedida pela geração de um NACK. Este *bit* é automaticamente zerado após a condição de START ser gerada e o endereço do escravo transmitido.

UCTXSTART = 0 → Não gerar a condição de START.

UCTXSTART = 1 → Gerar a condição de START.

**(R/W) Bit 0: UCSWRST – Reset habilitado por software**

**(Software reset enable)**

O programador faz o *reset* da USCI. Vale lembrar que a configuração da USCI deve acontecer com este *bit* ativado.

UCSWRST = 0 → *Reset* desabilitado, USCI liberada para operar.

UCSWRST = 1 → *Reset* habilitado.

**9.8.3. UCBxBRW – Registrador de controle do *baud rate***

**(USCI\_Bx Baud Rate Control Register)**

Com este registrador o programador seleciona a frequência da linha SCL, ou seja, ele especifica o *baud rate* de sua transmissão. Como pode ser visto na figura, ele pode ser acessado como um registrador de 16 *bits*, ou como 2 registradores de 8 *bits*. A relação entre eles está representada na equação abaixo. Para a seleção da frequência da linha SCL, consultar a figura 9.21.

$$UCBxBRW = 256 \times UCBxBR1 + UCBxBR0$$

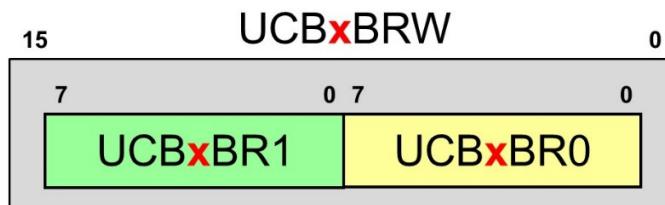


Figura 9.31. Descrição do registrador UCBxBRW – Controle do Baud Rate.

#### 9.8.4. UCBxSTAT – Registrador de estado (USCI\_Bx Status Register)

Com este registrador o programador acompanha alguns acontecimentos importantes do barramento I<sup>2</sup>C.

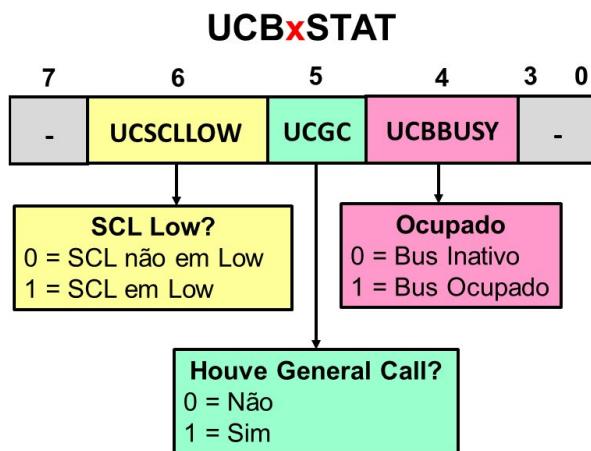


Figura 9.32. Descrição dos bits do registrador UCBxSTAT.

##### (R) Bit 7: Reservado

Bit reservado, sua leitura sempre resulta em 0.

##### (R) Bit 6: UCSCLLOW – USCL em nível baixo (SCL low)

Indica se a linha de relógio SCL está sendo mantida em nível baixo. Seu uso faz sentido quando a USCI está no modo mestre.

UCSCLLOW = 0 → Linha SCL não está sendo mantida em nível baixo.

UCSCLLOW = 1 → Linha SCL está sendo mantida em nível baixo.

##### (R/W) Bit 5: UCGC – Recebida uma chamada geral

**(General call address received)**

Indica que chegou uma chamada geral. Esta *flag* é gerada quando uma condição de START é recebida.

UCGC = 0 → Nenhum endereço de chamada geral foi recebido.

UCGC = 1 → Foi recebido endereço de chamada geral.

**(R) Bit 4: UCBBUSY – Barramento ocupado****(Bus busy)**

Indica se o barramento I<sup>2</sup>C está sendo usado.

UCBBUSY = 0 → Barramento inativo.

UCBBUSY = 1 → Barramento ocupado.

**(R) Bits 3, 2, 1, 0: Reservados**

Bits reservados, sua leitura sempre resulta em 0.

### 9.8.5. UCBxRXBUF – Registrador de Recepção (USCI\_Bx Receive Data)

Este registrador de 8 *bits* é só de leitura e contém o último dado recebido pelo barramento I<sup>2</sup>C. A leitura deste registrador zera a *flag* UCRXIFG.

**UCBxRXBUF**



Figura 9.33. Descrição do registrador UCBxRXBUF.

### 9.8.6. UCBxTXBUF – Registrador de Transmissão (USCI\_Bx Transmit Data)

Este registrador de 8 *bits* é de leitura ou escrita e contém o dado a ser transmitido pelo barramento I<sup>2</sup>C. A escrita neste registrador zera a *flag* UCTXIFG.

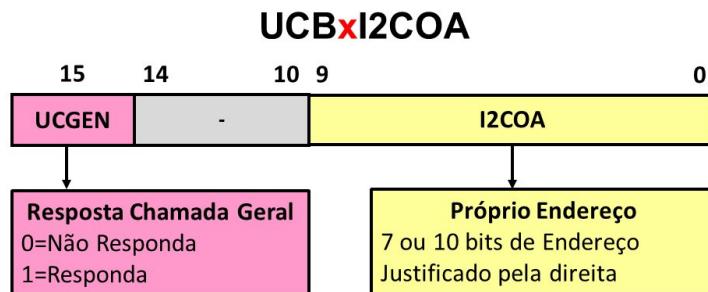
**UCBxTXBUF**



*Figura 9.34. Descrição do registrador UCBxTXBUF.*

### 9.8.7. UCBxI2COA – Registrador do Endereço Próprio da USCI\_Bx (*USCI\_Bx I2C Own Address Register*)

Este registrador de 16 bits contém o endereço da USCI\_Bx e também permite habilitar a resposta à chamada geral. Quando em responde à chamada geral, a USCI opera no modo escravo receptor.



*Figura 9.35. Descrição do registrador UCBxI2COA.*

#### (R/W) Bit 15: UCGCEN – Habilitação da resposta à Chamada Geral (*General Call response enable*)

Indica se a USCI\_Bx deve ou não responder à chamada geral.

UCGCEN = 0 → Não responda à chamada geral.

UCGCEN = 1 → Responda à chamada geral.

#### (R) Bits 14:10: Reservados

Bits reservados, sua leitura sempre resulta em 0.

#### (R/W) Bits 9:0: Endereço próprio da USCI\_Bx (*I2C own address*)

Estes bits especificam o endereço da USCI\_Bx quando ela opera no modo escravo. O endereço é alinhado pela direita. Se for selecionado o endereçamento de 7 bits, os bits de 9-7 são ignorados.

### 9.8.8. UCBxI2CSA – Registrador do Endereço do Escravo (*USCI\_Bx I2C Slave Address Register*)

Este registrador de 16 bits contém o endereço do escravo a ser acessado quando a USCI\_Bx opera no modo mestre.

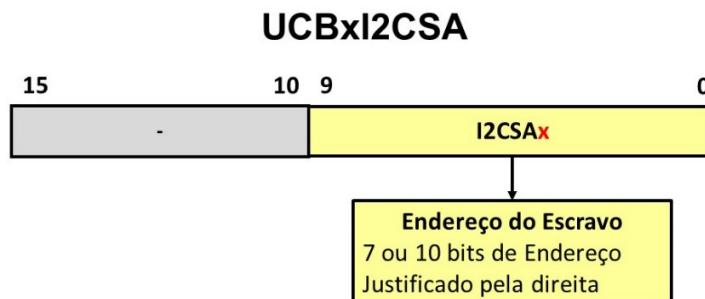


Figura 9.36. Descrição do registrador UCBxI2CSA.

#### (R) Bits 15:10: Reservados

Bits reservados, sua leitura sempre resulta em 0.

#### (R/W) Bits 9:0: Endereço do escravo

##### (I2C slave address)

Estes bits especificam o endereço do escravo a ser acessado. Somente é usado quando a USCI\_Bx está no modo mestre. O endereço é alinhado pela direita. Se for selecionado o endereçamento de 7 bits, os bits de 9-7 são ignorados.

### 9.8.9. UCBxIE – Registrador de Habilitação de Interrupção da USCI\_Bx (USCI\_Bx I2C Interrupt Enable Register)

Este registrador de 8 bits permite habilitar a interrupção para diversos eventos do barramento I<sup>2</sup>C.

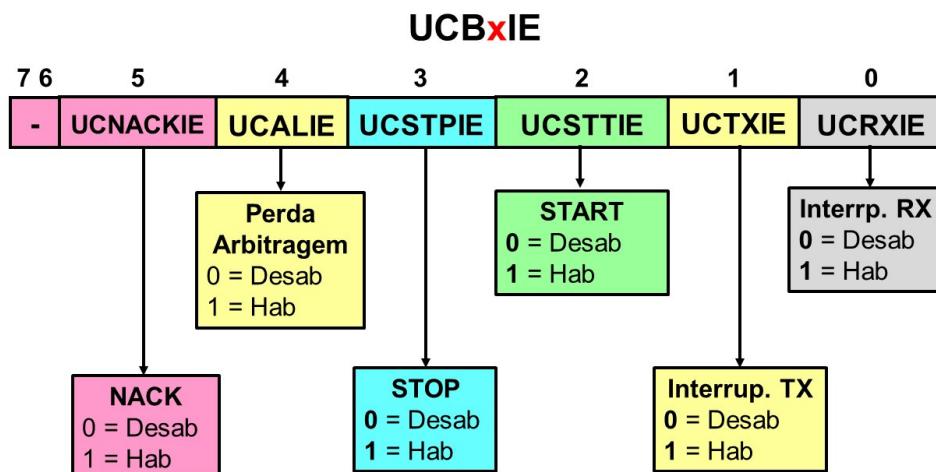


Figura 9.37. Descrição do registrador UCBxIE.

**(R) Bits 7, 6: Reservados**

Bits reservados, sua leitura sempre resulta em 0.

**(R/W) Bit 5: UCNAKIE – Habilita interrupção por uma não confirmação (NAK)  
(Not-acknowledge interrupt enable)**

UCNACKIE = 0 → Interrupção desabilitada.

UCNACKIE = 1 → Interrupção habilitada.

**(R/W) Bit 4: UCALIE – Habilita da interrupção por perda de arbitragem  
(Arbitration lost interrupt enable)**

UCALIE = 0 → Interrupção desabilitada.

UCALIE = 1 → Interrupção habilitada.

**(R/W) Bit 3: UCSTPIE – Habilita da interrupção por receber condição de STOP  
(STOP condition interrupt enable)**

UCSTPIE = 0 → Interrupção desabilitada.

UCSTPIE = 1 → Interrupção habilitada.

**(R/W) Bit 2: UCSTTIE – Habilita da interrupção por receber condição de START  
(START condition interrupt enable)**

UCSTTIE = 0 → Interrupção desabilitada.

UCSTTIE = 1 → Interrupção habilitada.

**(R/W) Bit 1: UCTXIE – Habilita da interrupção por transmissão completada  
(Transmit interrupt enable)**

UCTXIE = 0 → Interrupção desabilitada.

UCTXIE = 1 → Interrupção habilitada.

**(R/W) Bit 0: UCRXIE – Habilita da interrupção por recepção completada  
(Receive interrupt enable)**

UCRXIE = 0 → Interrupção desabilitada.

UCRXIE = 1 → Interrupção habilitada.

**9.8.10. UCBxIFG – Registrador de Flags de Interrupção da USCI\_Bx  
(USCI\_Bx I<sup>2</sup>C Interrupt Flag Register)**

Este registrador de 8 bits indica eventos do barramento I<sup>2</sup>C e são ponto de partida de todas as interrupções destinadas à USCI\_Bx.

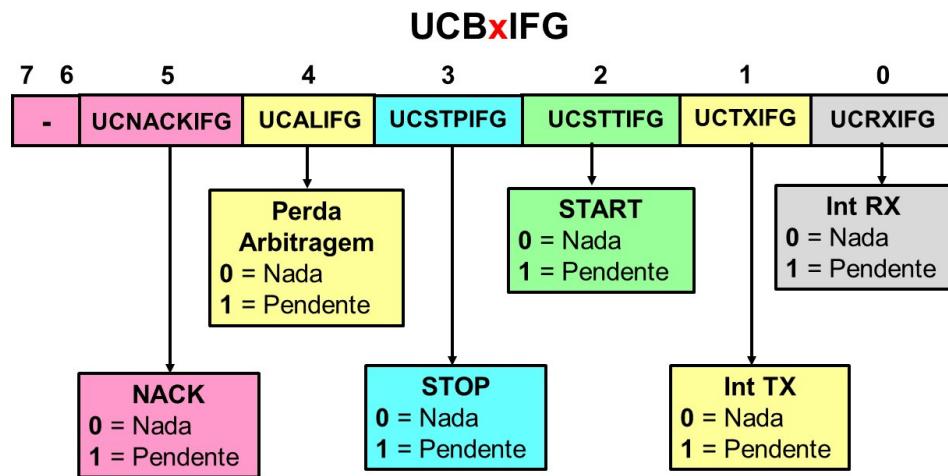


Figura 9.38. Descrição do registrador UCBxIFG.

#### (R) Bits 7, 6: Reservados

Bits reservados, sua leitura sempre resulta em 0.

#### (R/W) Bit 5: UCNACKIFG – Interrupção por recepção de uma não confirmação (NAK) (Not-acknowledge received interrupt flag)

Esta flag é automaticamente apagada quando uma condição de START é recebida.

UCNACKIFG = 0 → Não há interrupção pendente.

UCNACKIFG = 1 → Interrupção pendente.

#### (R/W) Bit 4: UCALIFG – Interrupção por perda de arbitragem (Arbitration lost interrupt flag)

UCALIFG = 0 → Não há interrupção pendente.

UCALIFG = 1 → Interrupção pendente.

#### (R/W) Bit 3: UCSTPIFG – Interrupção por receber condição de STOP (STOP condition interrupt flag)

Esta flag é automaticamente apagada quando uma condição de START é recebida.

UCSTPIFG = 0 → Não há interrupção pendente.

UCSTPIFG = 1 → Interrupção pendente.

#### (R/W) Bit 2: UCSTTIFG – Interrupção por receber condição de START (START condition interrupt flag)

Esta flag é automaticamente apagada quando uma condição de START é recebida.

UCSTTIFG = 0 → Não há interrupção pendente.

UCSTTIFG = 1 → Interrupção pendente.

#### (R/W) Bit 1: UCTXIFG – Interrupção por transmissão completada (USCI Transmit interrupt flag)

Esta *flag* é ativada quando o registrador UCBxTXBUF está vazio. Ela é apagada quando se escreve em UCBxTXBUF.

UCTXIE = 0 → Não há interrupção pendente.

UCTXIE = 1 → Interrupção pendente.

#### **(R/W) Bit 0: UCRXIFG – Interrupção por recepção completada**

##### **(USCI Receive interrupt flag)**

Esta *flag* é ativada quando a USCI disponibiliza um dado no registrador UCBxRXBUF. Ela é apagada quando se lê o UCBxRXBUF.

UCRXIFG = 0 → Não há interrupção pendente.

UCRXIFG = 1 → Interrupção pendente.

### **9.8.11. UCBxIV – Registrador de Vetor de Interrupção da USCI\_Bx (USCI\_Bx I2C Interrupt Vector Register)**

Este é um registrado de 16 *bits* e indica o evento mais prioritário dentre os 6 possíveis eventos que podem provocar interrupção. A cada leitura, este registrador retorna um número correspondente à interrupção de maior prioridade dentre as pendentes. Em seguida apaga a *flag* desta interrupção. Na próxima leitura é retornado um número correspondente à segunda mais prioritária dentre as pendentes e assim por diante.



Figura 9.39. Descrição do registrador UCBxIV.

Tabela 9.3. Descrição dos valores retornados pela leitura do registrador UCBxIV

Prioridade	Valor	Flag	Significado
	0x0	-	Nenhuma interrupção pendente
Maior	0x2	UCALIFG	Perda de arbitragem
	0x4	UCNACKIFG	Recepção de uma não-confirmação (NAK)
	0x6	UCSTTIFG	Recepção de uma condição de START
	0x8	USCTPIFG	Recepção de uma condição de STOP
	0xA	UCRXIFG	Dado recebido
Menor	0xC	UCTXIFG	Buffer de transmissão vazio

## 9.9. Exercícios Resolvidos

Para se propor exercícios com o I<sup>2</sup>C, evidentemente, precisa-se de um dispositivo I<sup>2</sup>C. Felizmente o MSP430 F5529 oferece duas USCI que podem operar no protocolo I<sup>2</sup>C. Então, por facilidade no início, vamos usar a USCI\_B0 operando como mestre e a USCI\_B1 operando como escravo.

**ER 9.1.** Este exercício tem o objetivo de criar um escravo para ser usado pelos próximos exercícios. Ele tem uma certa complexidade, e pode ser estudado mais tarde. Se o leitor quiser, vá direto para o próximo exercício resolvido.

Pedido: Crie os arquivos para programar a USCI\_B1 para operar no modo escravo, tanto transmissor quanto receptor, usando o endereço 42. Quando for endereçada como escravo transmissor, ela envia em sequência os bytes 0, 1, 2, ..., 255, 0, 1, ... e assim por diante. Quando endereçada no modo escravo receptor, ela armazena no vetor b1\_vet os últimos bytes recebidos. Para não interferir com os programas a serem testados, todo o trabalho deve ser realizado com interrupções.

Curiosidade: Seria 42 a resposta para tudo?

Dica: 42 é o segundo maior número esfênico.

### Solução:

Aqui serão preparadas as rotinas fazer a USCI\_B1 operar como escravo no endereço 42 e assim permitir o ensaio da USCI\_B0 operando como mestre. Elas não devem ser rodadas de isoladas. Foram preparadas para o uso junto com outros programas. Para facilitar o uso dessas rotinas, serão criados dois arquivos:

- USCI\_B1.c → com as três funções abaixo e
  - USCI\_B1\_config: que configura a USCI como escravo com endereço 42 e habilita interrupções;
  - USCI\_B1\_rx: ISR que recebe um dado e armazena num vetor e
  - USCI\_B1\_tx: ISR que envia um dado (contador).
- USCI\_B1.h → header com as declarações necessárias.

Na tabela abaixo está a configuração dos registradores quando a USCI opera como escravo no endereço 42. Antes de iniciar a configuração, é necessário ativar o *reset* com o bit UCSWRST = 1. Note que o bit UCMST foi mantido em zero, o que caracteriza configuração como escravo. Como essa unidade vai trabalhar com escravo, não é necessário configurar o relógio. Para terminar a configuração da USCI\_B1, as interrupções de transmissão (UCTXIE) e de recepção (UCRXIE) foram habilitadas (após a unidade sair do *reset*, como mostrado na listagem).

Os pinos usados por essa unidade USCI\_B1 (P4.2 e P4.1) precisam ser configurados, de acordo com o que está na Tabela 9.5. É importante comentar que os *bits* de I/O da porta P4 podem ser mapeados para diferentes recursos. Porém, processo de inicialização da LaunchPad faz o mapeamento desses dois pinos para a porta USCI\_B1.

*Tabela 9.4. Configuração da USCI\_B1 para operar como escravo no endereço 42*

	7	6	5	4	3	2	1	0
<b>UCB1CTL0</b>	UCA10	UCSLA10	UCMM	-	UCMST	<b>UCMODE</b>		UCSYNC
	0	0	0	0	0	3		1
<b>UCB1CTL1</b>	UCSSEL		-	UCTR	UCTXNACK	UCTXSTP	UCTXSTT	UCWRST
	0		0	0	<u>0</u>	0	0	1/0
<b>UCB1IE</b>	-	-	UCNACKIE	UCALIE	UCSTPIE	UCSTTIE	UCTXIE	UCRXIE
	0	0	0	0	0	0	1	1
<b>UCB1I2COA</b>	42							

*Tabela 9.5. Configuração do GPIO usado pela porta USCI\_B1*

	7	6	5	4	3	2	1	0
<b>P4OUT</b>	<i>BIT7</i>	<i>BIT6</i>	<i>BIT5</i>	<i>BIT4</i>	<i>BIT3</i>	<i>BIT2</i>	<i>BIT1</i>	<i>BIT0</i>
	-	-	-	-	-	1	1	-
<b>P4REN</b>	<i>BIT7</i>	<i>BIT6</i>	<i>BIT5</i>	<i>BIT4</i>	<i>BIT3</i>	<i>BIT2</i>	<i>BIT1</i>	<i>BIT0</i>
	-	-	-	-	-	1	1	-
<b>P4SEL</b>	<i>BIT7</i>	<i>BIT6</i>	<i>BIT5</i>	<i>BIT4</i>	<i>BIT3</i>	<i>BIT2</i>	<i>BIT1</i>	<i>BIT0</i>
	-	-	-	-	-	1	1	-

No arquivo cabeçalho (USCI\_B1.h), cuja listagem está logo a seguir, há pouca coisa a ser comentada. Logo no início é criada uma constante (MAX\_B1) para definir o tamanho do vetor que vai receber os dados. Depois vem os protótipos das funções e a declaração das variáveis globais. Veja que todas são indicadas como voláteis.

#### *Listagem do arquivo cabeçalho para o ER 9.1*

```
// ER 9.1 - USCI_B1.h
// USCI_B1_Header
// Definições para USCI_B1

#ifndef USCI_B1_H_
```

```
#define USCI_B1_H_

#define MAX_B1 10 //Tamanho do vetor para receber dados

// Protótipo das funções
void USCI_B1_config(void);
void USCI_B1_rx(void);
void USCI_B1_tx(void);

// Variáveis globais
volatile char b1_vet[MAX_B1]; //Vetor para receber dados
volatile char b1; //Indexador para o vetor
volatile char b1_cont; //Contador para transmitir

#endif /* USCI_B1_H_ */
```

A próxima listagem traz o arquivo com as funções que fazem a USCI\_B1 operar como escravo. Note que após as inicializações, o vetor para receber os dados é zerado. Depois, seu indexador e o contador para a transmissão também são zerados. A parte importante é a interrupção que permite que este programa funcione de forma silenciosa quando o leitor for usá-lo no próximo exercício.

#### *Listagem da solução do ER 9.1*

```
//ER 9.1 - USCI_B1.c
// USCI_B1 programada com escravo com endereço = 42
// Escravo transmissor envia 0, 1, 2, ..., 255, 1, 2, ...
// Escravo receptor guardar os últimos 10 bytes
// P4.1 = SDA e P4.2 = SCL

#include <msp430.h>
#include "USCI_B1.h"

void USCI_B1_config(void){
    UCB1CTL1 = UCSWRST; //Resetar USCI_B1
    UCB1CTL0 = UCMODE_3 | //Modo I2C
        UCSYNC; //Síncrono
    UCB1I2COA = 42; //Endereço do Escravo
    UCB1CTL1 = 0; //UCSWRST=0
    UCB1IE = UCTXIE | UCRXIE; //Hab interrup TX e RX
    P4SEL |= BIT2 | BIT1; //Função alternativa
    P4REN |= BIT2 | BIT1; //Hab resistor
    P4OUT |= BIT2 | BIT1; //Pullup
    for (b1=0; b1<MAX_USCI_B1; b1++) //Zerar vetor
        b1_vet[b1]=0;
    b1=0; //Zerar indexador
```

```

    b1_cont=0;                      //Zerar contador
    __enable_interrupt();           //GIE=1, hab int geral
}

// ISR da USCI_B1
#pragma vector = USCI_B1_VECTOR      //Vetor 45
__interrupt void isr_USCI_B1(void){
    int n;
    n = __even_in_range(UCB1IV, 0xC);    //Consultar UCB1IV
    switch(n) {
        case 0x0: break;                case 0x2: break;
        case 0x4: break;                case 0x6: break;
        case 0x8: break;
        case 0xA: USCI_B1_rx();   break; //Rotina RX
        case 0xC: USCI_B1_tx();   break; //Rotina TX
    }
}

// Rotina para receber um byte e guardar no vetor
void USCI_B1_rx(void){
    b1_vet[b1++] = UCB1RXBUF;    //Guardar dado recebido
    if (b1>MAX_USCI_B1) b1=0;    //Verificar limite do vetor
}

// Rotina para transmitir o contador e incrementá-lo
void USCI_B1_tx(void){
    UCB1TXBUF = b1_cont++;       //Transmitir contador
}

```

**ER 9.2.** Usando a USCI\_B0 como mestre, verifique se o escravo 42 (USCI\_B1) responde ao endereçamento. Acenda o *led* verde (Led2, P4.7) caso positivo ou acenda o *led* vermelho (Led1, P1.0) caso negativo. Usando dois cabos, conecte os pinos P3.0 (SDA) e P3.1 (SCL) aos pinos P4.1 (SDA) e P4.2 (SCL), respectivamente. Configure a linha SCL para operar em 10 kHz.

### Solução:

Faremos uso dos arquivos que foram criados no exercício anterior. Por enquanto, não precisamos nos preocupar com seu entendimento. Os arquivos USCI\_B1.c e USCI\_B1.h devem estar presentes no mesmo diretório da solução deste problema. A figura abaixo indica como conectar as duas USCI.

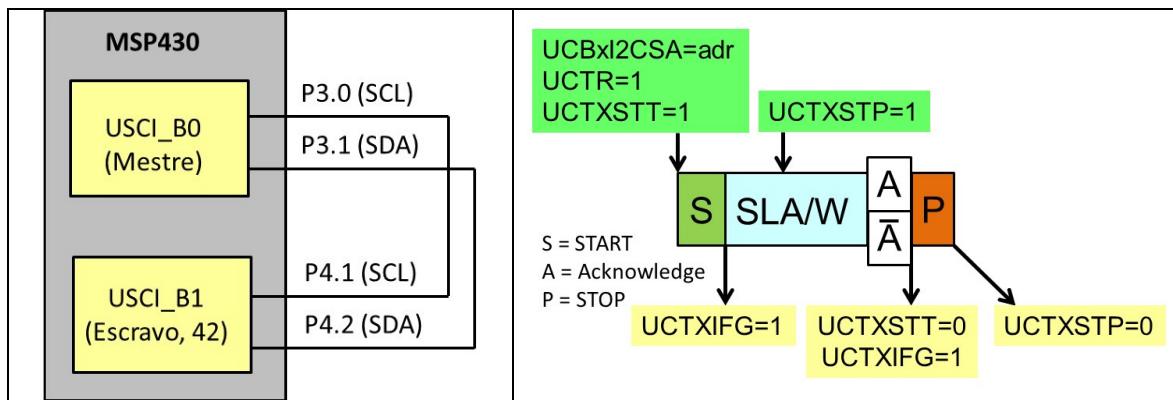


Figura 9.40. Conexão das duas USCI disponíveis no MSP430 F5529 e o diagrama de tempo indicando as ações realizadas e as respostas recebidas pelo barramento I<sup>2</sup>C.

As duas tabelas abaixo indicam as configurações dos diversos registradores. É importante indicar o bit UCMST = 1, que configura esta unidade como mestre. Se ela é mestre, é necessário especificar a frequência do relógio (SCL). Para este exemplo foi escolhida a taxa de 10 kbps gerada a partir do SMCLK (UCSSEL = 3).

$$UCxBRW = \frac{1.048.576}{10.000} = 104,8$$

Tabela 9.6. Configuração da USCI\_B1 para operar como escravo no endereço 42

	7	6	5	4	3	2	1	0
<b>UCB0CTL0</b>	UCA10	UCSLA10	UCMM	-	UCMST	UCMODE		UCSYNC
	0	0	0	0	1	3		1
<b>UCB0CTL1</b>	UCSSEL		-	UCTR	UCTXNACK	UCTXSTP	UCTXSTT	UCWRST
	3		0	0	0	0	0	1/0
<b>UCB0BRW</b>	105 (10 kbps)							

Tabela 9.7. Configuração do GPIO usado pela porta USCI\_B0

	7	6	5	4	3	2	1	0
<b>P3OUT</b>	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
	-	-	-	-	-	-	1	1
<b>P3REN</b>	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
	-	-	-	-	-	-	1	1
<b>P3SEL</b>	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
	-	-	-	-	-	-	1	1

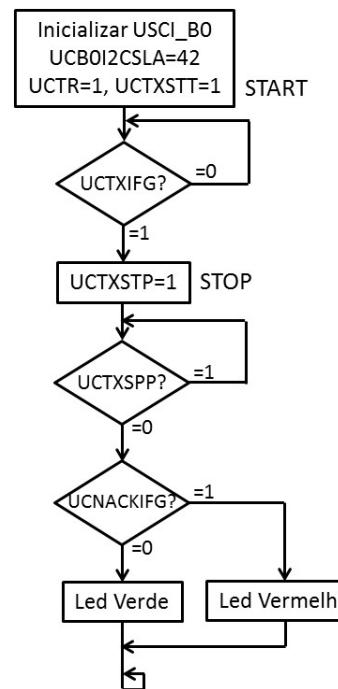
Para entender a sequência de ações necessárias para controlar o mestre que endereça um escravo, apresentamos o fluxograma ao lado.

Após inicializar a USCI\_B0, o programa define o endereço do escravo (42) e indica que ele vai ser endereçado como receptor, pois o mestre é o transmissor (UTR = 1).

O programa pede para gerar a condição de START (UCTXSTT = 1). A flag UCTXIFG = 1 indica que a condição de START foi gerada.

Note que em seguida o programa pede para gerar a condição de STOP (UCTXSTP = 1). Neste caso, após enviar o endereço, a USCI\_B0 recebe a resposta do escravo (ACK ou NACK) e em seguida gera o STOP pedido. Após gerar a condição de STOP, o bit UCTXSTP volta a zero.

Neste momento a flag UCNACKIFG pode ser consultada.



*Figura 9.41. Fluxograma para endereçar o escravo de endereço 42. É gerado um START e um STOP seguidos, sem a transmissão de qualquer dado.*

A listagem abaixo apresenta o programa solução. O leitor deve executar o programa alterando o endereço do escravo, que é especificado na linha UCB0I2CSA = 42. O led verde acende apenas com o endereço 42 e o vermelho para os demais endereços.

#### *Listagem da solução do ER 9.2*

*(No mesmo diretório devem estar presentes os arquivos USCI\_B1.h e USCI\_B1.c).*

```

// ER 9.2
// Endereçar escravo e verificar ACK

// Mestre -> USCI_B0: P3.0 = SDA e P3.1 = SCL
// Escravo -> USCI_B1: P4.1 = SDA e P4.2 = SCL, endereço 42

#include <msp430.h>
#include "USCI_B1.h"           //<--- ER 9.1

#define TRUE    1
#define FALSE   0

#define BR10K 105 //com SMCLK

// Funções usadas
  
```

```

void USCI_B0_config(void);
void leds_config(void);

int main(void){
    WDTCTL = WDTPW | WDTHOLD;      // stop watchdog timer

    USCI_B1_config();      //<--- ER 9.1, USCI_B1 como escrava
    USCI_B0_config();      //USCI_B1 como mestre
    leds_config();         //Configurar Leds
    UCB0I2CSA = 42;        //Endereço do escravo

    UCB0CTL1 |= UCTR | UCTXSTT;           //Gerar START
    while ( (UCB0IFG & UCTXIFG) == 0);   //Esperar pelo START

    UCB0CTL1 |= UCTXSTP;                 //Gerar STOP

No passo a passo, o programa prende aqui no while, STP=0 nunca acontece
Remover esta linha?
    while ( (UCB0CTL1 & UCTXSTP) == UCTXSTP);   //Esperar pelo STOP

    //Teste do ACK
    if ((UCB0IFG & UCNACKIFG) == 0) P4OUT |= BIT7; //ACK
    else
                    P1OUT |= BIT0; //NACK
    while(TRUE);     //Prender execução
    return 0;
}

// Configurar USCI_B0 como mestre
// P3.0 = SDA e P3.1 = SCL
void USCI_B0_config(void){
    UCB0CTL1 = UCSWRST;      //Ressetar USCI_B1
    UCB0CTL0 = UCMST |      //Modo Mestre
                UCMODE_3 | //I2C
                UCSYNC;      //Síncrono
    UCB0BRW = BR10K;        //10 kbps
    UCB0CTL1 = UCSSEL_3;    //SMCLK e UCSWRST=0
    P3SEL |= BIT1 | BIT0;   //Funções alternativas
    P3REN |= BIT1 | BIT0;
    P3OUT |= BIT1 | BIT0;
}

// Configurar Leds
void leds_config(void){
    P1DIR |= BIT0;          P1OUT &= ~BIT0; //Led Vermelho
    P4DIR |= BIT7;          P4OUT &= ~BIT7; //Led Verde
}

```

**ER 9.3.** Neste exercício vamos fazer nossa primeira transmissão. Vamos o byte 85 (0x55) para o escravo 42 e conferir na posição 0 do vetor `b1_vet`, se ele realmente chegou.

### Solução:

Os arquivos `USCI_B1.c` e `USCI_B1.h` devem estar presentes no mesmo diretório da solução deste problema, pois eles simulam um escravo no endereço 42.

Vamos endereçar o escravo 42 como receptor e enviar para ele um único byte (0x55). Depois, com o CCS, vamos consultar a posição zero do vetor `b1_vet`, que é onde o escravo armazena os resultados, para conferir se o dado enviado foi recebido. Para enviar um único dado, é preciso um certo cuidado. O segredo está em solicitar a geração da condição de STOP assim que a transmissão do dado for iniciada. Abaixo está a explicação detalhada das etapas deste programa.

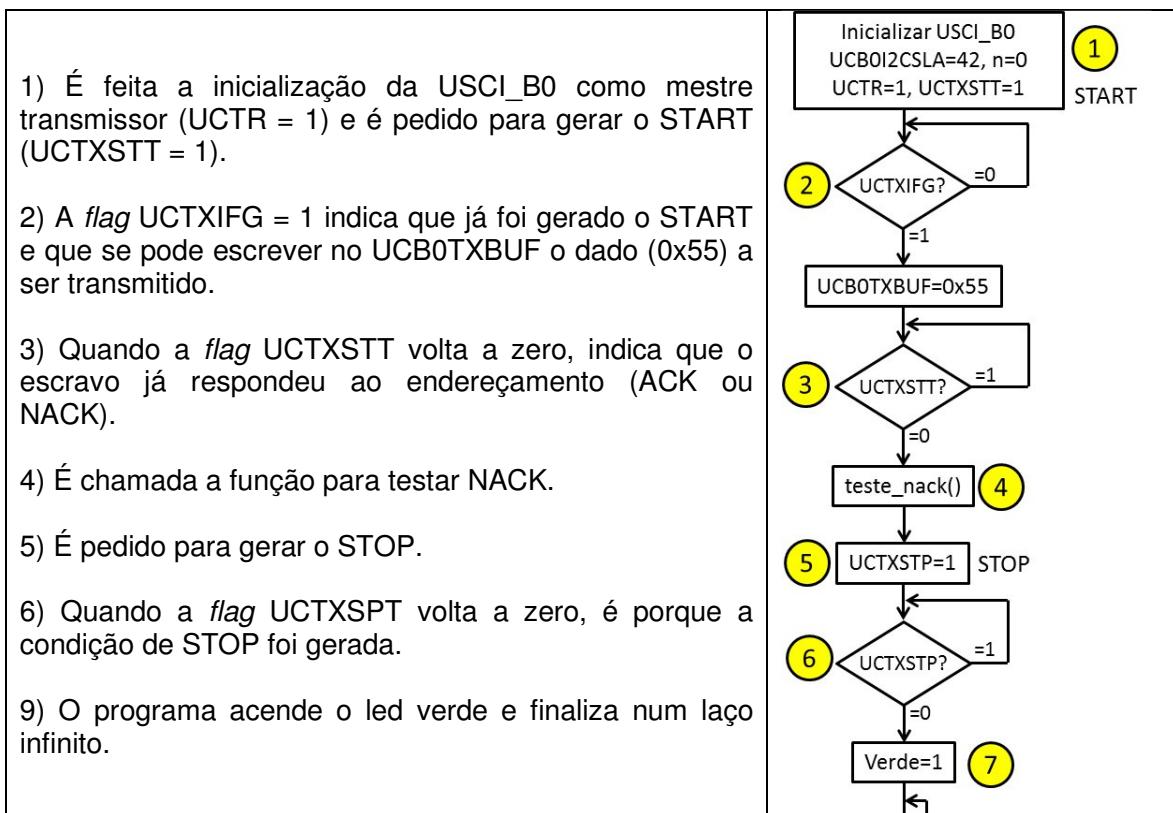


Figura 9.42. Fluxograma para endereçar o escravo de endereço 42 e enviar um dado.

A função `teste_nack` é muito simples. Ela testa a flag `UCNACKIFG`. Caso esteja em 1 é porque o escravo enviou um NACK. Neste caso ela simplesmente acende o led vermelho e prende a execução num laço infinito.

*Listagem da solução do ER 9.3**(No mesmo diretório devem estar presentes os arquivos USCI\_B1.h e USCI\_B1.c).*

```

// ER 9.3
// Endereçar escravo e transmitir 0x55
// Transmitir um único dado

// Mestre -> USCI_B0: P3.0 = SDA e P3.1 = SCL
// Escravo -> USCI_B1: P4.1 = SDA e P4.2 = SCL, endereço 42

#include <msp430.h>
#include "USCI_B1.h"           // <--- ER 9.1

#define TRUE    1
#define FALSE   0

#define BR10K 105 // com SMCLK

// Funções usadas
void teste_nack(void);
void config_USCI_B0(void);
void config_leds(void);

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;    // stop watchdog timer

    config_USCI_B1();          // <--- ER 9.1, USCI_B1 como escrava
    config_USCI_B0();          // USCI_B1 como mestre
    config_leds();              // Configurar Leds
    UCB0I2CSA = 42;            // Endereço do escravo

    UCB0CTL1 |= UCTR | UCTXSTT;      // Gerar START
    while ( (UCB0IFG & UCTXIFG) == 0); // Esperar TXIFG=1 (START)
    UCB0TXBUF=0x55;                // Primeiro número no TXBUF

    while ( (UCB0CTL1 & UCTXSTT) == UCTXSTT); // Esperar TXSTT=0
    (confirmação)
    teste_nack();

    UCB0CTL1 |= UCTXSTP;           // Gerar STOP
    while ( (UCB0CTL1 & UCTXSTP) == UCTXSTP); // Esperar STOP

    P4OUT |= BIT7; // Verde indica sucesso
    while(TRUE); // Prender execução
    return 0;
}

// Verificar se aconteceu NACK
void teste_nack(void) {

```

```

if (((UCB0IFG & UCNACKIFG) == UCNACKIFG) {
    P1OUT |= BIT0;                                //Vermelho=falha
    UCB0CTL1 |= UCTXSTP;                          //Gerar STOP
    while ((UCB0CTL1 & UCTXSTP) == UCTXSTP);     //Esperar STOP
    while(1);          //Travar
}

// Configurar USCI_B0 como mestre
// P3.0 = SDA e P3.1 = SCL
void config_USCI_B0(void){
    UCB0CTL1 = UCSWRST;           //Ressetar USCI_B1
    UCB0CTL0 = UCMST      |     //Modo Mestre
                UCMODE_3  |     //I2C
                UCSYNC;        //Síncrono
    UCB0BRW = BR10K;             //10 kbps
    UCB0CTL1 = UCSSEL_3;         //SMCLK e UCSWRST=0
    P3SEL |= BIT1 | BIT0;        //Funções alternativas
    P3REN |= BIT1 | BIT0;
    P3OUT |= BIT1 | BIT0;
}

// Configurar Leds
void config_leds(void){
    P1DIR |= BIT0;      P1OUT &= ~BIT0;
    P4DIR |= BIT7;      P4OUT &= ~BIT7;
}

```

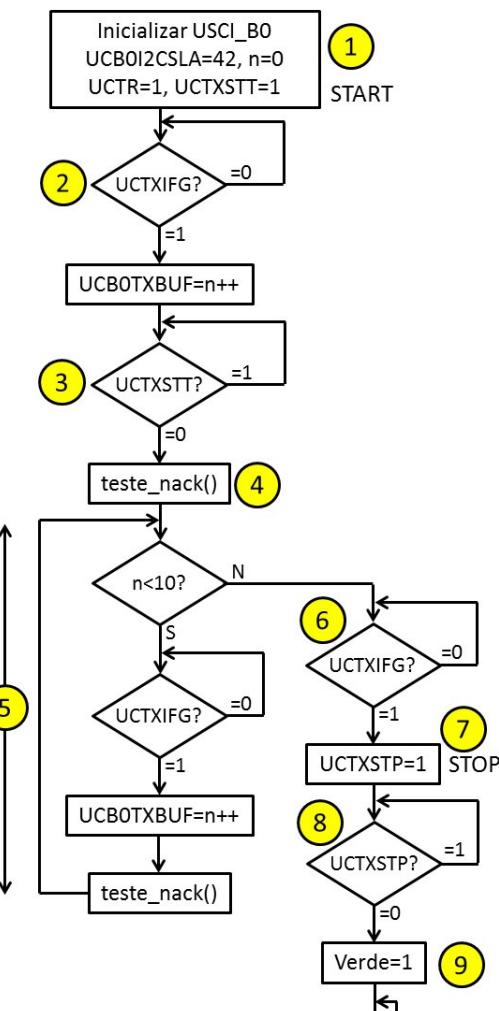
**ER 9.4.** Se o exercício anterior funcionou, vamos fazer uma transmissão mais sofisticada. Vamos enviar a sequência 0, 1, 2, ..., 9 para o escravo 42.

#### Solução:

Os arquivos USCI\_B1.c e USCI\_B1.h devem estar presentes no mesmo diretório da solução deste problema, pois eles simulam um escravo no endereço 42.

Agora vamos endereçar o escravo 42 como receptor e enviar para ele a sequência de 0 até 9. Depois, com o CCS, vamos consultar o vetor `b1_vet` que é onde o escravo armazena os resultados e conferir se o programa funcionou. Abaixo está a explicação da solução, indicando os pontos mais importantes e, em seguida, é apresentada a listagem da solução.

- 1) É feita a inicialização da USCI\_B0 como mestre transmissor (UCTR = 1), o contador n é inicializado e é pedido para gerar o START (UCTXSTT = 1).
  - 2) A flag UCTXIFG = 1 indica que já foi gerado o START e que se pode escrever no UCB0TXBUF o primeiro dado a ser transmitido.
  - 3) Quando a flag UCTXSTT volta a zero, indica que o escravo já respondeu ao endereçamento (ACK ou NACK).
  - 4) É chamada a função para testar NACK.
  - 5) Corresponde ao laço de transmissão onde se espera UCTXIFG = 1, escreve-se o novo dado em UCB0TXBUF e consulta-se a resposta do escravo (ACK ou NACK).
  - 6) Ao sair do laço, é preciso esperar UCTXIFG = 1 para se ter certeza de que a transmissão o último dado já foi iniciado. Do contrário, o último dado pode não ser transmitido.
  - 7) Após iniciar a transmissão do último dado, é solicitada a geração de STOP (UCTXSTP).
  - 8) Quando a flag UCTXSPT volta a zero, é porque a condição foi gerada.
  - 9) O programa acende o led verde e finaliza num laço infinito.



*Figura 9.43. Fluxograma para endereçar o escravo de endereço 42 e enviar uma sequência de dados.*

### *Listagem da solução do ER 9.4*

(No mesmo diretório devem estar presentes os arquivos USCI\_B1.h e USCI\_B1.c).

```
// ER 9.4
// Endereçar escravo e transmitir 0, 1, ..., 9

// Mestre -> USCI_B0: P3.0 = SDA e P3.1 = SCL
// Escravo -> USCI_B1: P4.1 = SDA e P4.2 = SCL, endereço 42

#include <msp430.h>
#include "USCI_B1.h"           //<--- ER 9.1
```

```
#define TRUE    1
#define FALSE   0

# define BR10K 105 //com SMCLK

// Funções usadas
void teste_nack(void);
void USCI_B0_config(void);
void leds_config(void);

int main(void){
    char n=0;      //números a serem enviados
    WDTCTL = WDTPW | WDTHOLD;    // stop watchdog timer

    USCI_B1_config();    //<-- ER 9.1, USCI_B1 como escrava
    USCI_B0_config();    //USCI_B0 como mestre
    leds_config();        //Configurar Leds
    UCB0I2CSA = 42;      //Endereço do escravo

    UCB0CTL1 |= UCTR | UCTXSTT;           //Gerar START
    while ( (UCB0IFG & UCTXIFG) == 0);    //Esperar TXIFG=1 (START)
    UCB0TXBUF=n++;                      //Primeiro número no TXBUF

    while ( (UCB0CTL1 & UCTXSTT) == UCTXSTT);    //Esperar TXSTT=0
    teste_nack();

    //Transmitir o restante
    while(n<10){
        while ( (UCB0IFG & UCTXIFG) == 0);        //Esperar TXIFG=1 (START)
        UCB0TXBUF=n++;                          //Número seguinte
        teste_nack();
    }
    // Certificar que iniciou a transmissão do último número
    while ( (UCB0IFG & UCTXIFG) == 0);    //TXIFG=1 --> iniciou trans.

    UCB0CTL1 |= UCTXSTP;                  //Gerar STOP
    while ( (UCB0CTL1 & UCTXSTP) == UCTXSTP); //Esperar STOP

    P4OUT |= BIT7;    //Verde indica sucesso
    while(TRUE);     //Prender execução
    return 0;
}

void teste_nack(void){ ... }          //Copiar do ER 9.3
void USCI_B0_config(void){ ... }      //Copiar do ER 9.3
void leds_config(void){ ... }         //Copiar do ER 9.3
```

**ER 9.5.** Vamos agora trabalhar como mestre receptor e receber um único dado do escravo 42. Lembre-se de que o primeiro dado que o escravo (USCI\_B1) transmite é o zero.

**Solução:**

Os arquivos USCI\_B1.c e USCI\_B1.h devem estar presentes no mesmo diretório da solução deste problema, pois eles simulam um escravo no endereço 42. É preciso lembrar que a USCI\_B1 quando endereçada como transmissora, envia a sequência 0, 1, 2, ..., 255, 0, 1, ...

Agora vamos endereçar o escravo 42 como transmissor e receber apenas um *byte*. De acordo com o programa USCI\_B1.c será o *byte* zero. Depois vamos usar o CCS, para verificar o dado recebido. Para que se receba um único *byte*, é importante gerar o STOP assim que se inicia a recepção do primeiro *byte*, ou seja, logo após o ACK do escravo. O bit UCTXNACK é mantido em zero, o que significa que um ACK é gerado pelo mestre após a recepção do dado.

O leitor deve notar que foi criada a variável `volatile char dado=0xFF` para receber o dado transmitido pelo escravo. Como se sabe que o escravo vai transmitir zero, essa variável é inicializada com 0xFF. O programa não faz qualquer uso dessa variável, e então há o risco de o compilador removê-la, por isso foi usado o modificador `volatile`. Após executar o programa, é possível consultar a variável `dado` e verificar que ela mudou para zero, como era esperado.

A variável `b1_cont` pode ser consultada. Espera-se que seja igual a 1, já que o escravo transmitiu um único dado. Entretanto, vamos encontrá-la igual a 2, o que indica que o escravo transmitiu um *byte* que não foi recebido pelo mestre. O leitor é convidado a fazer tentativas para evitar que o escravo envie este *byte* extra.

Logo a seguir é apresentado o fluxograma detalhado de todas as etapas realizadas, junto com as explicações pertinentes. Em seguida está a listagem do programa solução.

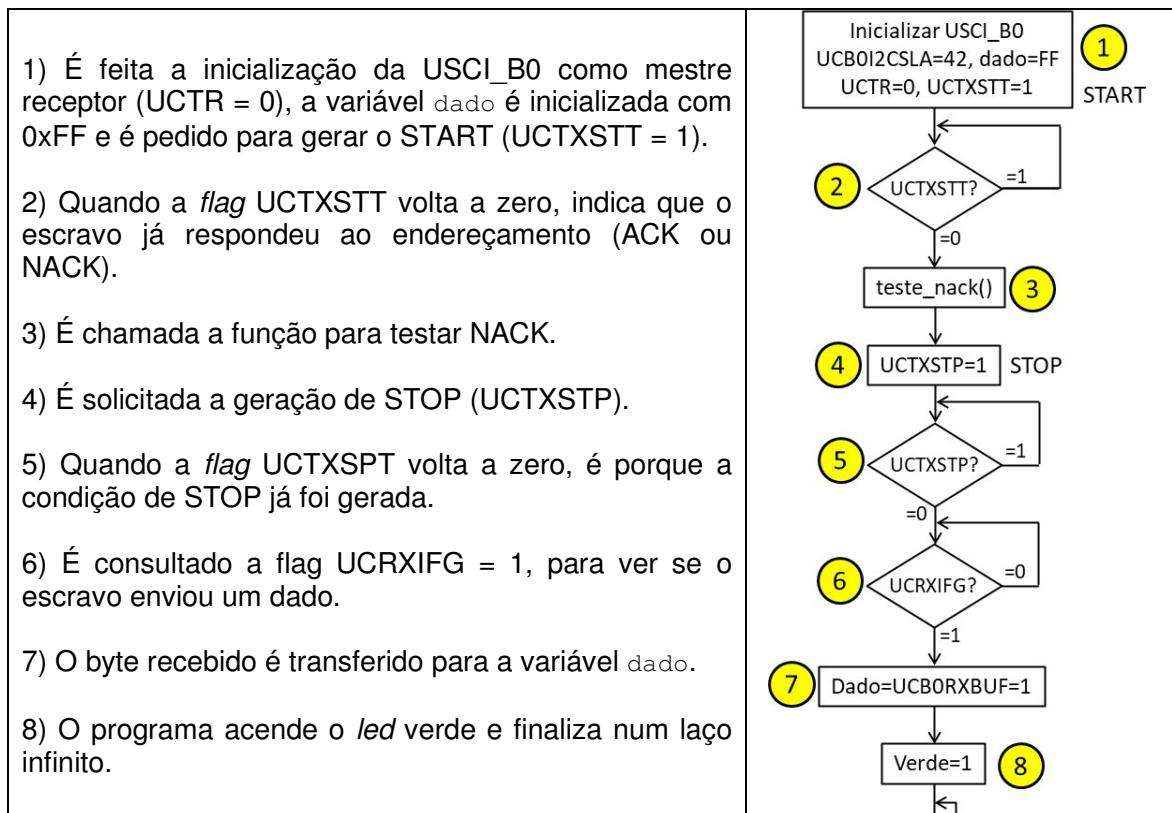


Figura 9.44. Fluxograma para endereçar o escravo de endereço 42 e receber um dado.

#### Listagem da solução do ER 9.5

(No mesmo diretório devem estar presentes os arquivos `USCI_B1.h` e `USCI_B1.c`).

```

// ER 9.5
// Endereçar escravo 42 para transmitir
// Mestre recebe um único dado

// Mestre -> USCI_B0: P3.0 = SDA e P3.1 = SCL
// Escravo -> USCI_B1: P4.1 = SDA e P4.2 = SCL, endereço 42

#include <msp430.h>
#include "USCI_B1.h"           //<--- ER 9.1

#define TRUE    1
#define FALSE   0

#define BR10K 105 //com SMCLK

// Funções usadas
void teste_nack(void);
void USCI_B0_config(void);
  
```

```

void config_leds(void);

int main(void)
{
    volatile char dado=0xFF;
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer

    USCI_B1_config(); //<--- ER 9.1, USCI_B1 como escrava
    USCI_B0_config(); //USCI_B0 como mestre
    leds_config(); //Configurar Leds
    UCB0I2CSA = 42; //Endereço do escravo

    UCB0CTL1 |= UCTXSTT; //Gerar START
    while ( (UCB0CTL1 & UCTXSTT) == UCTXSTT); //Esperar TXSTT=0
    teste_nack();
    UCB0CTL1 |= UCTXSTP; //Gerar STOP

    while ( (UCB0CTL1 & UCTXSTP) == UCTXSTP); //Esperar STOP

    while ( (UCB0IFG & UCRXIFG) == 0); //Esperar RXIFG=1
        dado=UCB0RXBUF; //Receber o dado

    P4OUT |= BIT7; //Verde indica sucesso
    while(TRUE); //Prender execução
    return 0;
}

void teste_nack(void){ ... } //Copiar do ER 9.3
void USCI_B0_config(void){ ... } //Copiar do ER 9.3
void leds_config(void){ ... } //Copiar do ER 9.3

```

**ER 9.6.** Agora que temos certeza de que conseguimos receber um dado do escravo, vamos tentar uma sequência de 10 dados.

### Solução:

Os arquivos USCI\_B1.c e USCI\_B1.h devem estar presentes no mesmo diretório da solução deste problema, pois eles simulam um escravo no endereço 42. É preciso lembrar que a USCI\_B1 quando endereçada como transmissora, envia a sequência 0, 1, 2, ..., 255, 0, 1, ...

Agora vamos endereçar o escravo 42 como transmissor e receber apenas 10 bytes. É pedido para gerar a condição de STOP após a recepção do nono byte.

Usando o CCS podemos verificar o conteúdo da variável `vetor` e constatar que foram recebidos os 10 valores esperados. Além disso, vamos ver que a variável `b1_cont` está

em 11, o que é o valor esperado. Isto significa que o escravo não transmitiu nenhum dado extra. O bit UCTXNACK é mantido em zero, o que significa que um ACK é gerado pelo mestre após a recepção de cada byte.

A seguir apresentamos um fluxograma com os detalhes desta recepção e a listagem do programa solução.

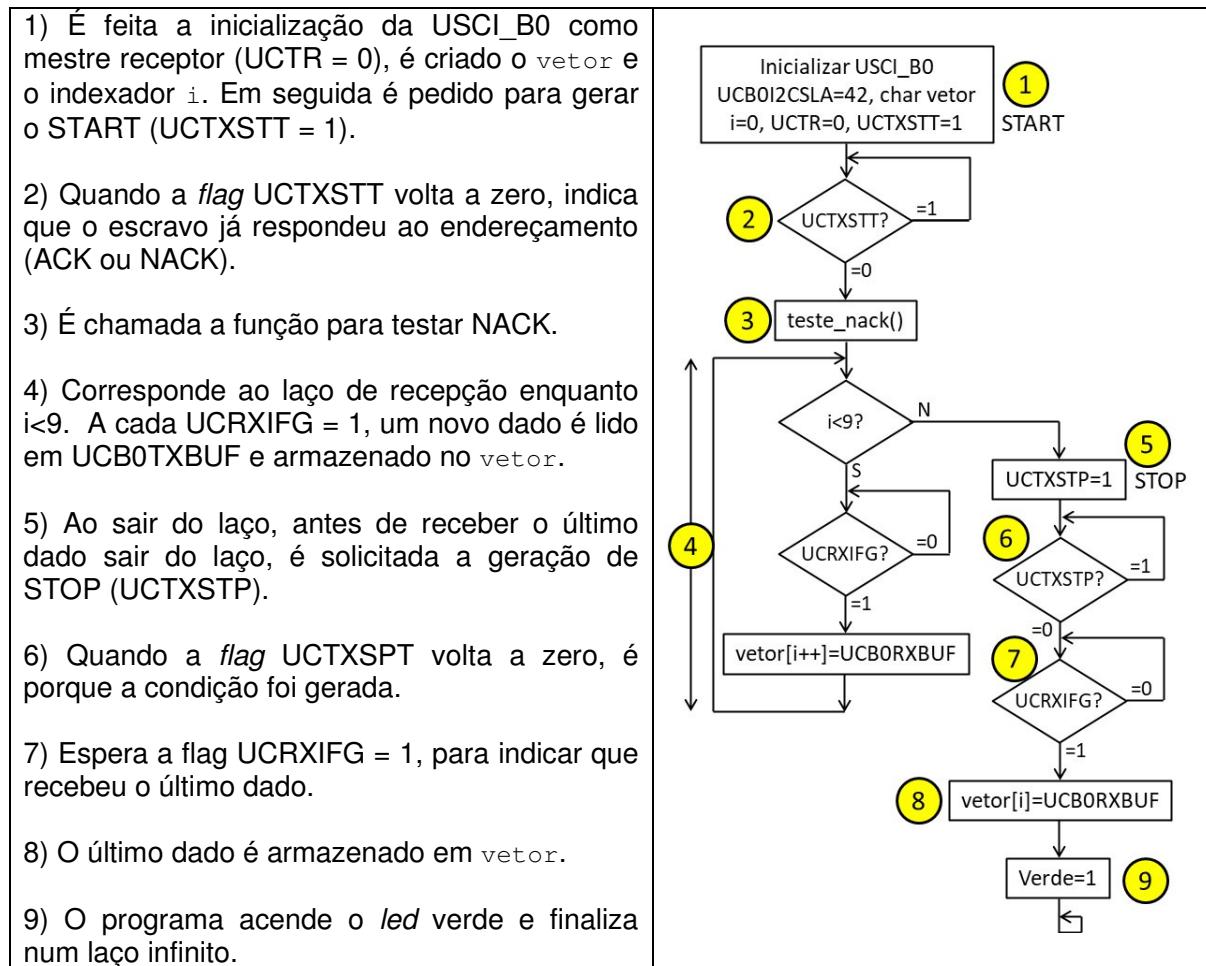


Figura 9.45. Fluxograma para receber do escravo de endereço 42 uma sequência de dados.

#### Listagem da solução do ER 9.6

(No mesmo diretório devem estar presentes os arquivos USCI\_B1.h e USCI\_B1.c).

```

// ER 9.6
// Endereçar escravo 42 para transmitir
// Receber 10 bytes
  
```

```

// Mestre -> USCI_B0: P3.0 = SDA e P3.1 = SCL
// Escravo -> USCI_B1: P4.1 = SDA e P4.2 = SCL, endereço 42

#include <msp430.h>
#include "USCI_B1.h"           //<--- ER 9.1

#define TRUE    1
#define FALSE   0

#define BR10K 105 //com SMCLK

// Funções usadas
void teste_nack(void);
void USCI_B0_config(void);
void leds_config(void);

int main(void)
{
    volatile char vetor[10], i=0;
    WDTCTL = WDTPW | WDTHOLD;    // stop watchdog timer

    USCI_B1_config();          //<--- ER 9.1, USCI_B1 como escrava
    USCI_B1_config();          //USCI_B0 como mestre
    leds_config();              //Configurar Leds
    UCB0I2CSA = 42;            //Endereço do escravo

    UCB0CTL1 |= UCTXSTT;        //Gerar START
    while ( (UCB0CTL1 & UCTXSTT) == UCTXSTT); //Esperar TXSTT=0
    teste_nack();

    //Laço para receber 10 bytes
    while(i<9){
        while ( (UCB0IFG & UCRXIFG) == 0);      //Esperar RXIFG=1
        vetor[i++]=UCB0RXBUF;                    //Receber o dado
    }

    UCB0CTL1 |= UCTXSTP;          //Gerar STOP
    while ( (UCB0CTL1 & UCTXSTP) == UCTXSTP); //Esperar STOP

    while ( (UCB0IFG & UCRXIFG) == 0);      //Esperar RXIFG=1
    vetor[i]=UCB0RXBUF;              //Receber o dado

    P4OUT |= BIT7;    //Verde indica sucesso
    while(TRUE);      //Prender execução
    return 0;
}

void teste_nack(void){ ... }      //Copiar do ER 9.3
void USCI_B0_config(void){ ... }  //Copiar do ER 9.3

```

void leds_config(void) { ... }	//Copiar do ER 9.3
--------------------------------	--------------------

**ER 9.7.** Escreva um programa que descubra o endereço de todos os dispositivos I<sup>2</sup>C que estão presentes no barramento. Este programa é muito útil para fazer o teste inicial de toda a conexão I<sup>2</sup>C. Use dois vetores: um para guardar os endereços dos escravos receptores e outro para guardar o endereço dos escravos transmissores. O primeiro elemento destes vetores deve ser a quantidade (*qtd*) de endereços descobertos, como mostrado a seguir.

char vet\_wr = [qtd, adr1, adr2, ...] → endereços dos escravos receptores  
 char vet\_rd = [qtd, adr1, adr2, ...] → endereços dos escravos transmissores

### Solução:

Este exercício é, de certa forma, exagerado porque um escravo I<sup>2</sup>C normalmente opera como transmissor e receptor. Entretanto, ele foi proposto assim, para esclarecer as duas formas de se endereçar escravos. Para facilitar a solução, o programa apresentado a seguir faz uso da função indicada abaixo que retorna `TRUE` se o escravo respondeu e `FALSE` caso negativo.

- char i2c\_test (char adr, char modo)
  - adr é o endereço, dentro da faixa de 0 até 127, a ser testado
  - modo é o modo de endereçamento do escravo (receptor ou transmissor).

A Figura 9.46 apresenta o diagrama de tempo indicando as operações realizadas nos dois casos. Nossa ação é endereçar um escravo e verificar se ele responde com um ACK ou um NACK. No caso de um ACK, é porque o escravo desse endereço está presente.

Na busca por um escravo receptor é possível pedir para gerar uma condição de STOP enquanto o endereço é enviado. Isto evita a fase de transmissão de dado. Entretanto, na busca por um escravo transmissor, só é garantido gerar a condição de STOP após o *flag* UCTXSTT ir para 0, quando então já foi iniciada a fase de recepção de um dado. Neste caso não é possível evitar a transmissão de um dado. Então, para indicar ao escravo transmissor que este dado recebido é o único, o programa ativa o *bit* UCTXNACK para gerar um NACK em resposta ao dado enviado pelo escravo transmissor.

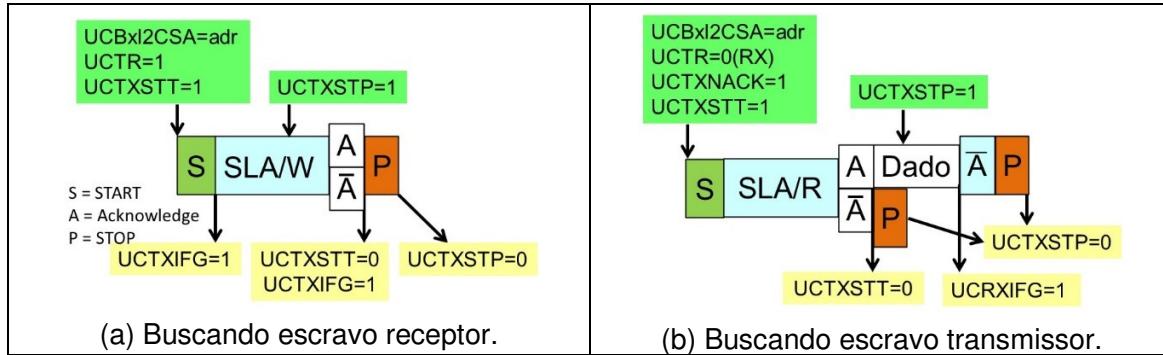


Figura 9.46. Diagrama de tempo indicando as operações realizadas para testar um escravo receptor ou um escravo transmissor.

A seguir é apresentada a listagem completa do programa solução, cujo funcionamento é simples. O programa faz as inicializações e fica aguardando o acionamento da chave S1. Ao acionar a chave S1, o led vermelho é aceso enquanto é feita uma busca completa pelos escravos. Quando a busca termina, o led verde é aceso e o programa volta a aguardar o usuário acionar a chave S1, quando então repete a busca. Note que a constante MAX\_ADDR define a quantidade máxima de escravos que podem ser encontrados. Na listagem está marcada com cores os trechos responsáveis pela busca dos dois tipos de escravos.

#### Listagem da solução do ER 9.7

```
// ER 9.7
// Varredura de endereços I2C
// Led vermelho aceso = buscando
// Led verde aceso = busca terminada
// Usa USCI_B0: P3.0 = SDA e P3.1 = SCL
// Chave S1 dá início à busca

#include <msp430.h>

#define MAX_ADDR 50

#define TRUE    1
#define FALSE   0

#define ESCR_RD  1 //Endereçar Escravo para ler
#define ESCR_WR  0 //Endereçar Escravo para escrever

#define BR10K 105 //com SMCLK

void USCI_B0_config(void);
char i2c_test(char adr, char modo);
void leds_s1_config(void);
void rebote(int x);
```

```

int main(void) {
    char vet_wr[MAX_ADR];          //Lista endereços Escravo Receptor
    char vet_rd[MAX_ADR];          //Lista endereços Escravo Transmissor
    char i;                        //Indexador dos vetores
    char adr;                      //Endereço de escravo

    WDTCTL = WDTPW | WDTHOLD;      // stop watchdog timer
    leds_s1_config();              //Leds e S1
    USCI_B0_config();              //Configurar USCI_B0

    while(TRUE){                   //Laço infinito

        while ( (P2IN&BIT1) == BIT1); //Esperar S1
        P4OUT &= ~BIT7;             //Apagar verde
        P1OUT |= BIT0;              //Acender vermelho
        rebote(1000);

        // Zerar vetores
        for (i=0; i<MAX_ADR; i++)
            vet_wr[i]=vet_rd[i]=0;

        // Procurar pelos escravos transmissores
        i=1;
        for (adr=0; adr<127; adr++){
            if (i2c_test(adr,ESCR_RD) == TRUE)
                vet_rd[i++]=adr;
            if (i==MAX_ADR) break;
        }
        vet_rd[0]=i-1;

        // Procurar pelos escravos receptores
        i=1;
        for (adr=0; adr<127; adr++){
            if (i2c_test(adr,ESCR_WR) == TRUE)
                vet_wr[i++]=adr;
            if (i==MAX_ADR) break;
        }
        vet_wr[0]=i-1;

        P1OUT &= ~BIT0;              //Apagar vermelho
        P4OUT |= BIT7;               //Acender verde
        while ( (P2IN&BIT1) == 0);   //Esperar soltar S1
        rebote(1000);
    }

    while(TRUE); //Travar execução
    return 0;
}

// Testar o endereço adr para escrita ou leitura
// modo = ESCR_RD --> Endereçar escravo para leitura

```

```

// modo = ESCR_WR --> Endereçar escravo para escrita
char i2c_test(char adr, char modo){
    UCB0I2CSA = adr; //Endereço a ser testado

    if (modo==ESCR_WR){ //WR = Escravo receptor
        UCB0CTL1 |= UCTR; //Mestre TX --> escravo RX
        UCB0CTL1 |= UCTXSTT; //Gerar STASRT
        while ( (UCB0IFG & UCTXIFG) == 0); //TXIFG=1, START iniciado
    }
    else{ //RD = Escravo transmissor
        UCB0CTL1 &= ~UCTR; //Mestre RX <-- escravo TX
        UCB0CTL1 |= UCTXNACK; //NACK ao receber um dado
        UCB0CTL1 |= UCTXSTT; //Gerar START
        while ( (UCB0CTL1 & UCTXSTT) == UCTXSTT); //Esperar START
    }

    UCB0CTL1 |= UCTXSTP; //Gerar STOP
    while ( (UCB0CTL1 & UCTXSTP) == UCTXSTP); //Esperar STOP

    //Teste do ACK
    if ((UCB0IFG & UCNACKIFG) == 0) return TRUE; //Chegou ACK
    else return FALSE; //Chegou NACK
}

// Configurar USCI_B0 como mestre
// P3.0 = SDA e P3.1 = SCL
void USCI_B0_config(void){
    UCB0CTL1 = UCSWRST; //Ressetar USCI_B1
    UCB0CTL0 = UCMST | //Modo Mestre
                UCMODE_3 | //I2C
                UCSYNC; //Síncrono
    UCB0BRW = BR10K; //10 kbps
    UCB0CTL1 = UCSSEL_3; //SMCLK e UCSWRST=0
    P3SEL |= BIT1 | BIT0; //Funções alternativas
    P3REN |= BIT1 | BIT0;
    P3OUT |= BIT1 | BIT0;
}

// Configurar I/O
void leds_s1_config(void){
    P1DIR |= BIT0; P1OUT &= ~BIT0; //Led Vermelho
    P4DIR |= BIT7; P4OUT &= ~BIT7; //Led Verde
    P2DIR &= ~BIT1; //P2.1 = entrada
    P2REN |= BIT1; //Habilitar resistor
    P2OUT |= BIT1; //Selecionar Pullup
}

// Delay para evitar rebotes
void rebote(int x){
    volatile int i;
}

```

```

    for (i=0; i<x; i++);
}

```

**ER 9.8.** Vamos agora para um exercício um pouco mais sofisticado. Infelizmente sua descrição é longa. Usaremos o CI PCF8574 que é um expansor com 8 portas de I/O quasi-bidirecionais e interface I<sup>2</sup>C. Sua foto está nas Figuras 9.47.a e 9.47.b. O leitor é convidado para buscar pelo seu manual na Internet. Para facilitar seu uso, ao invés do *chip*, vamos trabalhar com uma pequena placa (muito comum no mercado) que já traz esse CI soldado, como mostrado na Figura 9.47.c. Esta placa foi projetada para ser montada na parte traseira de um LCD e facilitar seu uso, como está na foto da Figura 9.48.d. Entretanto, por enquanto, vamos usar apenas a placa.

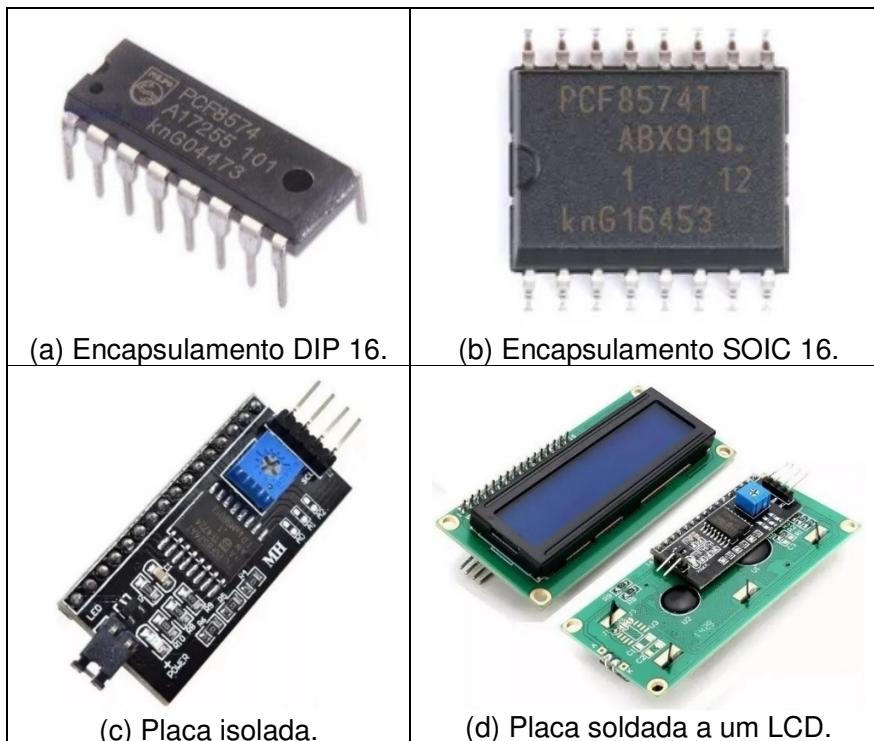


Figura 9.47. Dois tipos de encapsulamento do chip PCF8574 e seu uso em uma pequena placa para acionar um display LCD.

A Figura 9.48 apresenta o esquema elétrico da placa em questão. O conector CON1 permite a conexão com o barramento I<sup>2</sup>C do MSP430. Note que já estão presentes os dois resistores (4,7 kΩ) de *pullup* para as linhas SCL e SDA. Pelo conector CON2 vamos ter acesso à porta de I/O do PCF8574. Dos 8 pinos disponíveis (P0, P1, ..., P7) não vamos usar o P3, porque ele comanda um transistor. Os outros 7 pinos estão disponíveis. Existem duas versões do PCF8574 com endereços diferentes:

- PCF8574 → com o endereço 0x27 e

- PCF8574T → com o endereço 0x3F.

O usuário terá de descobrir e confirmar qual é o endereço usado pela sua placa. O exercício anterior pode ajudar.

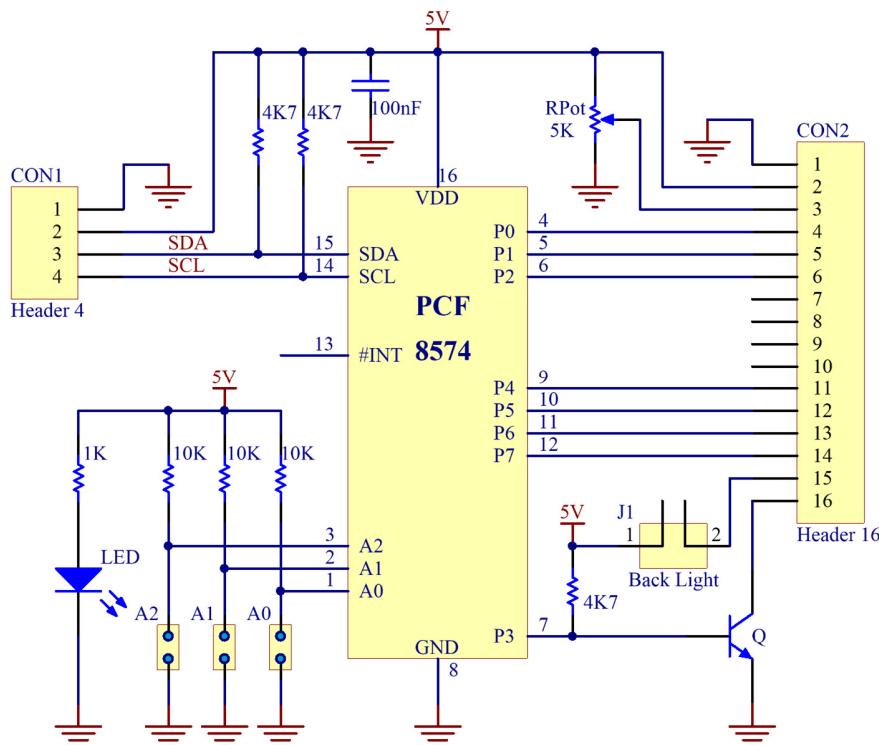


Figura 9.48. Esquema elétrico da placa com o CI PCF8574 a ser usada neste exercício.

Pedido: Usando a placa com o PCF8574 construa um contador binário de 4 bits, controlado por três chaves, como mostrado na Figura 9.49, com as seguintes funções:

- SW+ → soma 1 ao contador;
  - SW- → subtraí 1 do contador e
  - SWC → Zera o contador.

Para auxiliar na montagem, é recomendado o uso de um protoboard. O pino 1 do conector CON2 é o que está mais próximo do conector I<sup>2</sup>C que é o CON1.

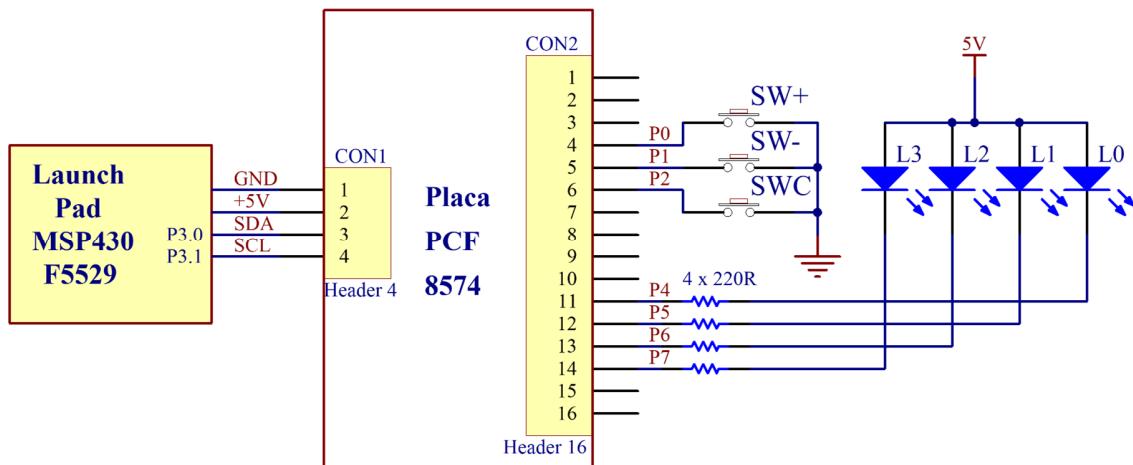


Figura 9.49. Uso da placa com o PCF8574 para construir um contador de 4 bits controlado por 3 chaves.

### Solução:

O objetivo deste exercício é exemplificar o uso de um dispositivo I<sup>2</sup>C para leitura e escrita. Antes de darmos início à solução, é preciso fazermos alguns comentários sobre o uso da placa com o CI PCF8574. Na Figura 9.48 é possível ver que ela já traz os resistores de *pullup*, demandados pelo I<sup>2</sup>C. Entretanto, esses dois resistores estão conectados aos 5V. Será que isso não vai trazer problemas para nosso MSP que está operando em 3,3 V? Será que não vai acionar o diodo de proteção e, eventualmente, queimar um pino?

A Figura 9.50 apresenta apenas os itens que nos interessam nesta questão. Já estudamos que o diodo de proteção entra em ação quando a tensão no pino ultrapassa os 3,6 V e precisamos então calcular a corrente que vai passar por esse diodo, sabendo que seu limite é de 2 mA. Como mostrado nesta figura, a corrente será de 298 µA, o que é bem tolerado pelo diodo de proteção. Concluímos, então, que o *pullup* conectado aos 5 V não causa problemas para o MSP.

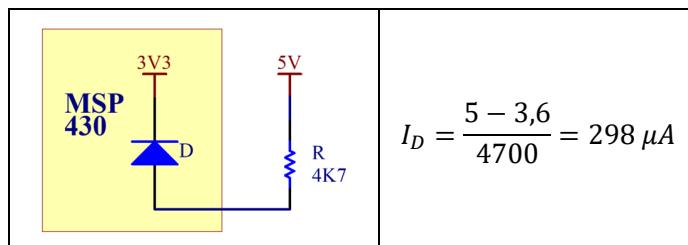


Figura 9.50. Acionamento do diodo de proteção pelo circuito de *pullup* do barramento I<sup>2</sup>C e cálculo da corrente por esse componente.

O leitor pode ter estranhado que na Figura 9.49 os *leds* parecem estar invertidos. Cada *led* acende quando o pino correspondente está em nível baixo. Isto tem uma razão. De acordo com o manual do PCF8574, cada pino tem condições de fornecer até 1 mA, o que é muito pouco para acender um *led*. Entretanto, cada pino é capaz de drenar até 25 mA. Assim, fizemos a ligação ao contrário. Um *led* comum, quando aceso tem uma queda de tensão de 2,2 V. Podemos então calcular a corrente consumida por cada *led*, quando aceso. Em geral, procuramos deixar a corrente dos *leds* na faixa de 10 a 20 mA. Isto significa que poderíamos reduzir os resistores para 180 Ω.

$$I_{Led} = \frac{5 - 2,2}{220} = 12,7 \text{ mA}$$

Vamos agora a um detalhe importante deste CI: suas portas são quasi-bidcionais e não bidicionais com as do MSP. Seu comportamento é muito semelhante às linhas da porta I<sup>2</sup>C com dreno aberto e resistor de *pullup*. Quando usada como saída, o usuário pode colocar qualquer pino da porta em nível baixo ou nível alto, de acordo com seu desejo. Entretanto, quando quer usar um pino como entrada, é preciso antes programá-lo para nível alto.

A figura abaixo apresenta o diagrama simplificado de um *bit* desta porta, aqui denominado de Pino x. O transistor Q, quando conduzindo leva o Pino x para nível baixo. Quando este transistor está cortado, o resistor de *pullup* leva o Pino x para nível alto. Porém, para usar o Pino x como entrada, é preciso deixar o transistor Q cortado, o que se consegue ao se programar nível alto neste pino. Assim, um agente externo pode deixar este pino em nível alto ou arrastá-lo para nível baixo. Em suma:

- Como saída, escrever 0 ou 1 à vontade;
- Como entrada, escreve 1 antes de ler o pino.

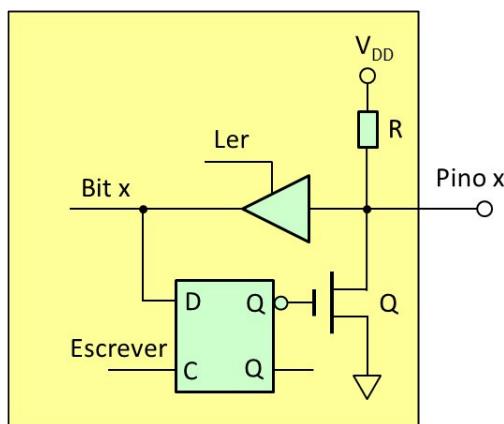


Figura 9.51. Diagrama simplificado de uma porta quasi-bidirecional.

A listagem abaixo apresenta a solução deste exercício. Note que os *bits* P0, P1 e P2 fazem a leitura do estado das chaves, por isso, em toda escrita eles serão mantidos em nível alto. Os bits P4, P5, P6 e P7 acionam os *leds*. Como cada *led* acende quando o pino correspondente está em nível baixo, será necessário inverter os bits do contador. Além disso, é preciso deslocar os *bits* em 4 posições para a esquerda, para que eles ocupem as posições de 4 até 7. Essas operações estão marcadas na listagem em cor abóbora.

Merece comentário a função `i2c_test()` que permite verificar qual o endereço do PCF8574 que está sendo usado. A função `chaves()` é responsável por ler as três chaves e sinalizar quando uma delas é acionada. Veja que esta sinalização é feita nos três *bits* menos significativos do valor retornado. O leitor vai notar a falta de tratamento de rebotes. Como agora o acesso às chaves é feito através do PCF8574 e seu acesso com barramento I<sup>2</sup>C consome tempo, foi possível dispensar o tratamento dos rebotes. Entretanto, se o leitor notar sua necessidade, introduza-os.

Duas observações finais. Note que as funções `i2c_read()` e `i2c_write()`, bem próximo do seu final trazem uma chamada para a função `delay(50)` que não está sendo usada, pois está comentada. Não foi o caso deste exercício, porém, em algumas outras aplicações sentimos a necessidade de gerar um pequeno atraso logo após a condição de STOP. É como se o periférico necessitasse de uma pequena pausa entre um STOP e o próximo START. O segundo comentário é sobre a frequência da linha SCL que agora é de 100 kHz (aproximadamente), o que corresponde ao limite da velocidade deste CI.

Duas sugestões para melhorar esse programa:

- E se ao invés de incrementar o contador (`cont++`) usássemos `cont += 16`?
- Que tal iniciar o contador com `cont = 0xFF` e inverter as operações de incremento e decremento, como mostrado a seguir  
`SW+ → cont -= 16`  
`SW- → cont += 16` e  
`SWC → cont = 0xFF`.

### Listagem da solução do ER 9.8

```
// ER 9.8
// Usa o PCF8574

#include <msp430.h>

#define TRUE      1
#define FALSE     0

#define ABERTA    1
#define FECHADA   0

#define BR10K    105 // (SMCLK) 1.048.576/105 ~= 10kHz
#define BR100K   11 // (SMCLK) 1.048.576/11 ~= 100kHz
```

```

char chaves(void);
char i2c_read(void);
void i2c_write(char dado);
char i2c_test(char adr);
void USCI_B0_config(void);
void leds_config(void);
void delay (int x);

int main(void)
{
    char cont=0,extra; //Contador
    char ler;           //Receber estado das chaves
    char adr;           //Guardar endereço do PCF_8574
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer
    leds_config();       //Leds
    USCI_B0_config();   //Configurar USCI_B0

    if      (i2c_test(0x27) == TRUE) adr=0x27;
    else if (i2c_test(0x3F) == TRUE) adr=0x3F;
    else{
        P1OUT |= BIT0; //Sinalizar problema
        while(TRUE);   //Travar execução
    }
    P4OUT |= BIT7;      //Verde aceso = OK
    UCB0I2CSA = adr;   //Endereço a ser usado
    i2c_write(0xFF);   //Todas as Portas em 1 (leds apagados)

    // Laço Principal
    while(TRUE){
        ler=chaves();
        if ( (ler&BIT0) == BIT0)     cont++;
        if ( (ler&BIT1) == BIT1)     cont--;
        if ( (ler&BIT2) == BIT2)     cont=0;

        extra=(cont<<4)^0xF0;    //Deslocar e inverter
        extra |= 7;               //P0=P1=P2=1
        i2c_write(extra);        //Atualizar leds
    }
    return 0;
}

// Estado das chaves
// Retorna um nr de 0 até 7, um bit por chave
// bit2 = SWC, bit1 = SW-, bit0 = SW+
char chaves(void){
    static char p_mais=ABERTA; //Estado anterior de SW+
    static char p_menos=ABERTA; //Estado anterior de SW-
    static char p_clear=ABERTA; //Estado anterior de SWC
    char aux;                  //Auxiliar

```

```

char res=0;                      //Resultado
aux=i2c_read()&7;                //Separar 3 bits da direita

// Testar SW+
if ( (aux&BIT0) == 0){           //Atual: SW+ Fechada
    if (p_mais == ABERTA ){
        res |= BIT0;             //SW+ acionada
        p_mais=FECHADA;
    }
}
else{
    if (p_mais==FECHADA)
        p_mais=ABERTA;          //SW+ liberada
}

// Testar SW-
if ( (aux&BIT1) == 0){           //Atual: SW- Fechada
    if (p_menos == ABERTA ){
        res |= BIT1;            //SW- acionada
        p_menos=FECHADA;
    }
}
else{
    if (p_menos==FECHADA)
        p_menos=ABERTA;          //SW- liberada
}

// Testar SWC
if ( (aux&BIT2) == 0){           //Atual: SWC Fechada
    if (p_clear == ABERTA ){
        res |= BIT2;
        p_clear=FECHADA;
    }
}
else{
    if (p_clear==FECHADA)
        p_clear=ABERTA;          //SWC liberada
}

return res;
}

// Ler a porta do PCF
char i2c_read(void){
    char dado;
    UCB0CTL1 &= ~UCTR;              //Mestre RX
    UCB0CTL1 |= UCTXSTT;            //Gerar START
    while ( (UCB0CTL1 & UCTXSTT) == UCTXSTT);
    if ( (UCB0IFG & UCNACKIFG) == UCNACKIFG){ //NACK?
        P1OUT |= BIT0;              //NACK=Sinalizar problema
        while(1);                  //Travar execução
    }
}

```

```

}

UCB0CTL1 |= UCTXSTP; //Gerar STOP + NACK
while ( (UCB0IFG & UCRXIFG) == 0); //Esperar RX
dado=UCB0RXBUF;
while( (UCB0CTL1&UCTXSTP) == UCTXSTP);
return dado;
}

// Escrever dado na porta
void i2c_write(char dado){
    UCB0CTL1 |= UCTR | //Mestre TX
                UCTXSTT; //Gerar START
    while ( (UCB0IFG & UCTXIFG) == 0) ; //Esperar TXIFG=1
    UCB0TXBUF = dado; //Escrever dado
    while ( (UCB0CTL1 & UCTXSTT) == UCTXSTT) ; //Esperar STT=0
    if ( (UCB0IFG & UCNACKIFG) == UCNACKIFG){ //NACK?
        P1OUT |= BIT0; //NACK=Sinalizar problema
        while(1); //Travar execução
    }
    UCB0CTL1 |= UCTXSTP; //Gerar STOP
    while ( (UCB0CTL1 & UCTXSTP) == UCTXSTP) ; //Esperar STOP
    //delay(50);
}

// Testar o endereço adr para escrita
char i2c_test(char adr){
    UCB0I2CSA = adr; //Endereço a ser testado
    UCB0CTL1 |= UCTR; //Mestre TX --> escravo RX
    UCB0CTL1 |= UCTXSTT; //Gerar STASRT
    while ( (UCB0IFG & UCTXIFG) == 0); //Já iniciou o START
    UCB0CTL1 |= UCTXSTP; //Gerar STOP
    while ( (UCB0CTL1 & UCTXSTP) == UCTXSTP); //Esperar STOP
    if ((UCB0IFG & UCNACKIFG) == 0) return TRUE; //Chegou ACK
    else return FALSE; //Chegou NACK
}

// Configurar USCI_B0 como mestre
// P3.0 = SDA e P3.1 = SCL
void USCI_B0_config(void){
    UCB0CTL1 = UCSWRST; //Ressetar USCI_B1
    UCB0CTL0 = UCMST | //Modo Mestre
                UCMODE_3 | //I2C
                UCSYNC; //Síncrono
    //UCB0BRW = BR10K; //10 kbps
    UCB0BRW = BR100K; //100 kbps
    UCB0CTL1 = UCSSEL_3; //SMCLK e UCSWRST=0
    P3SEL |= BIT1 | BIT0; //Funções alternativas
    P3REN |= BIT1 | BIT0;
    P3OUT |= BIT1 | BIT0;
}

```

```

}

// Configurar I/O
void leds_config(void){
    P1DIR |= BIT0; P1OUT &= ~BIT0; //Led Vermelho
    P4DIR |= BIT7; P4OUT &= ~BIT7; //Led Verde
}

// Delay para depois do STOP
void delay (int x){
    volatile int i;
    for (i=0; i<x; i++);
}

```

**ER 9.9.** Neste exercício vamos exemplificar o uso da porta I<sup>2</sup>C para acessar o CI MPU-6050 (placa GY-521) que integrado um acelerômetro de 3 eixos e um giroscópio também de 3 eixos. Assim, este exercício pede para escrever um programa que apresente as leituras dos registradores de aceleração e giro.

### Solução:

O objetivo deste exercício é exemplificar o uso de um dispositivo I<sup>2</sup>C um pouco mais complexo, no caso, o MPU-6050. A placa com este acelerômetro da Invensense mais comum no Brasil é a GY-521, apresentada na figura abaixo e, ao lado, está uma sugestão para sua conexão ao MSP430 usando a porta USCI\_B0. Essa placa possui um regulador interno de 3,3 V, assim, pode ser alimentada com 5 V. Seus pinos são:

- VCC = 5V;
- GND = Terra;
- SCL e SDA = porta I<sup>2</sup>C do MSP;
- AD0 = endereço alternativo e
- INT = pino de interrupção.
- XDA e XCL = I<sup>2</sup>C externo, não são usados neste estudo.

O pino AD0 permite que se usem dois desses dispositivos num único barramento I<sup>2</sup>C. Quando solto, este pino AD0 vai para zero graças a um resistor de *pulldown* interno. Com um resistor de *pullup* externo é possível colocá-lo em nível alto e assim configurar a placa para operar no endereço alternativo.

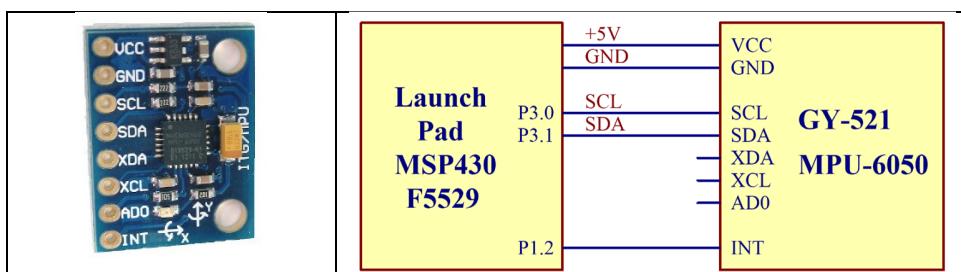


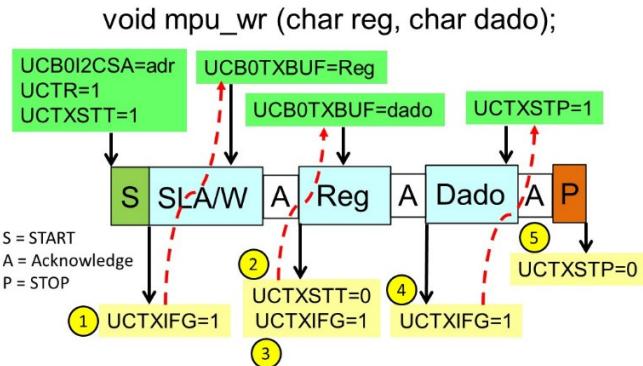
Figura 9.52. Foto da placa GY-521 e uma sugestão para sua conexão com a LaunchPad.

Este chip (MPU-6050) disponibiliza ao usuário um acelerômetro de 3 eixos (X, Y e Z) e um giroscópio de 3 eixos (X, Y e Z). Ele pertence à classe Unidade de Medida Inercial (IMU – *Inertial Motion Unit*) de 6 eixos. O acelerômetro trabalha nas escalas +/- 2 g, +/- 4 g, +/- 8 g e +/- 16 g. O giroscópio trabalha nas escalas de +/- 250 gr/s, +/- 500 gr/s, +/- 1000 gr/s e +/- 2000 gr/s (gr/s = graus/segundo). Os valores entregues são inteiros de 16 bits com sinal. Isto significa que varrem a faixa de -32.768 até 32.767 (65.536 passos). Por exemplo, supondo a escala de +/- 2g, a resolução é dada por 4 g / 65.536 (ou então, 2 g / 32.767).

Não é nosso objetivo estudar a fundo este dispositivo. O leitor é recomendado a buscar pelo manual e a estudar o [Apêndice E](#). A listagem abaixo apresenta o programa solução. Note que a função `i2c_test()`, já conhecida, foi usada logo no início do programa para verificar se o dispositivo responde ao endereço I<sup>2</sup>C. Caso ele responda, o registrador UCB0I2CSA é atualizado com a constante `MPU_ADR` e depois não é mais alterado.

O MPU6050 é operado através de registradores que podem ser lidos ou escritos. Vamos então descrever duas funções muito importantes, uma serve para escrever num registrador e a outra para ler um registrador. Abordaremos primeiro, por ser mais simples, a função que escreve um dado num registrador qualquer. Para este caso, o MPU deve ser endereçado como escravo receptor e em seguida deve-se enviar o endereço do registrador e depois o dado a ser escrito. A função responsável por toda esta operação é a `void mpu_wr(char reg, char dado)`.

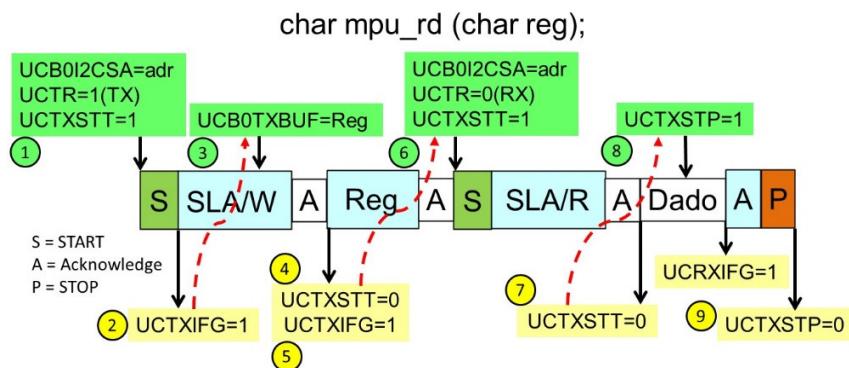
A figura 9.53 apresenta um diagrama de tempo detalhado, indicando com números todas as fases importantes para se escrever num registrador. Antes de iniciarmos a explicação, o leitor deve notar que a atualização do registrador UCB0I2CSA já foi feita. Tudo começa com a geração do START. Assim, que UCTXIFG vai a 1, é possível escrever no *buffer* de transmissão o endereço registrador a ser acessado. Quando UCTXSTT vai para zero, é porque o escravo já confirmou o endereçamento e já é possível verificar o ACK. Em seguida, espera-se novamente por UCTXIFG igual a 1 para escrever no *buffer* de transmissão o dado a ser enviado. Quando UCTXIFG for a 1 novamente, é porque a transmissão do último dado já foi iniciada e então já se pode gerar o STOP. Como última operação, espera-se UCTXSTP = 0, indicando então que o STOP já foi gerado.



*Figura 9.53. Diagrama de tempo indicando as etapas para se escrever um dado num determinado registrador do MPU-6050.*

A leitura de um registrador é um pouco mais longa. Para este caso, o MPU deve ser endereçado como escravo receptor para receber o endereço do registrador a ser lido. Em seguida ele é novamente endereçado (START repetido), agora como escravo transmissor para enviar o conteúdo do registrador endereçado na fase anterior. A função responsável por toda esta operação é a `char mpu_rd(char reg)`.

A figura 9.54 apresenta um diagrama de tempo detalhado desta operação, indicando com números todas as fases importantes. O início se parece com a operação anterior. O mestre transmissor gera um START e assim que detecta UCTXIFG igual a 1, escreve no buffer o endereço do registrador a ser acessado. Quando UCTXIFG vai a 1 pela segunda vez, significa que a transmissão deste endereço já foi iniciada. Neste ponto é pedido para o mestre receptor (UCTR=0) gerar um START (repetido), pois agora é preciso endereçar o escravo transmissor. Quando UCTXSTT vai para 0, é possível verificar se o escravo transmissor enviou um ACK. Em seguida já se pode solicitar o envio do STOP porque a recepção foi iniciada. Quando UCRXIFG vai a 1, se pode ler o dado enviado pelo MPU. Para terminar, espera-se a confirmação do STOP aguardando UCTXSTP=0.

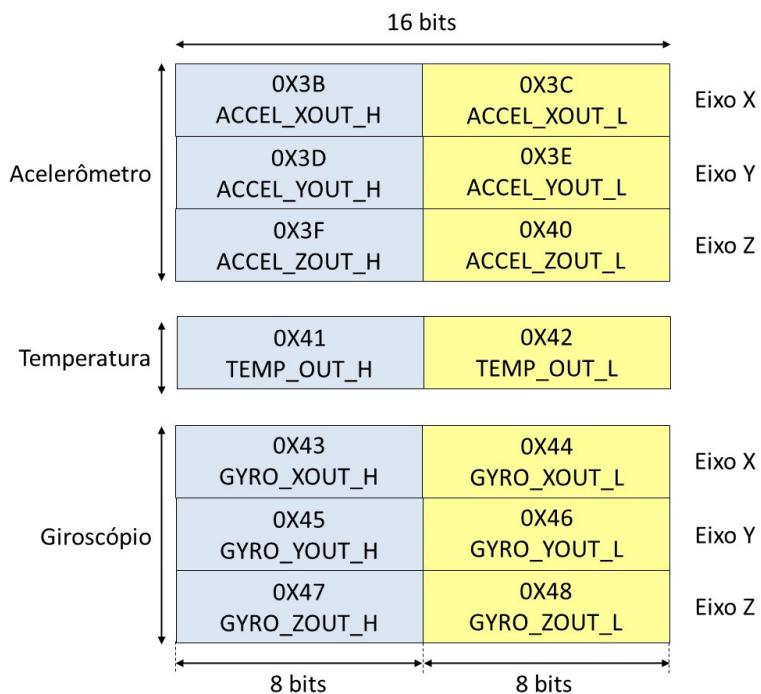


*Figura 9.54. Diagrama de tempo indicando as etapas para se ler um dado de um determinado registrador do MPU-6050.*

Para usar o MPU-6050 é preciso interpretar suas leituras. A Figura 9.55 apresenta os 14 registradores responsáveis pelas informações de aceleração e giro. Note que estão em sequência e que existe dois registradores para indicar a temperatura do *chip*. A temperatura é importante porque interfere no comportamento dos sensores. Esses registradores são lidos a partir do endereço 0x3B (ACCEL\_XOUT\_H). A interface do MPU avança seu ponteiro interno a cada leitura. Precisamos então de uma função para ler, em sequência, esses 14 registradores. Tal função está descrita a seguir:

```
void mpu_rd_vet(char reg, char *vt, char qtd),
```

onde *reg* é o registrador de início da sequência, *vt* é o vetor para receber as leituras e *qtd* é a quantidade de registradores a serem lidos. Terminada a leitura, é preciso montar as palavras de 16 bits, lembrando que o *byte* mais significativo veio primeiro. Esta operação é simples e está assinalada em abóbora na listagem solução, onde são atualizadas as variáveis *ax*, *ay*, *az*, *tp*, *gx*, *gy* e *gz*.



*Figura 9.55. Concatenação dos registradores para se obter as leituras do MPU-6050.*

A função `MPU_config()` faz a configuração do MPU-6050, onde é importante a seleção da escala de +/- 2g para o acelerômetro e +/- 250 graus/seg para o giroscópio. Conhecendo

as escalas, é possível calcular o valor em “g”, em graus Celsius ou em “graus/seg”, usando as equações abaixo. A Tabela 9.8 apresenta uma série de exemplos.

$$\text{Para a acelerômetro: } \text{Aceleração} = \frac{\text{valor (ax,ay ou az)}}{32767} \text{ g}$$

$$\text{Para o giroscópio: } \text{Giro} = \frac{\text{valor (gx,gy ou gz)}}{32767} \text{ graus/seg}$$

$$\text{Para a temperatura: } T_{\text{Celsius}} = \frac{\text{valor (tp)}}{340} + 36,53 \text{ }^{\circ}\text{C}$$

*Tabela 9.8. Exemplo de conversão das leituras para “g”, “graus/seg” ou “Celsius”*

<b>ax</b>	<b>ax</b>	<b>ay</b>	<b>az</b>	<b>tp</b>	<b>gx</b>	<b>gy</b>	<b>gz</b>
800	800	-1200	15000	-5678	-20000	500	700
0,048g	0,048 g	-0,073 g	0,916 g	19,83 °C	152 gr/s	3,81 gr/s	5,34 gr/s

gr = graus

Ao rodar o programa na LaunchPad, o leitor pode inserir um *break point* na primeira linha do laço principal para poder examinar os valores lidos. Pode ainda, mudar a propriedade deste *break point* para “refresh all windows”, com isto ele vai poder examinar os valores se alterando enquanto movimenta a placa.

Finalmente, note que este laço principal é quebrado quando a chave S1 é acionada. Este é um cuidado para evitar que um acesso seja interrompido, o que poderia colocar o MPU-6050 num estado indefinido. Por exemplo, imagine que o usuário mandou interromper o programa logo após o MPU ser acessado para a escrita. O MPU vai ficar eternamente esperando por um dado que nunca vai chegar. Esta espera deverá ser interrompida com uma condição de START, mas vamos evitar esse caso. Assim, antes de interromper a execução do programa, açãone a chave S1.

### *Listagem da solução do ER 9.9*

```
// ER 9.9
// Exemplo com MPU-6050 (GY-521)
// Usa USCI_B0: P3.0 = SDA e P3.1 = SCL
// Acionar S1 antes de interromper o programa

#include <msp430.h>

#define TRUE      1
#define FALSE     0
#define BR10K    105 // (SMCLK) 1.048.576/105 ~= 10kHz
```

```

#define BR100K 11 // (SMCLK) 1.048.576/11 ~= 100kHz

// MPU-6050 algumas constantes
#define MPU_ADR          0x68    //Endereço I2C do MPU
#define MPU_WHO          0x68    //Resposta ao Who am I
#define SMPLRT_DIV       0x19
#define CONFIG           0x1A
#define GYRO_CONFIG      0x1B
#define ACCEL_CONFIG     0x1C
#define ACCEL_XOUT_H     0x3B
#define PWR_MGMT_1        0x6B
#define WHO_AM_I         0x75    //Registrador Who am I

// Protótipo das funções
void mpu_config(void);
void mpu_rd_vet(char reg, char *vt, char qtd);
void mpu_wr(char reg, char dado);
char mpu_rd(char reg);
char i2c_test(char adr);
void USCI_B0_config(void);
void leds_s1_config(void);
void delay (int x);

int main(void){
    char who;
    char vetor[16];
    volatile int ax,ay,az,tp,gx,gy,gz;

    WDTCTL = WDTPW | WDTHOLD;    // stop watchdog timer
    leds_s1_config();           //Configurar Leds e S1
    USCI_B0_config();          //Configurar USCI_B0

    // MPU responde ao endereçamento com ACK?
    if (i2c_test(MPU_ADR) == FALSE){
        P1OUT |= BIT0;          //Sinalizar problema
        while(TRUE);            //Travar execução
    }
    UCB0I2CSA = MPU_ADR;        //Endereço a ser usado

    // Leitura do registrador WHO AM I retorna o esperado?
    if (mpu_rd(WHO_AM_I) != MPU_WHO){
        P1OUT |= BIT0;          //Sinalizar problema
        while(TRUE);            //Travar execução
    }
    mpu_config();               //Configurar MPU
    P4OUT |= BIT7;              //Verde aceso = OK

    while (TRUE){
        mpu_rd_vet(ACCEL_XOUT_H, vetor, 14);    //Ler 14 regs
        ax=vetor[ 0];   ax=(ax<<8)+vetor[ 1]; //aceleração eixo x
    }
}

```

```

        ay=vetor[ 2];    ay=(ay<<8)+vetor[ 3]; //aceleração eixo y
        az=vetor[ 4];    az=(az<<8)+vetor[ 5]; //aceleração eixo z
        tp=vetor[ 6];    tp=(tp<<8)+vetor[ 7]; //temperatura
        gx=vetor[ 8];    gx=(gx<<8)+vetor[ 9]; //giro eixo x
        gy=vetor[10];    gy=(gy<<8)+vetor[11]; //giro eixo y
        gz=vetor[12];    gz=(gz<<8)+vetor[13]; //giro eixo z
        if ( (P2IN&BIT1) == 0) break;           //S1 acionada?
    }
    P4OUT &= ~BIT7;      //Verde apagado
    while (TRUE);
    return 0;
}

// Configurar o MPU
void mpu_config(void){
    mpu_wr(PWR_MGMT_1, 1);      //Acodar e Relógio=PLL gx
    delay(100);                //Esperar acordar
    mpu_wr(CONFIG, 6);         //Taxa = 1 kHz, Banda=5Hz
    mpu_wr(SMPLRT_DIV, 9);     //Taxa de amostr. = 100 Hz
    mpu_wr(GYRO_CONFIG, 0);    // +/- 250 graus/seg
    mpu_wr(ACCEL_CONFIG, 0);   // +/- 2g
}

// Ler sequência de dados do MPU
void mpu_rd_vet(char reg, char *vt, char qtd){
    char i;

    // Indicar registrador de onde começa a leitura
    UCB0CTL1 |= UCTR | UCTXSTT;          //Mestre TX + Gerar START
    while ( (UCB0IFG & UCTXIFG) == 0);    //Esperar TXIFG=1
    UCB0TXBUF = reg;                     //Escrever registrador
    while ( (UCB0CTL1 & UCTXSTT) == UCTXSTT); //STT=0?
    if ( (UCB0IFG & UCNACKIFG) == UCNACKIFG){ //NACK?
        P1OUT |= BIT0;                  //NACK=problema
        while(1);                      //Travar execução
    }

    // Configurar escravo transmissor
    UCB0CTL1 &= ~UCTR;                  //Mestre RX
    UCB0CTL1 |= UCTXSTT;                //START Repetido
    while ( (UCB0CTL1 & UCTXSTT) == UCTXSTT); //STT=0?

    // Ler a quantidade de dados, menos o último
    for (i=0; i<qtd-1; i++){
        while ((UCB0IFG & UCRXIFG) == 0); //Esperar RX
        vt[i]=UCB0RXBUF;                 //Ler dado
    }

    // Ler o último dado e gerar STOP
    UCB0CTL1 |= UCTXSTP;               //Gerar STOP
}

```

```

        while ((UCB0IFG & UCRXIFG) == 0);           //Esperar RX
        vt[i]=UCB0RXBUF;                           //Ler dado
        while ((UCB0CTL1 & UCTXSTP) == UCTXSTP);   //Esperar STOP
    }

// Ler um registrador do MPU
char mpu_rd(char reg){
    UCB0CTL1 |= UCTR | UCTXSTT;           //Mestre TX + Gerar START
    while ((UCB0IFG & UCTXIFG) == 0);     //Esperar TXIFG=1
    UCB0TXBUF = reg;                     //Escrever registrador
    while ((UCB0CTL1 & UCTXSTT) == UCTXSTT); //STT=0?
    if ((UCB0IFG & UCNACKIFG) == UCNACKIFG){ //NACK?
        P1OUT |= BIT0;                   //NACK=problema
        while(1);                      //Travar execução
    }

    // Configurar escravo transmissor
    UCB0CTL1 &= ~UCTR;                  //Mestre RX
    UCB0CTL1 |= UCTXSTT;                //START Repetido
    while ((UCB0CTL1 & UCTXSTT) == UCTXSTT); //STT=0?
    UCB0CTL1 |= UCTXSTP;               //Gerar STOP
    while ((UCB0IFG & UCRXIFG) == 0);   //Esperar RX
    while ((UCB0CTL1 & UCTXSTP) == UCTXSTP); //Esperar STOP
    return UCB0RXBUF;
}

// Escrever num registrador do MPU
void mpu_wr(char reg, char dado){
    UCB0CTL1 |= UCTR | UCTXSTT;           //Mestre TX + START
    while ((UCB0IFG & UCTXIFG) == 0);     //TXIFG=1?
    UCB0TXBUF = reg;                     //Escrever dado
    while ((UCB0CTL1 & UCTXSTT) == UCTXSTT); //Esperar STT=0
    if ((UCB0IFG & UCNACKIFG) == UCNACKIFG){ //NACK?
        P1OUT |= BIT0;                   //NACK=problema
        while(1);                      //Travar execução
    }
    while ((UCB0IFG & UCTXIFG) == 0);     //TXIFG=1?
    UCB0TXBUF = dado;                   //Escrever dado
    while ((UCB0IFG & UCTXIFG) == 0);     //TXIFG=1?
    UCB0CTL1 |= UCTXSTP;                //Gerar STOP
    while ((UCB0CTL1 & UCTXSTP) == UCTXSTP); //Esperar STOP
}

// Testar o endereço adr para escrita
char i2c_test(char adr){
    UCB0I2CSA = adr;                   //Endereço a ser testado
    UCB0CTL1 |= UCTR;                 //Mestre TX --> escravo RX
    UCB0CTL1 |= UCTXSTT;              //Gerar STASRT
    while ((UCB0IFG & UCTXIFG) == 0); //TXIFG, já iniciou o START
    UCB0CTL1 |= UCTXSTP;              //Gerar STOP
}

```

```
while ( (UCB0CTL1 & UCTXSTP) == UCTXSTP);      //Esperar STOP
if ((UCB0IFG & UCNACKIFG) == 0) return TRUE;    //Chegou ACK
else                                         return FALSE; //Chegou NACK
}

// Configurar USCI_B0 como mestre
// P3.0 = SDA e P3.1 = SCL
void USCI_B0_config(void){
    UCB0CTL1 = UCSWRST;           //Ressetar USCI_B0
    UCB0CTL0 = UCMST   |         //Modo Mestre
                UCMODE_3 |         //I2C
                UCSYNC;          //Síncrono
    //UCB0BRW = BR10K;           //10 kbps
    UCB0BRW = BR100K;           //100 kbps
    UCB0CTL1 = UCSSEL_3;         //SMCLK e UCSWRST=0
    P3SEL |= BIT1 | BIT0;       //Funções alternativas
    P3REN |= BIT1 | BIT0;       //Pullup
    P3OUT |= BIT1 | BIT0;       //Pullup
}

// Configurar I/O
void leds_s1_config(void){
    P1DIR |= BIT0; P1OUT &= ~BIT0; //Led Vermelho
    P4DIR |= BIT7; P4OUT &= ~BIT7; //Led Verde
    P2DIR &= ~BIT1;             //Chave S1
    P2REN |= BIT1;              //Chave S1
    P2OUT |= BIT1;              //Chave S1
}

// Gerar atrasos
void delay (int x){
    volatile int i;
    for (i=0; i<x; i++);
}
```

## 9.10. Exercícios Propostos

Apresentamos a seguir uma lista com diversos exercícios para que o leitor pratique o que foi estudado sobre I<sup>2</sup>C. É claro que para exercícios diferentes, necessitaremos de alguns outros dispositivos I<sup>2</sup>C. A lista é grande. Sempre que possível, verifique sua solução no LaunchPad.

**EP 9.1.** Vamos usar a unidade USCI\_B1 para projetar um escravo mais sofisticado que o usado no ER 9.1, mas ainda no endereço 42. Vamos denominá-lo de Escravo A. Este escravo retorna o complemento a 2 do último número de 16 bits recebido. Por exemplo, se neste escravo for escrito 0x1A2B, as próximas leituras deverão sempre retornar 0xE5D5. Na partida, se for feita uma leitura sem que nada tenha sido escrito, o escravo retorna 0xFFFF. Use a unidade USCI\_B0 para testá-lo.

**EP 9.2.** Usando a unidade USCI\_B1 para proponha e teste o Escravo B, que ainda usa o endereço 42. Este escravo tem 10 registradores de 16 bits. Os registradores estão, respectivamente, nos endereços internos R0, R1, ..., R9. O mestre pode escrever ou ler qualquer um desses registradores, individualmente ou em sequência.

A Figura 9.55 indica o protocolo e um exemplo para escrever num desses registradores. Note que o escravo é endereçado como receptor e são enviados o endereço do registrador (um número de 0 até 9) e depois o MSB e o LSB do número que se deseja armazenar.

Protocolo:	START	0x42/W	Reg	MSB	LSB	STOP
Exemplo:	START	0x42/W	5	0x12	0x34	STOP

*Figura 9.56. Protocolo usado para escrever num registrador do Escravo B e um exemplo para se escrever 0x1234 no registrador R5.*

A Figura 9.57 indica o protocolo e um exemplo para se ler um desses registradores. Note que o escravo é endereçado como receptor para receber o endereço do registrador (um número de 0 até 9) a ser lido. Em seguida, usando um START Repetido, o escravo é endereçado agora como transmissor para enviar o conteúdo do registrador previamente selecionado.

Protocolo	START	0x42/W	Reg	START	0x42/R	MSB	LSB	STOP
Exemplo	START	0x42/W	5	START	0x42/R	0x12	0x34	STOP

*Figura 9.57. Protocolo usado para ler um registrador do Escravo B e exemplo da leitura do registrador R5 que retornou 0x1234. Note o uso do START Repetido.*

Para dar mais flexibilidade, tanto a escrita quanto a leitura podem acessar os registradores em sequência. Por exemplo, na Figura 9.56, se fosse enviada mais uma palavra de 16 bits, ela seria grava no registrador 6 e assim por diante, de forma cíclica dentro da faixa R0, R1, ..., R9, R0, R1, ... . Na leitura exemplificada na Figura 9.57, após ler o registrador 5, seria lido o 6 e assim por diante.

**EP 9.3.** Vamos sofisticar o exercício anterior e criar o Escravo C. Adicione recursos no Escravo B, de forma a permitir que se façam as 4 operações com quaisquer registradores, segundo a sintaxe abaixo. Deve ser usada a representação em complemento a 2. As operações saturam em +32.767 ou em -32.768. A Figura 9.58 indica o protocolo para se usar as operações. A ordem dos registradores é importante. A Tabela 9.10 lista as possíveis operações.

Protocolo:	START	0x42/W	Rm	<b>sinal</b>	Rn	STOP
Exemplo:	START	0x42/W	5	*	6	STOP

Figura 9.58. Protocolo para operar dois registradores ( $Rm = Rm \text{ } \mathbf{sinal} \text{ } Rn$ ) e exemplo para a operação  $R5$  e  $R6$ , no caso  $R5 = R5 * R6$ .

Tabela 9.10. Descrição das 4 operações com os registradores do Escravo C

Operação	Símbolo	ASCII	Exemplo
Soma	+	0x2B	$Rm = Rm + Rn$
Subtração	-	0x2D	$Rm = Rm - Rn$
Multiplicação	*	0x2A	$Rm = Rm * Rn$
Divisão	/	0x2F	$Rm = Rm / Rn$

Observação: Devemos prever uma sinalização de erro caso de uma divisão por zero. Neste caso o led vermelho deve acender e o registrador que deveria receber o resultado deve ser carregado com o código 1202 (um mero capricho).

Curiosidade: Por que o código 1202? De onde ele saiu? Dica: 20 de julho do século passado.

**EP 9.4.** Este exercício propõe que a LaunchPad, usando as conexões indicadas abaixo, execute o jogo “Alunissagem com HP-25”. Este jogo está descrito no Apêndice H.

Programe o Escravo D para executar o jogo *moonlanding*. Como escravo receptor ele recebe a quantidade (**q**) de combustível a ser queimada e como escravo transmissor ele retorna o resultado da queima indicada, ou seja, a velocidade (**v**, 1 byte), a altitude (**h**, 2 bytes) e a quantidade (**f**) restante de combustível, como mostrado na Figura 9.59. O led verde acende caso o pouso tenha sucesso e o vermelho no caso de fracasso (*crash*). Mesmo no caso de falha, é possível consulta o Escravo D, porém ele não aceita mais comandos de queima de combustível.

Escrita	START	0x42/W	q	STOP	-	-	-
Leitura	START	0x42/R	v	MSB(h)	LSB(h)	f	STOP

Figura 9.59. Protocolo usado para interação com o Escravo D, que executa o jogo Moonlanding.

Dica: Em algum ponto do programa principal, que vai usar o Escravo D para jogar o Moonlanding, coloque um ponto de quebra (break point) para alterar o valor de queima a ser enviado. Logo após o escravo deve ser consultado para verificar as novas condições.

**EP 9.5.** Este exercício e os próximos fazem uso do Apêndice E que explica a máquina Enigma e apresenta alguns modelos simplificados. Vamos agora ao criar o Escravo E (E de Enigma) que opera por I<sup>2</sup>C.

Refaça o EP 2.44. Para testar, crie dois vetores, denominados de `char claro[]` e `char cifro[]`. O vetor `claro` contém a mensagem a ser cifrada e o vetor `cifro`, o resultado da cifragem. Use os recursos do CCS para verificar a correção da cifragem. É importante lembrar que o Enigma é uma máquina simétrica. Isto significa que se o vetor `cifro` for usado como entrada, ela retorna o vetor `claro`.

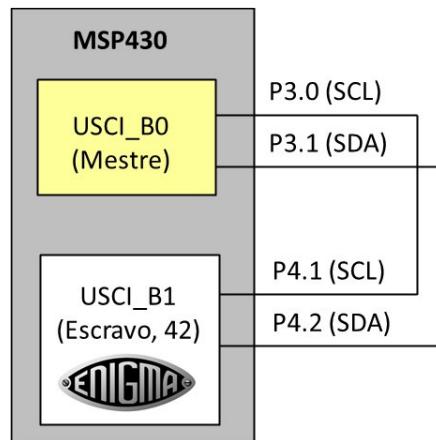


Figura 9.60. Esquema para a simulação da máquina Enigma.

**EP 9.6.** Se o leitor tiver a disponibilidade de um módulo Bluetooth HC-05, o exercício pode ficar mais interessante. Faça as conexões indicadas na figura 9.61. Usando as mesmas especificações do exercício anterior, programe a LaunchPad para simular a máquina Enigma 1. Será preciso usar um terminal serial no PC. O usuário digita o texto em claro e recebe de volta o texto cifrado.

**EP 9.7.** Usando as mesmas especificações, do EP 9.5, resolva o EP 2.47.

Sugestão: o leitor pode resolver antes os exercícios EP 2.45 e EP 2.46, já que eles têm um menor nível de dificuldade. Se tiver disponibilidade do HC-05, tente o esquema da Figura 9.61.

**EP 9.8.** Usando as mesmas especificações, do EP 9.5, resolva o EP 2.59, que pede a construção completa de uma máquina Enigma. Se tiver disponibilidade do HC-05, tente o esquema da Figura 9.61.

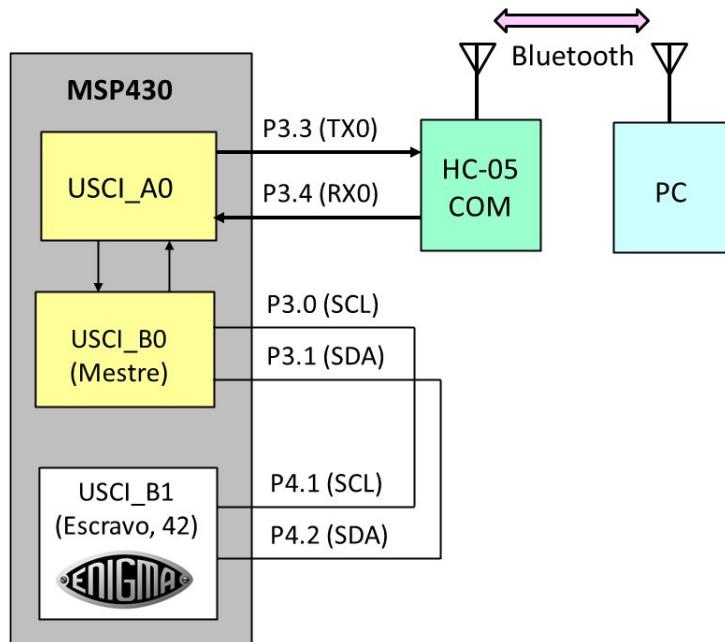


Figura 9.61. Esquema para que o PC faça uso de um canal Bluetooth para acessar o escravo Enigma.

**Colocar com um apêndice**  
**Gabarito para configurar os registradores da USCI\_Bx**

*Registradores de 8 bits*

	7	6	5	4	3	2	1	0
<b>UCBxCTL0</b>	UCA10	UCSLA10	UCMM	-	UCMST	UCMODE		UCSYNC
				0				
<b>UCBxCTL1</b>	UCSSEL		-	UCTR	UCTXNACK	UCTXSTP	UCTXSTT	UCWRST
			0					
<b>UCBxSTAT</b>	-	UCSLLOW	UCGC	UCBBUSY	-	-	-	-
	0				0	0	0	0
<b>UCBxIE</b>	-	-	UCNACKIE	UCALIE	UCSTPIE	UCSTTIE	UCTXIE	UCRXIE
	0	0						
<b>UCBxIFG</b>	-	-	UCNACKIFG	UCALIFG	UCSTPIFG	UCSTTIFG	UCTXIFG	UCRXIFG
	0	0						
<b>UCBxRXBUF</b>								
<b>UCBxTXBUF</b>								
<b>UCBxIV</b>								

*Registradores de 16 bits*

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<b>UCBxBRW</b>	Divisor para gerar o Baud Rate															
<b>UCBxI2COA</b>	UCGEN	-	-	-	-	-	-	-	-	-	-	-	I2COA			

		0	0	0	0	0	
<b>UCBxI2CSA</b>	-	-	-	-	-	-	I2CSA